What Is Parallel Computing?

Attempt to speed solution of a particular task by

1. Dividing task into sub-tasks
2. Executing sub-tasks simultaneously on multiple processors

Successful attempts require both

1. Understanding of where parallelism can be effective
2. Knowledge of how to design and implement good solutions

Methodology

Study problem, sequential program, or code segment

Look for opportunities for parallelism

Try to keep all processors busy doing useful work

Ways of Exploiting Parallelism

- Domain decomposition
- Task decomposition
- Pipelining

Domain Decomposition

First, decide how data elements should be divided among processors

Second, decide which tasks each processor should be doing

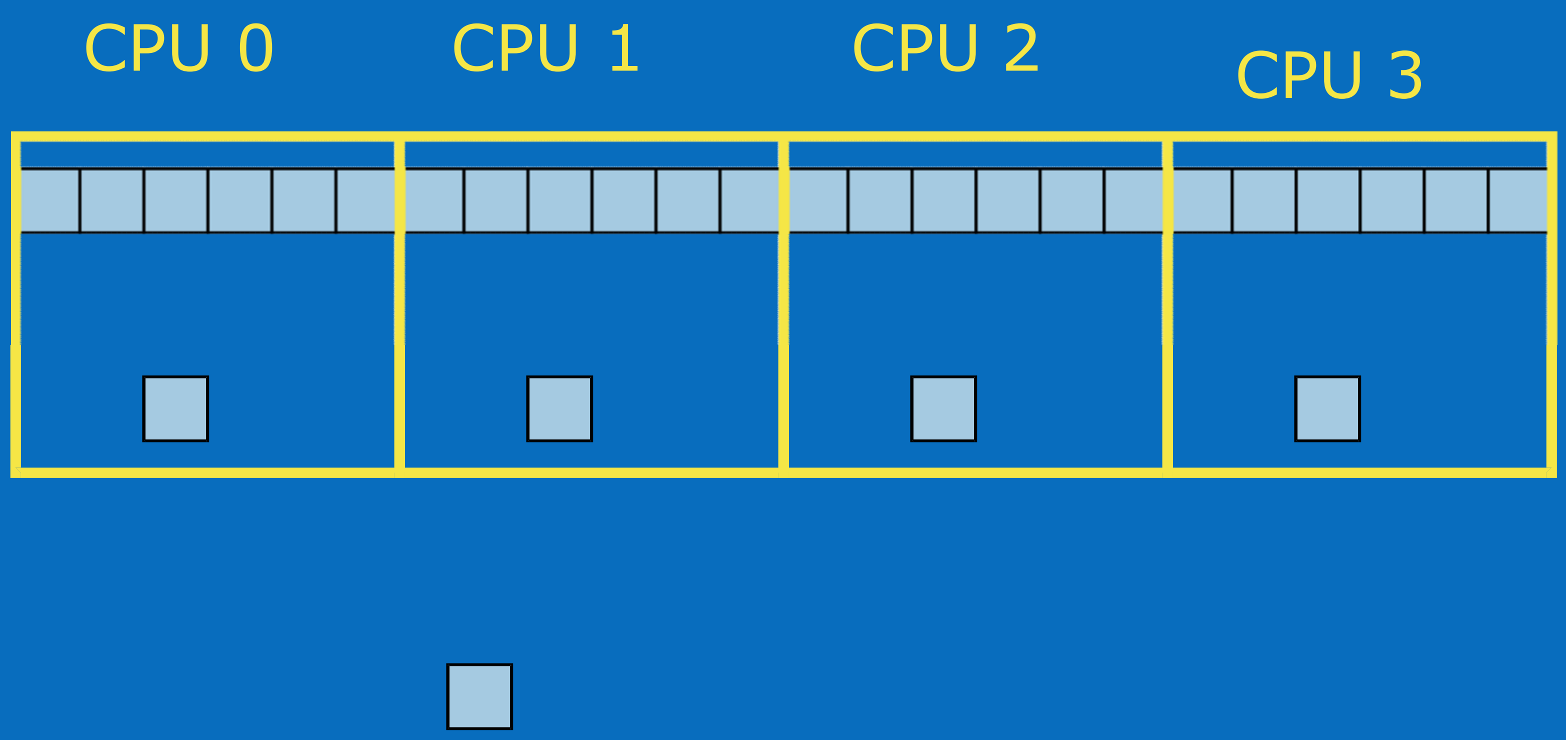
Example: Vector addition

Domain Decomposition

Find the largest element of an array

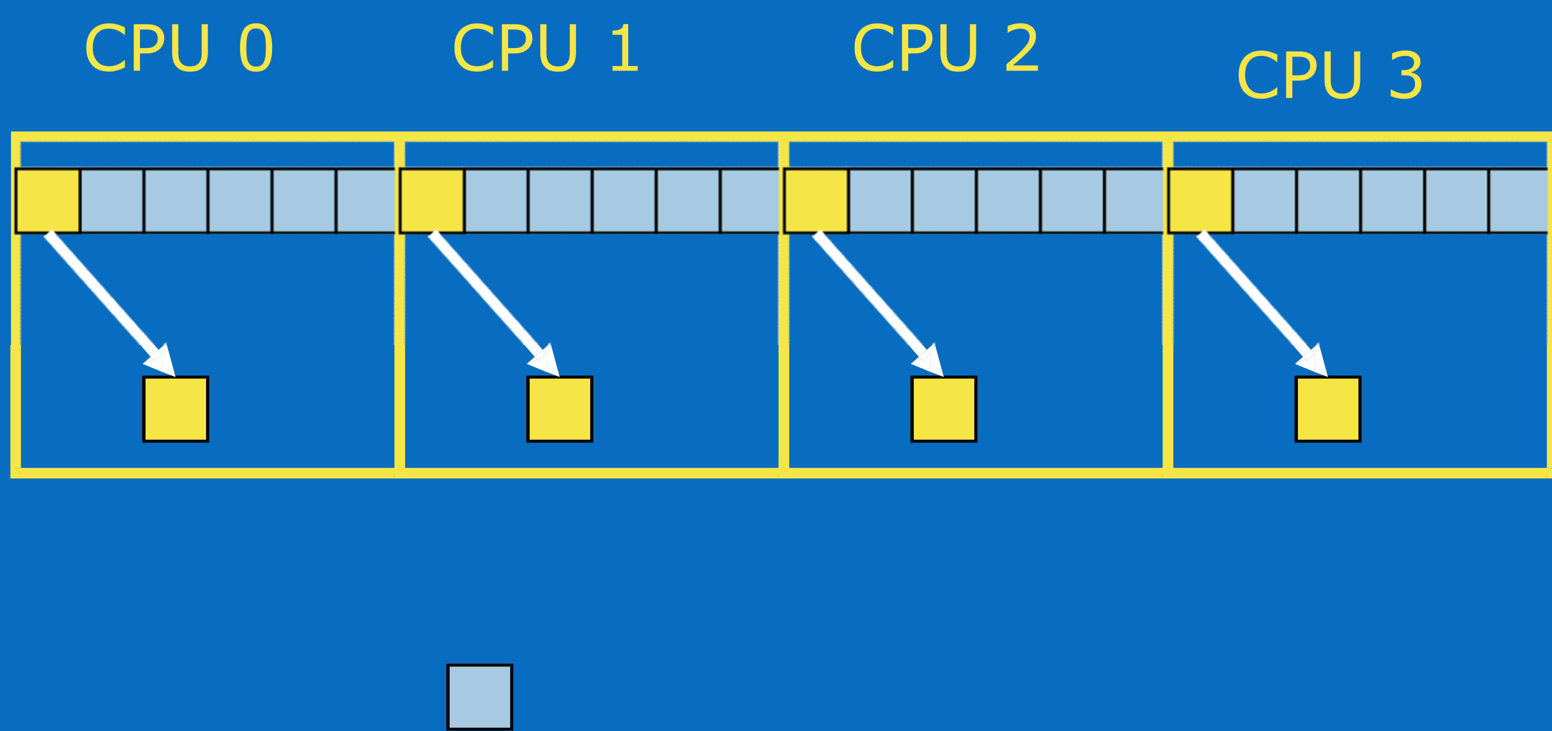
Domain Decomposition

Find the largest element of an array



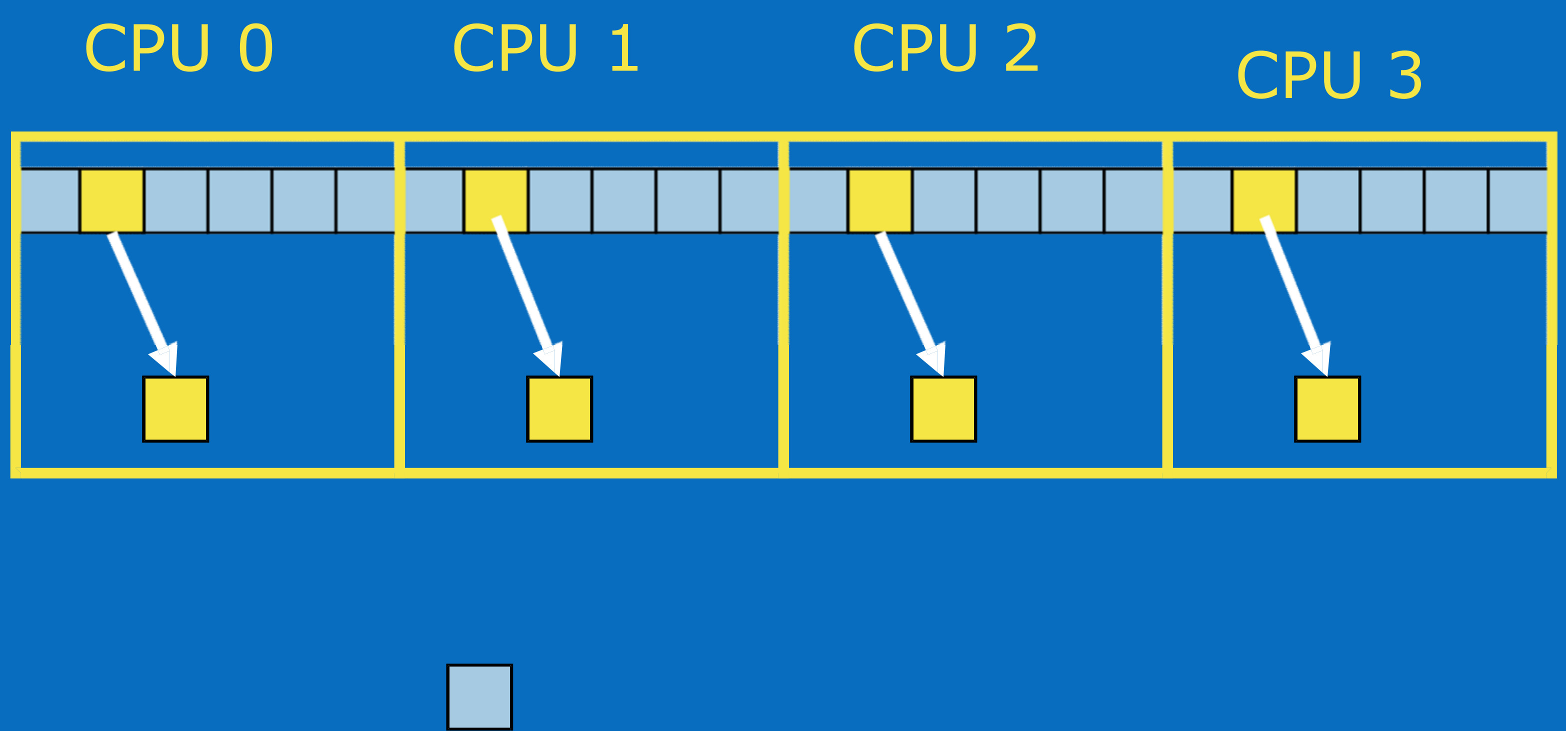
Domain Decomposition

Find the largest element of an array



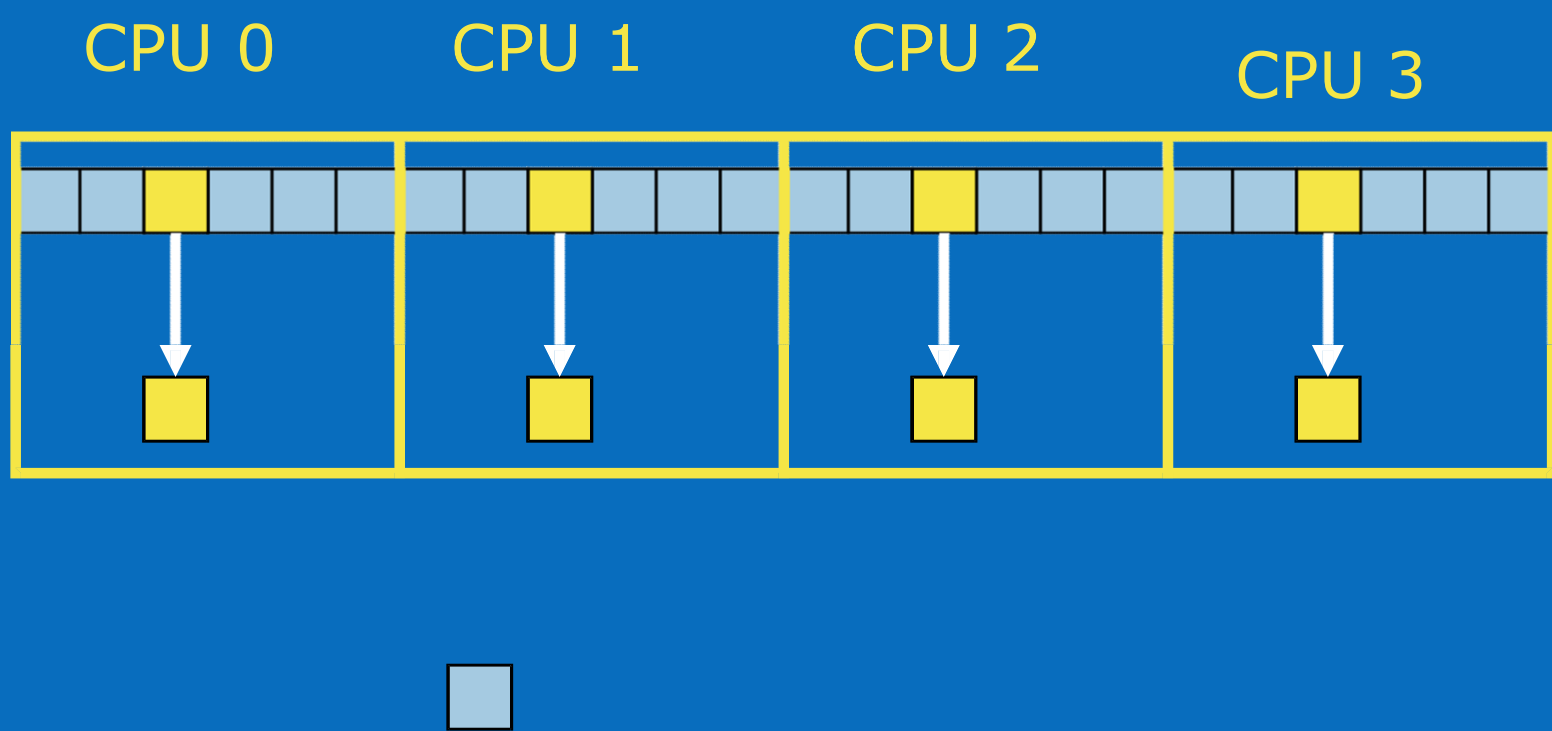
Domain Decomposition

Find the largest element of an array



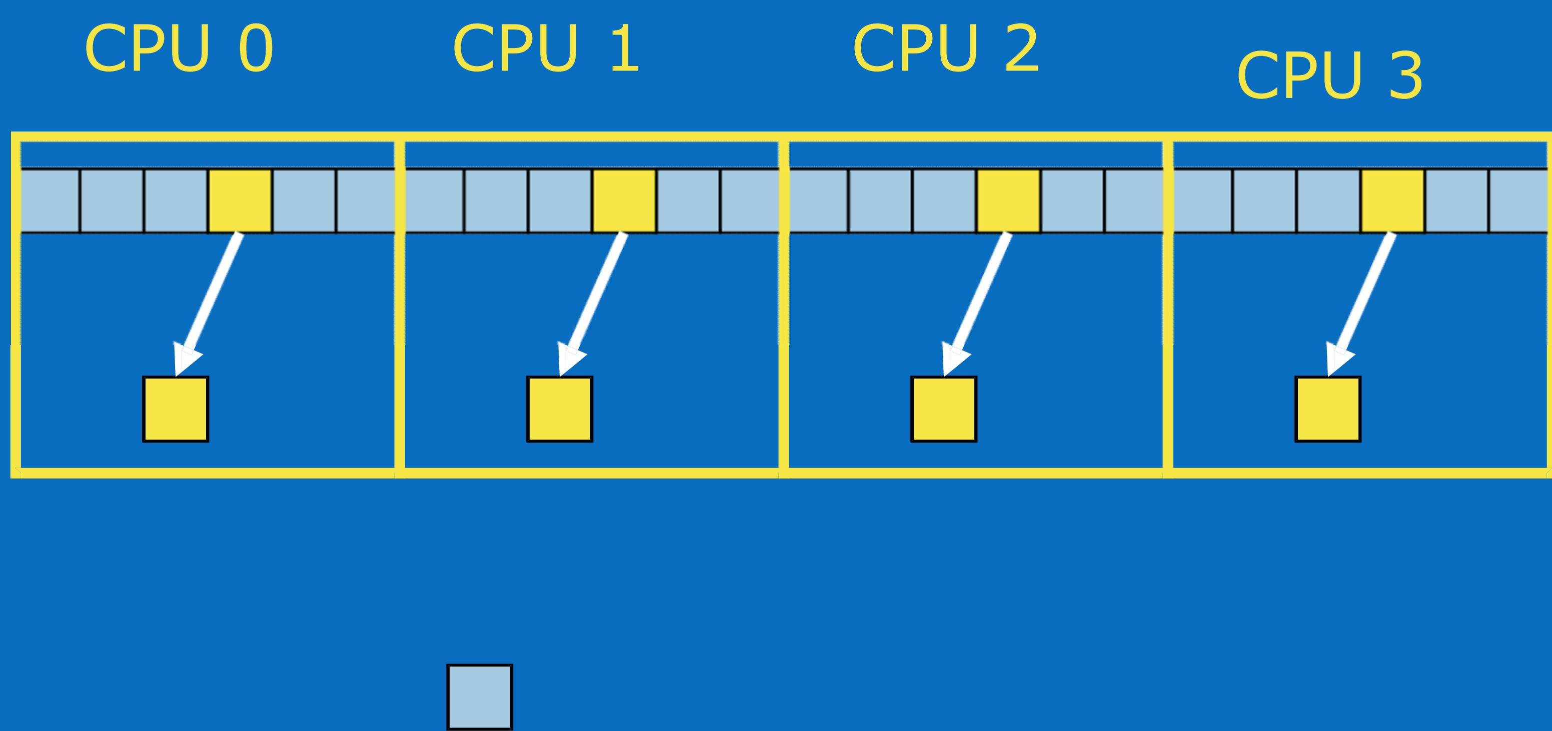
Domain Decomposition

Find the largest element of an array



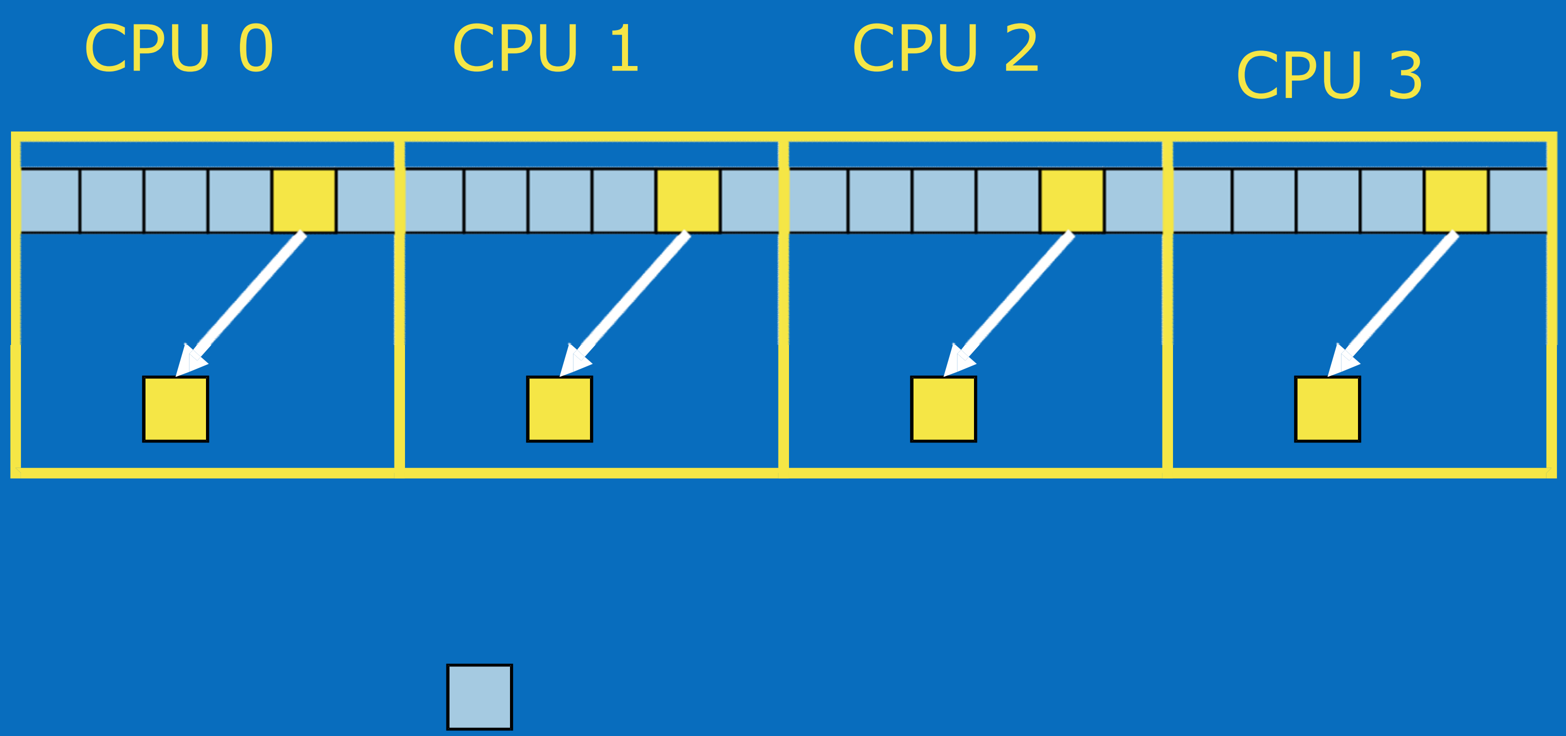
Domain Decomposition

Find the largest element of an array



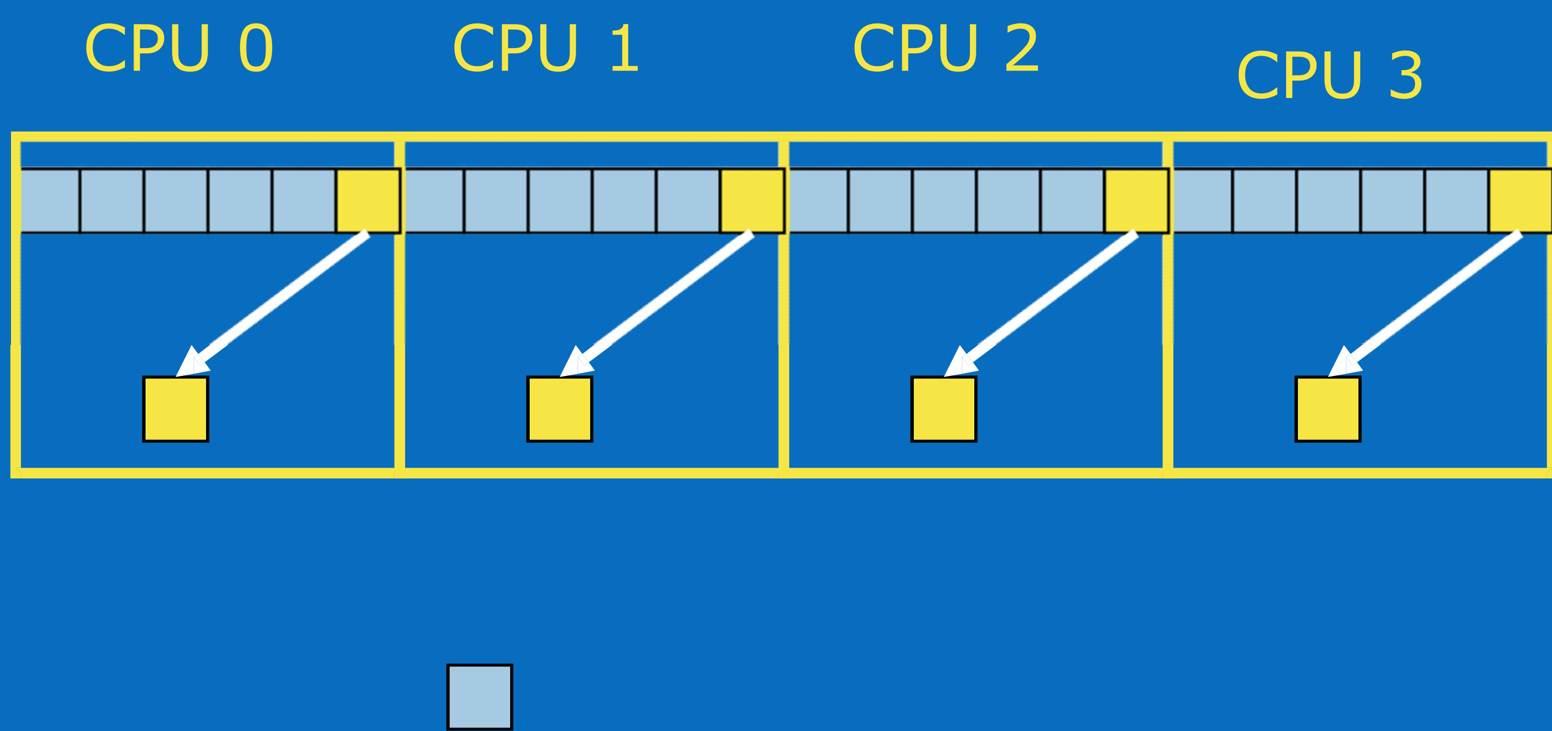
Domain Decomposition

Find the largest element of an array



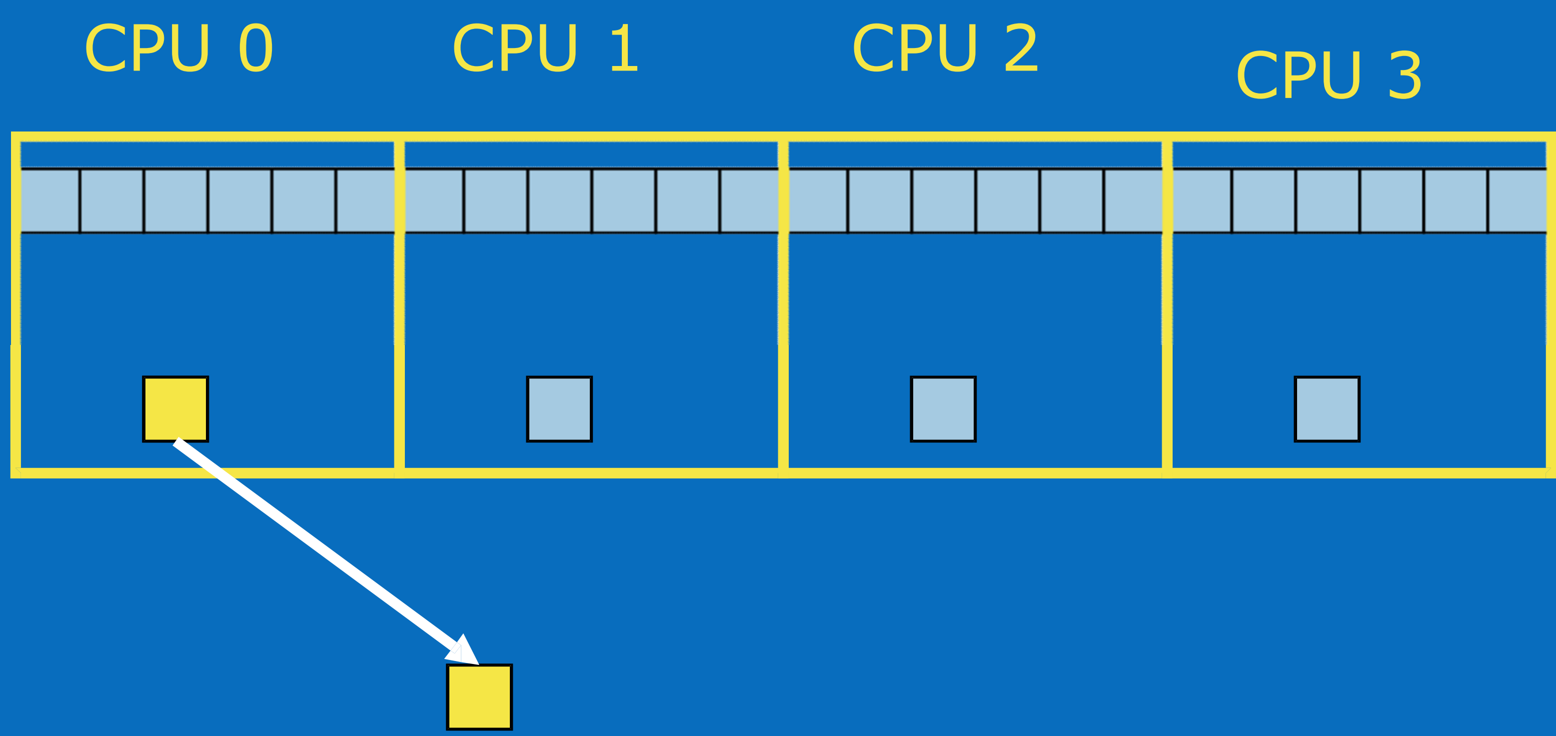
Domain Decomposition

Find the largest element of an array



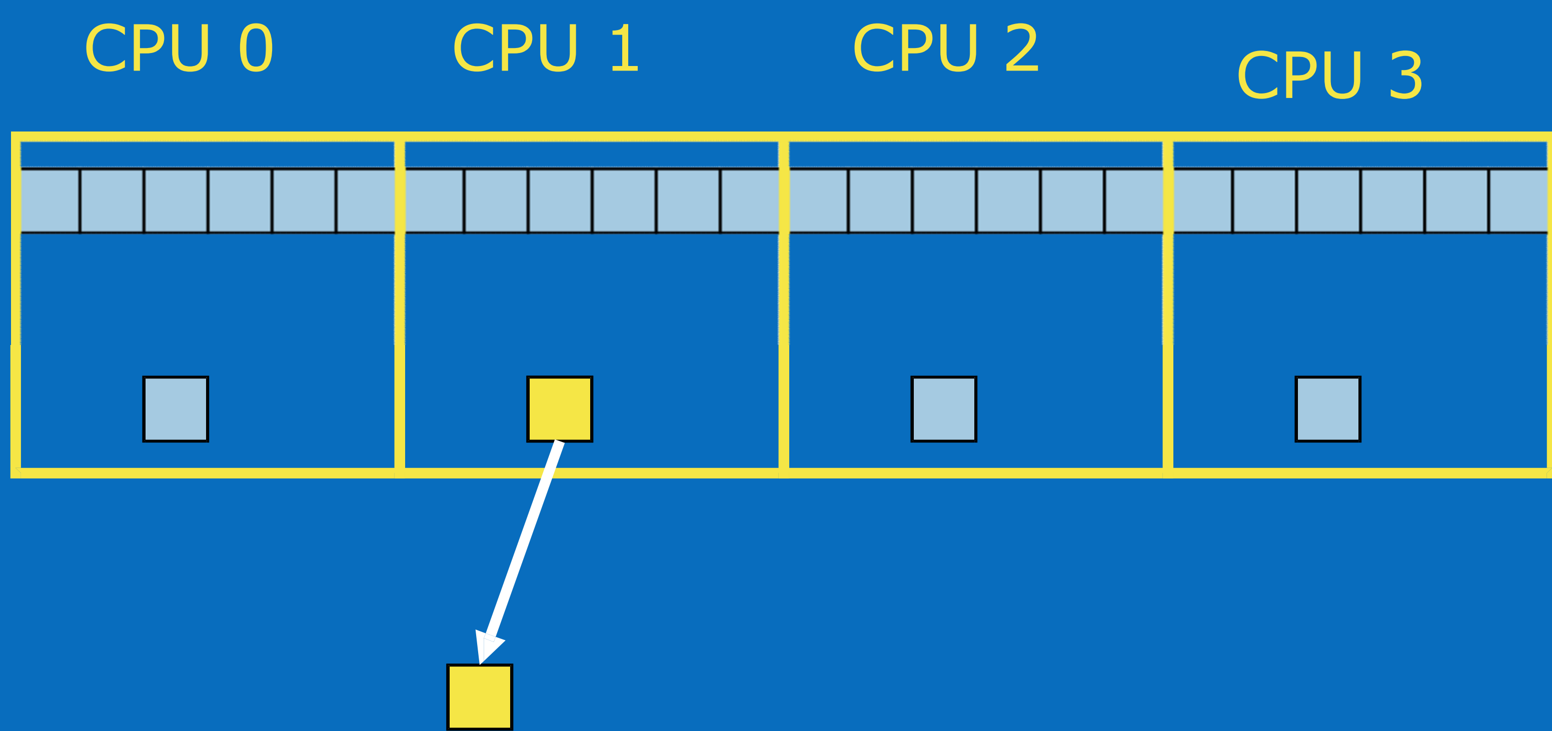
Domain Decomposition

Find the largest element of an array



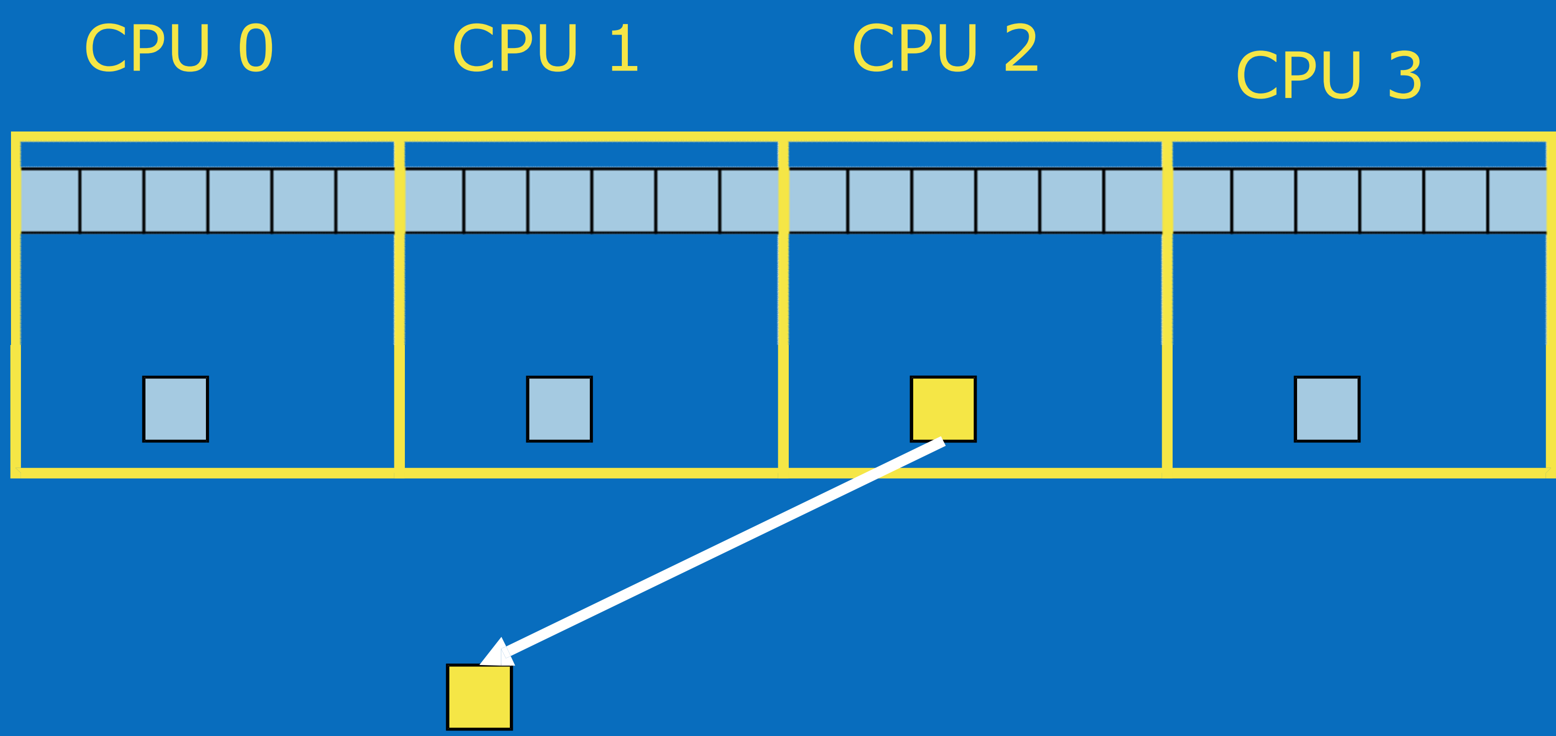
Domain Decomposition

Find the largest element of an array



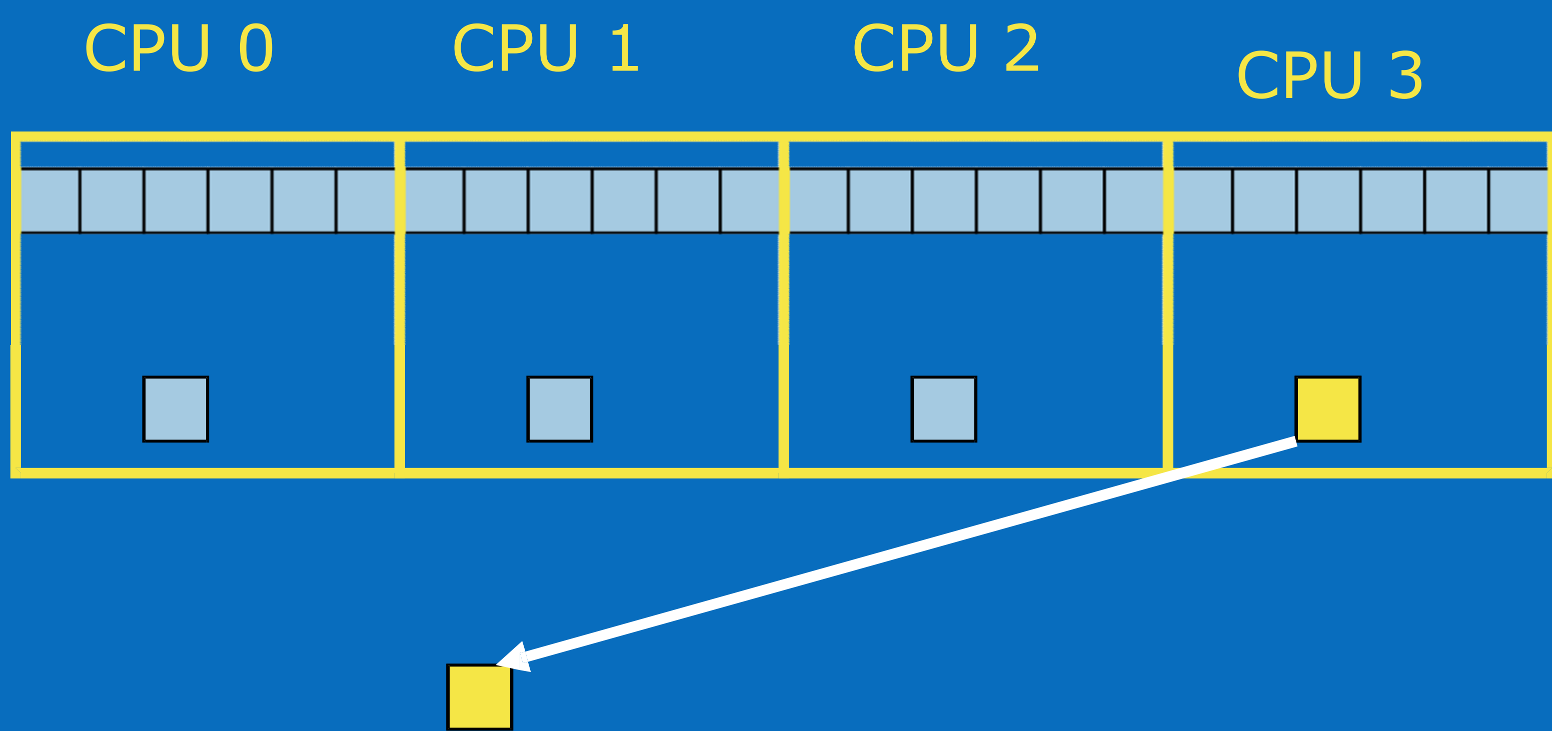
Domain Decomposition

Find the largest element of an array



Domain Decomposition

Find the largest element of an array



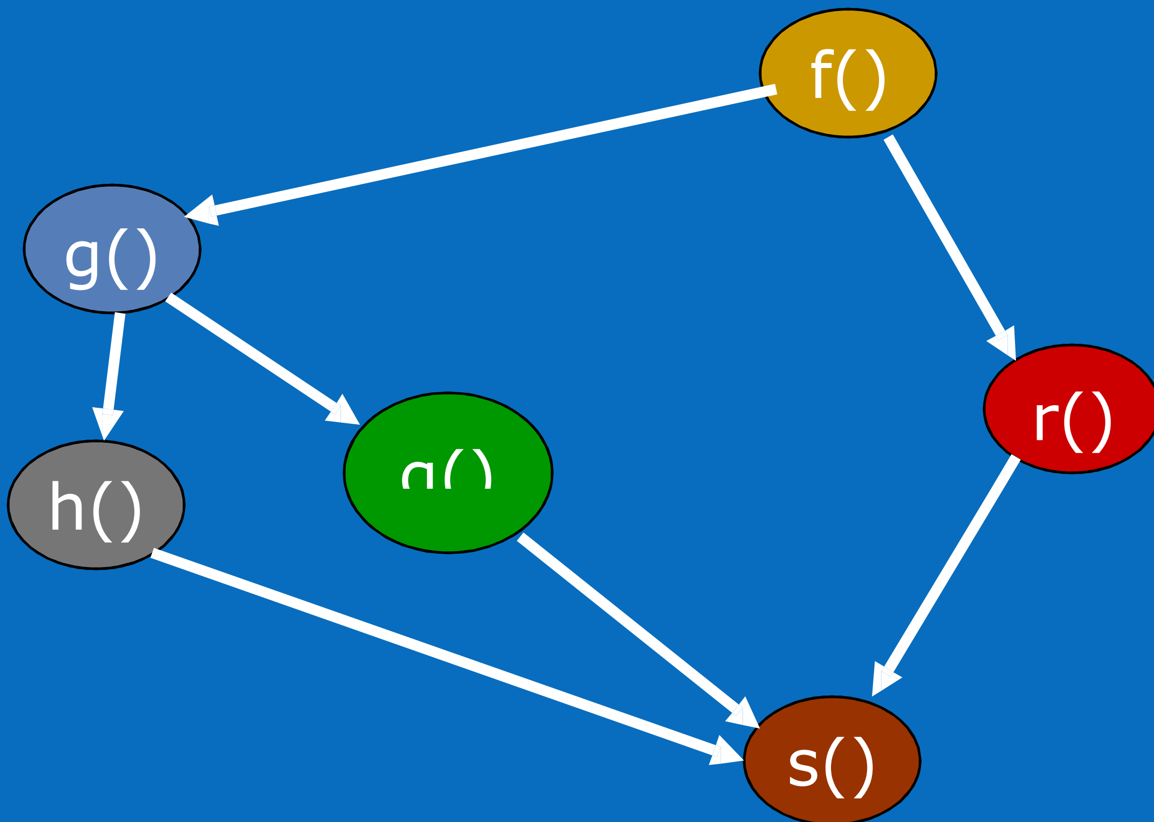
Task (Functional) Decomposition

First, divide tasks among processors

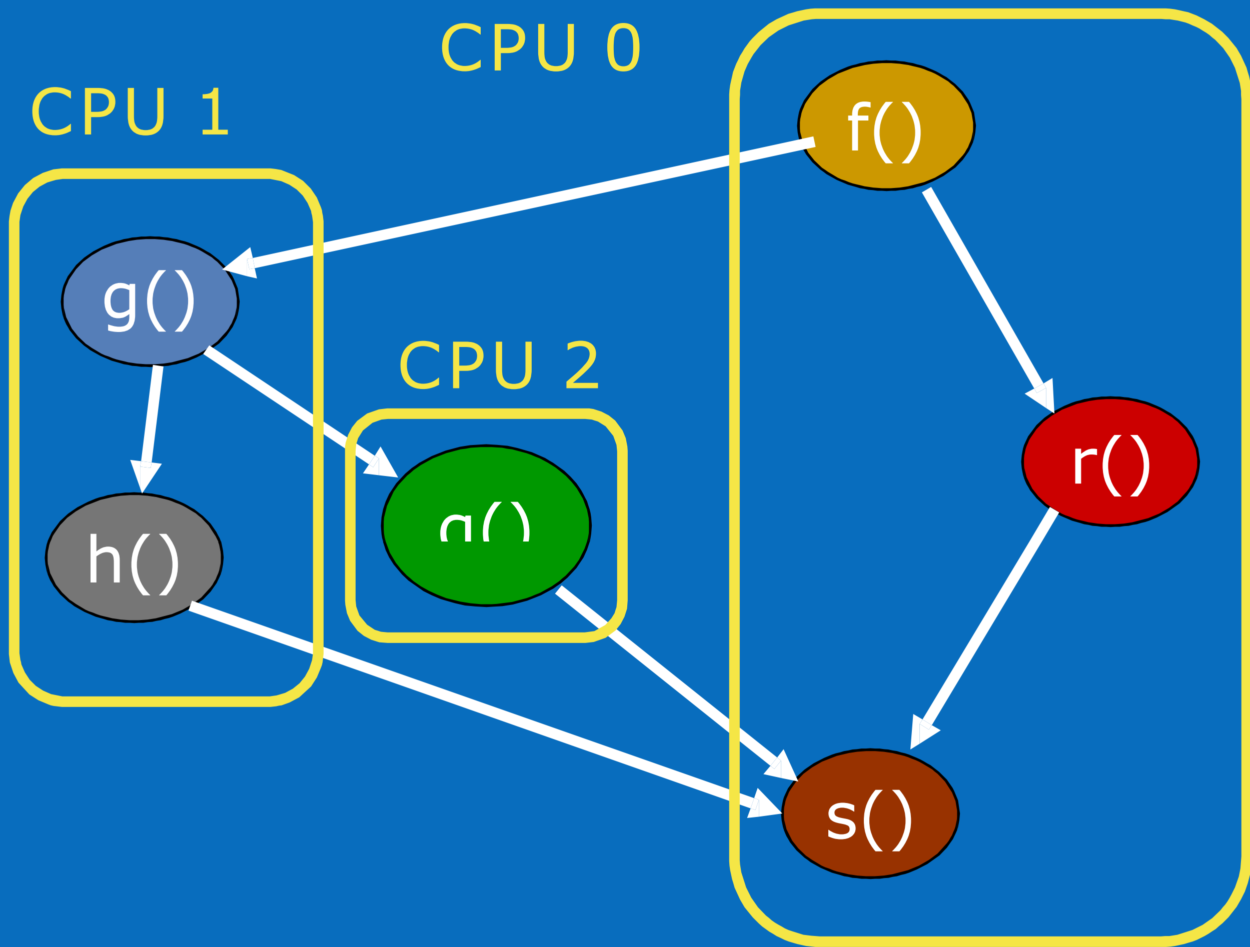
Second, decide which data elements are going to be accessed (read and/or written) by which processors

Example: Event-handler for GUI

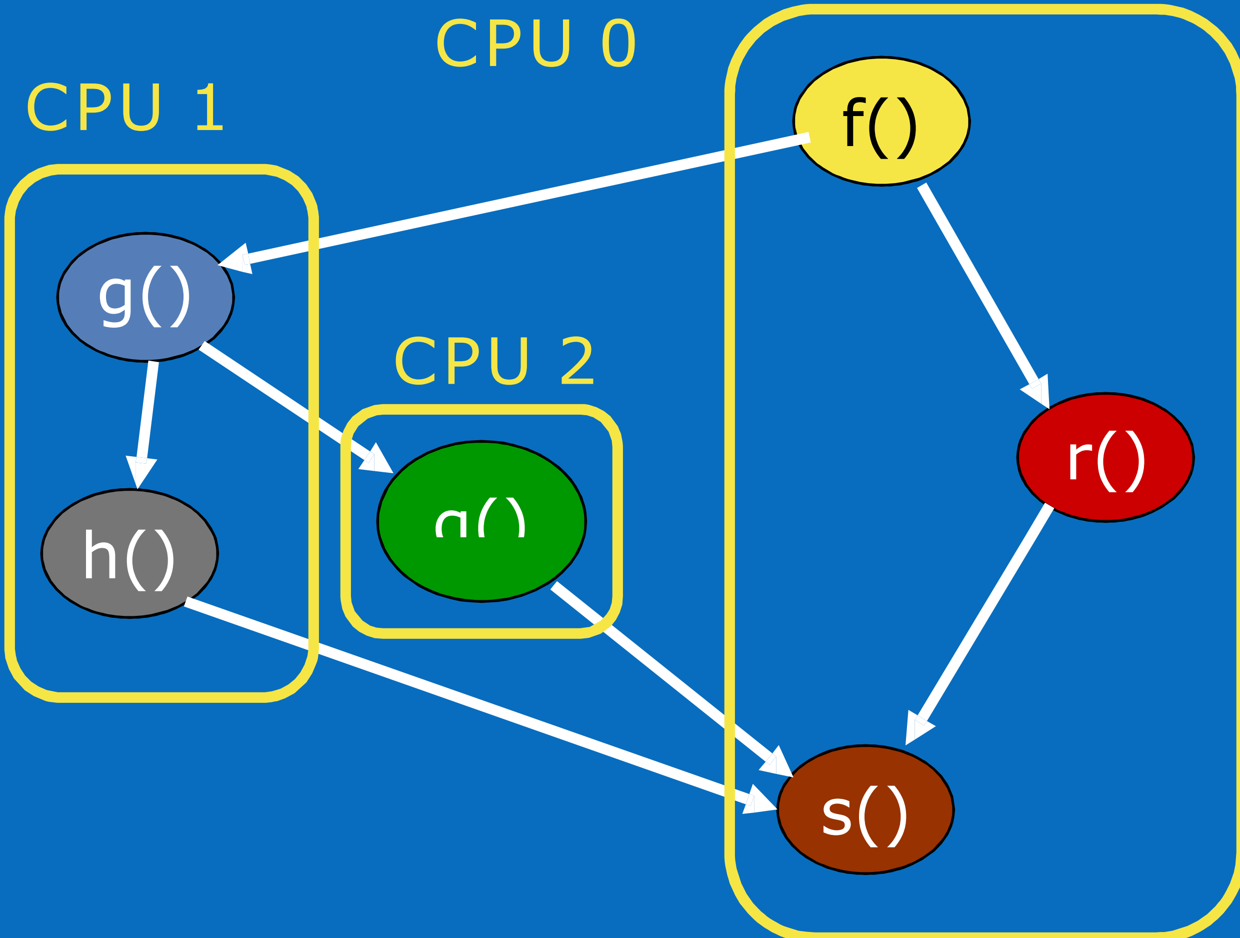
Task Decomposition



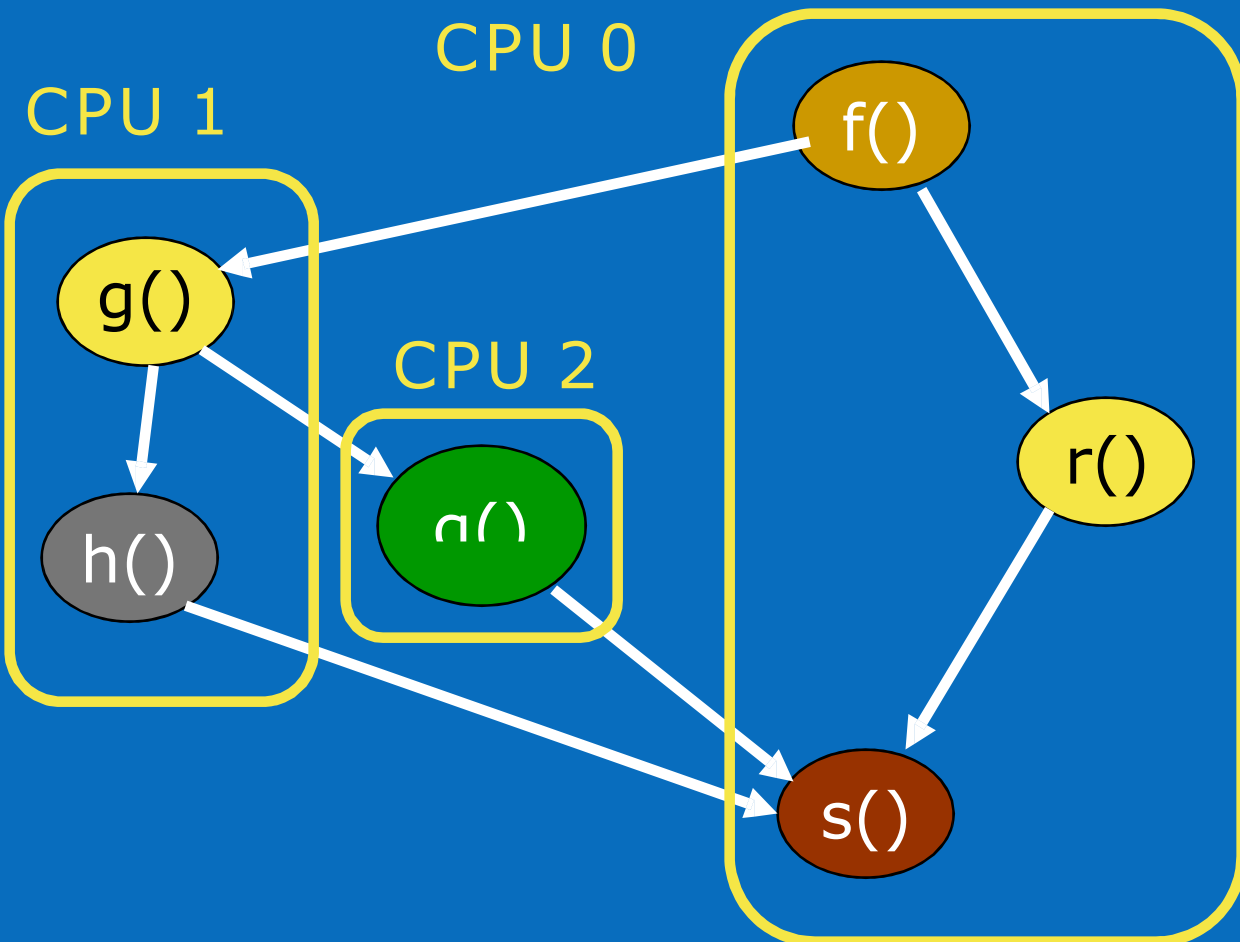
Task Decomposition



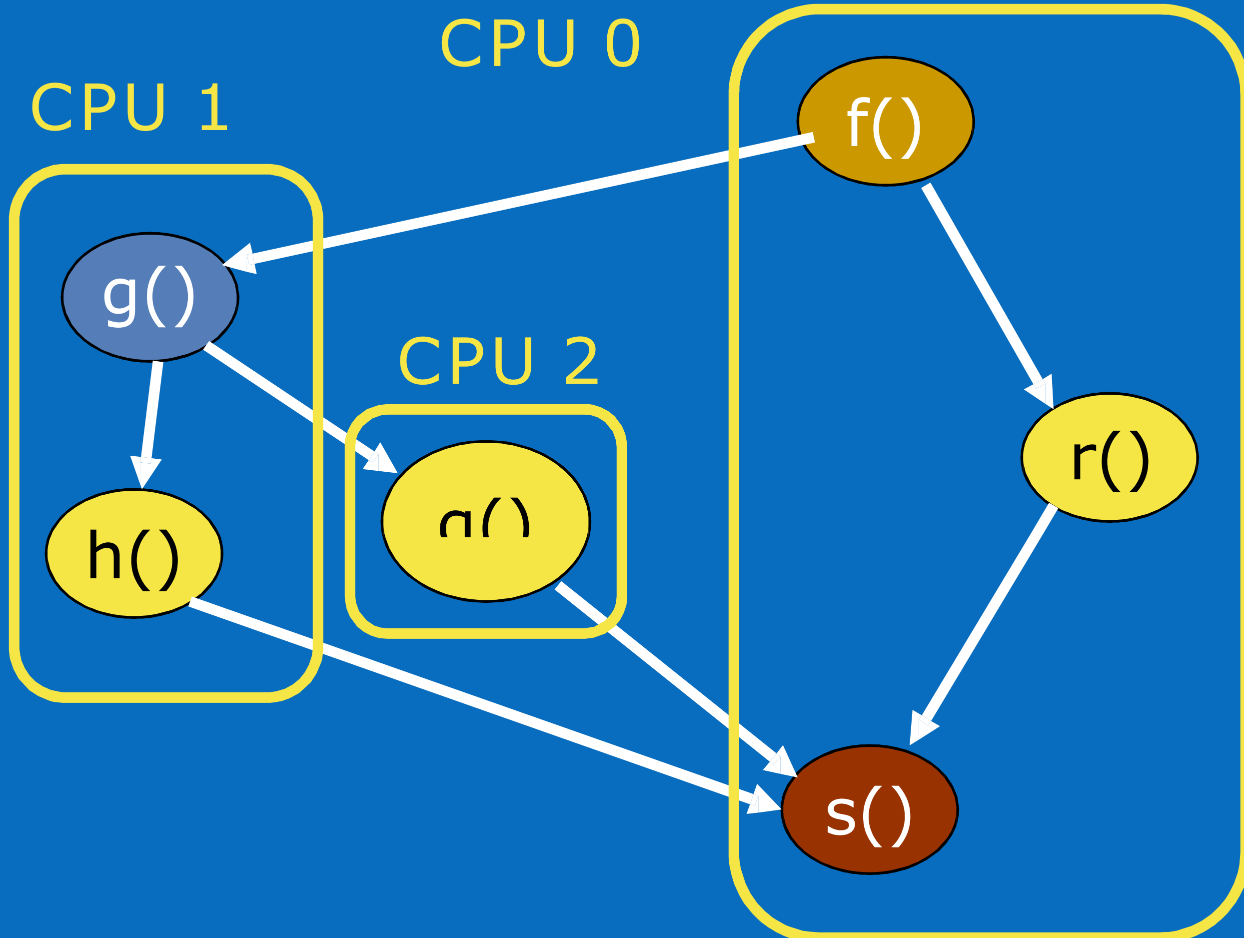
Task Decomposition



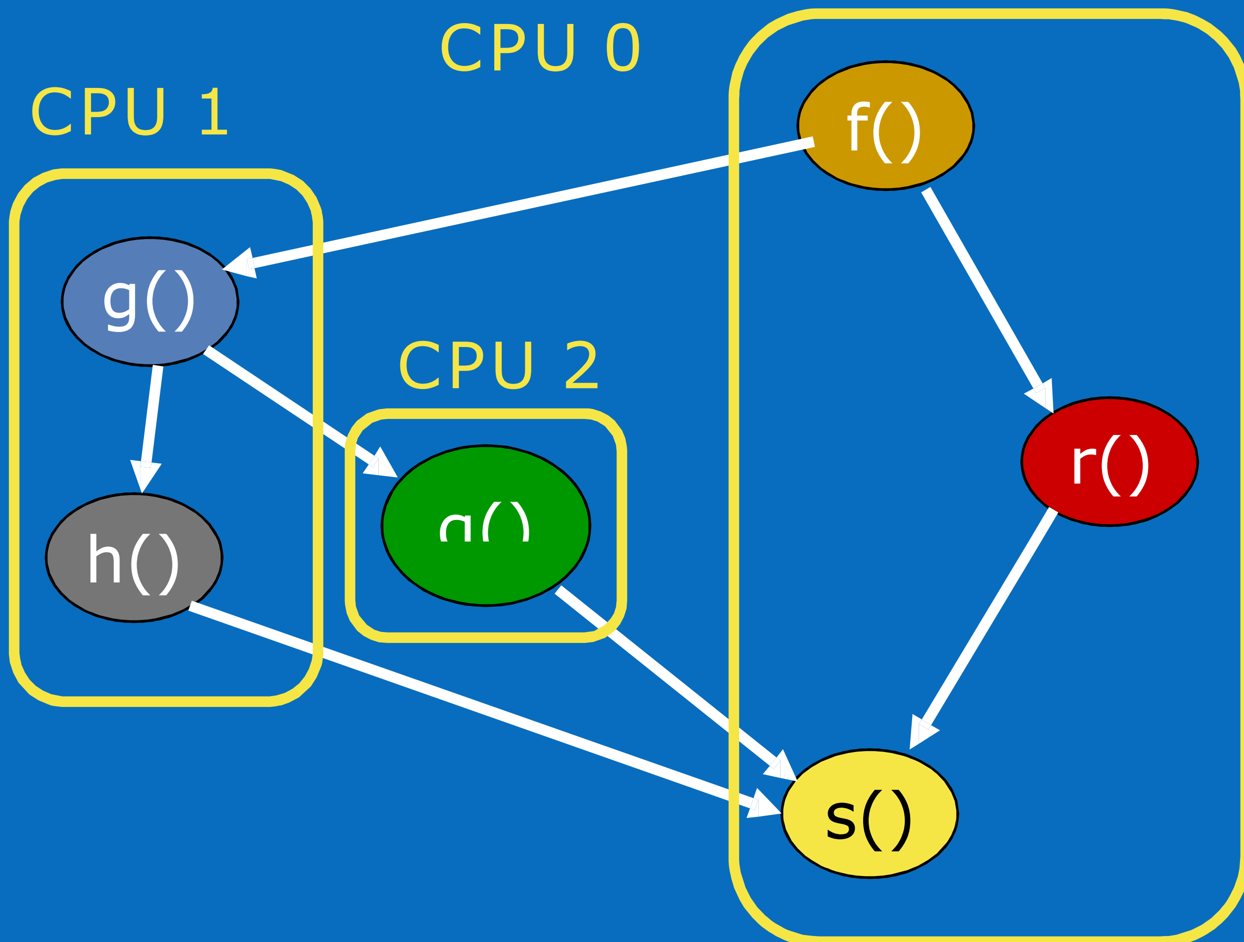
Task Decomposition



Task Decomposition



Task Decomposition

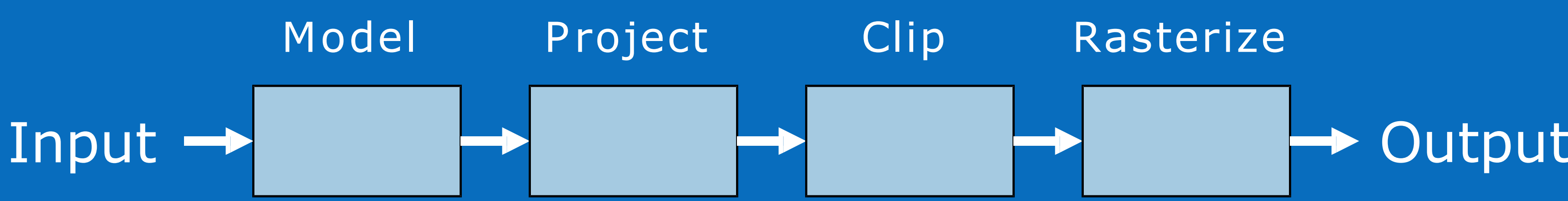


Pipelining

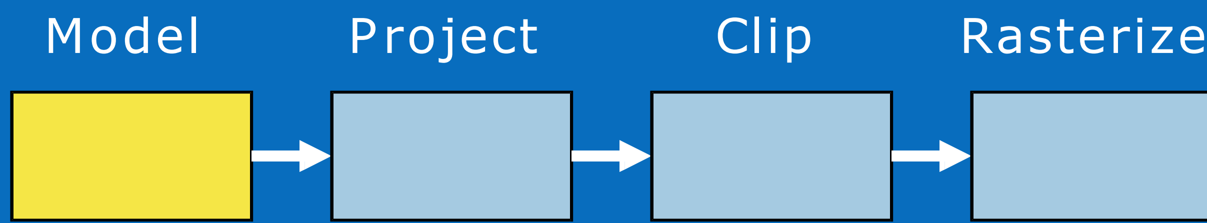
Special kind of task decomposition

“Assembly line” parallelism

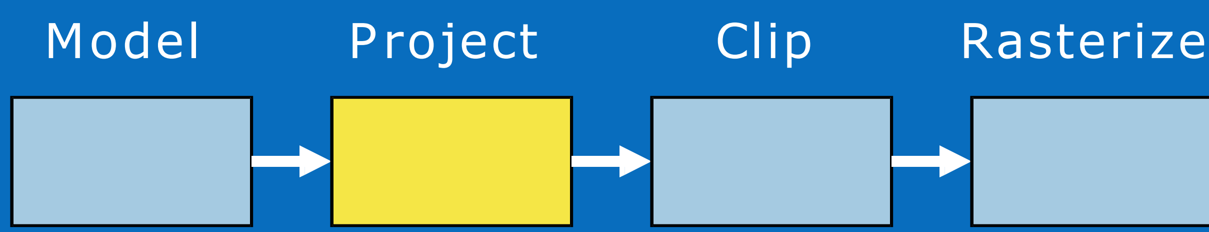
Example: 3D rendering in computer graphics



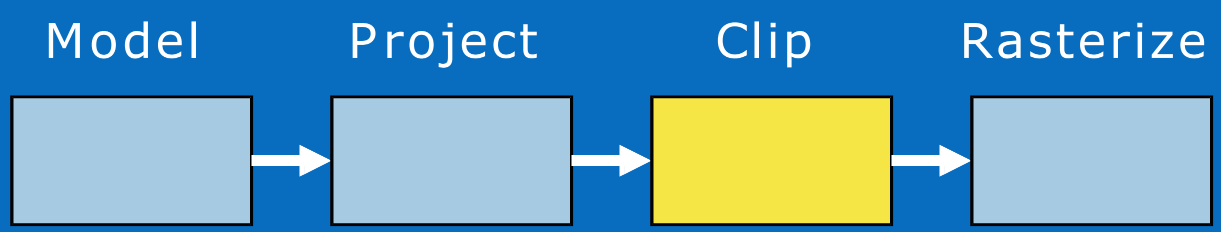
Processing One Data Set (Step 1)



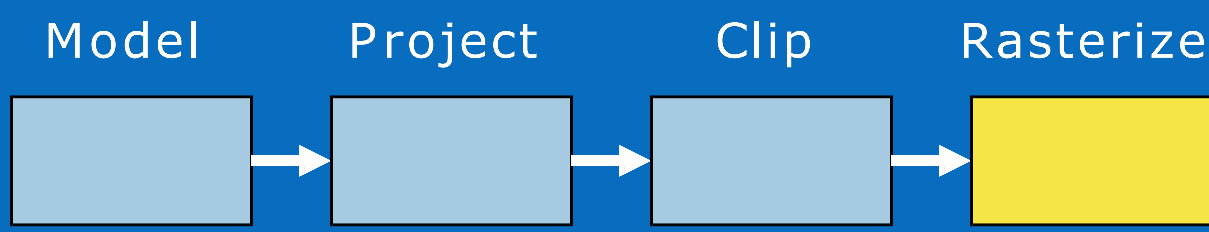
Processing One Data Set (Step 2)



Processing One Data Set (Step 3)

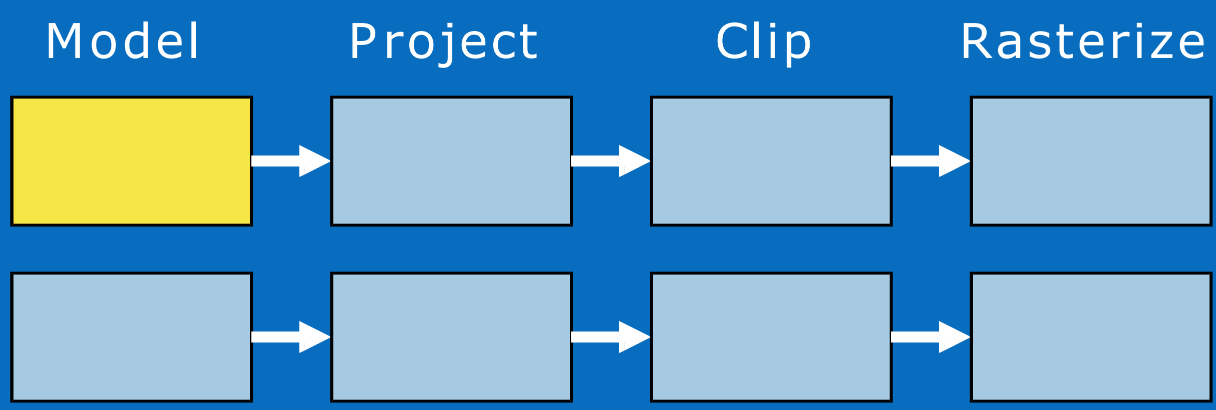


Processing One Data Set (Step 4)

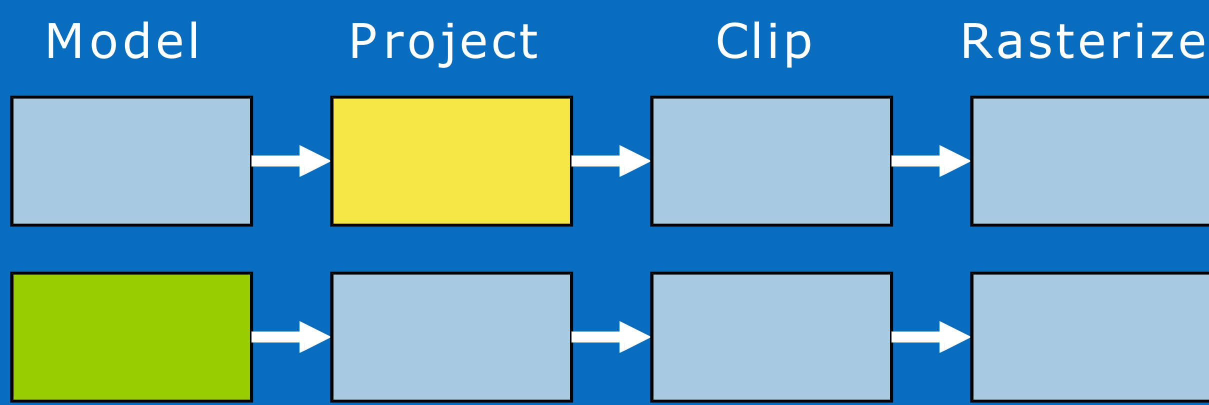


The pipeline processes 1 data set in 4 steps

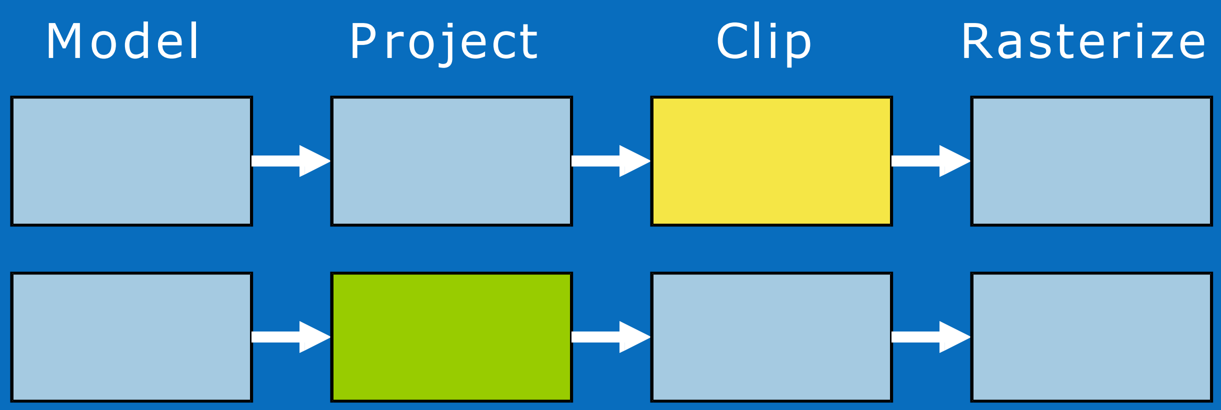
Processing Two Data Sets (Step 1)



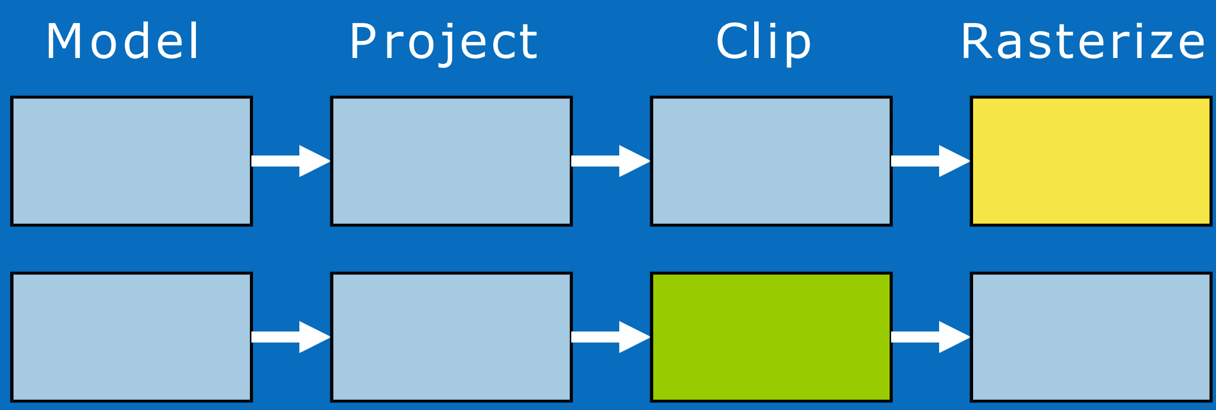
Processing Two Data Sets (Time 2)



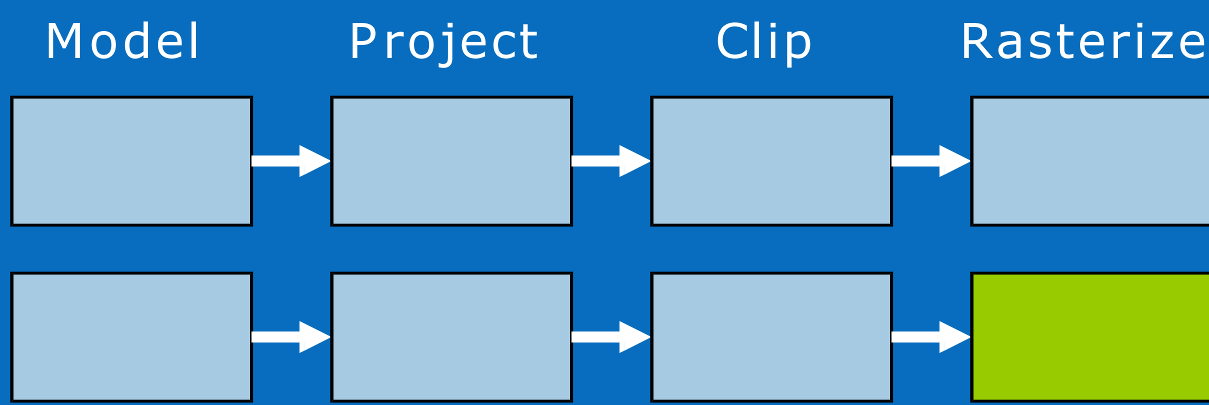
Processing Two Data Sets (Step 3)



Processing Two Data Sets (Step 4)

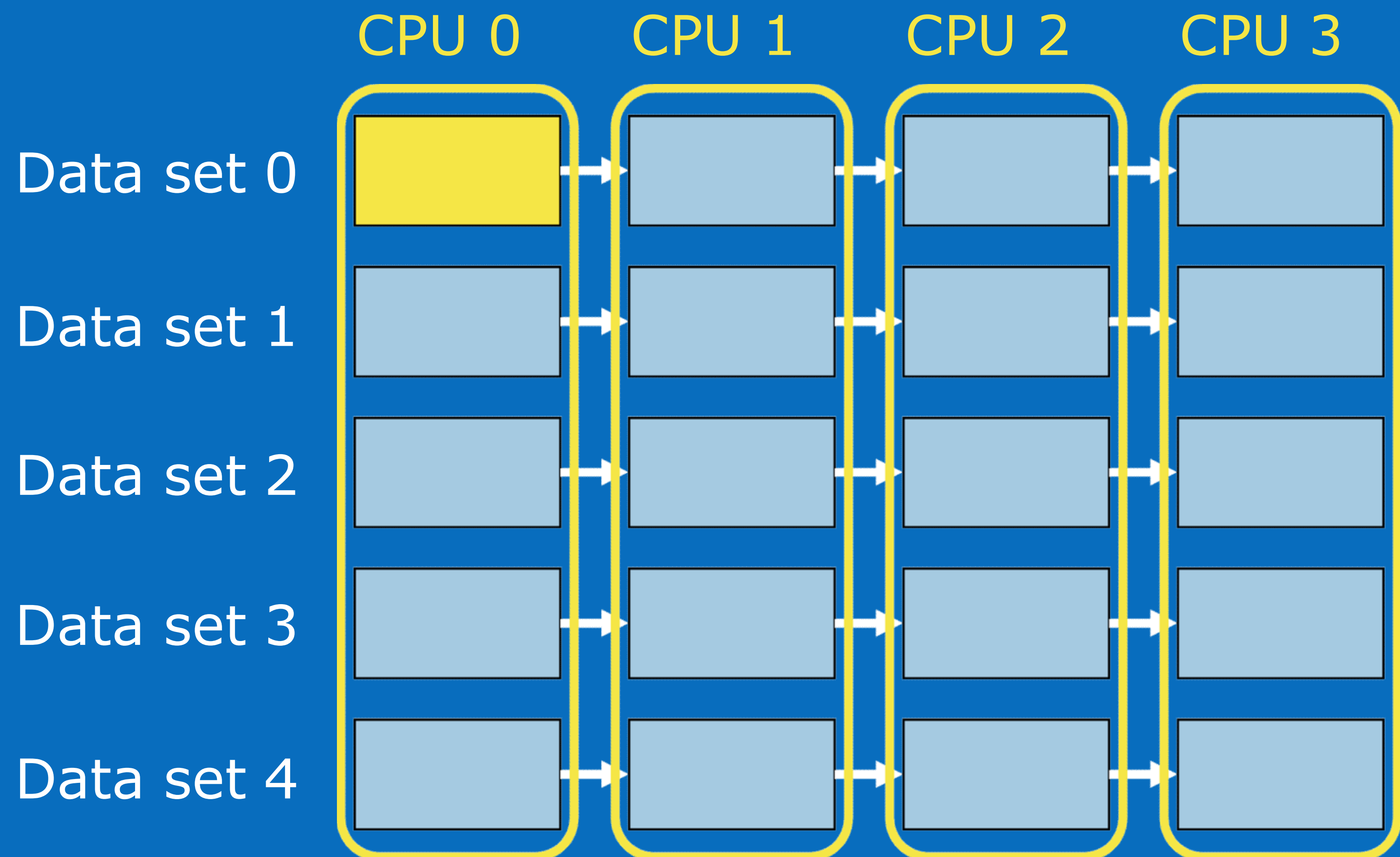


Processing Two Data Sets (Step 5)

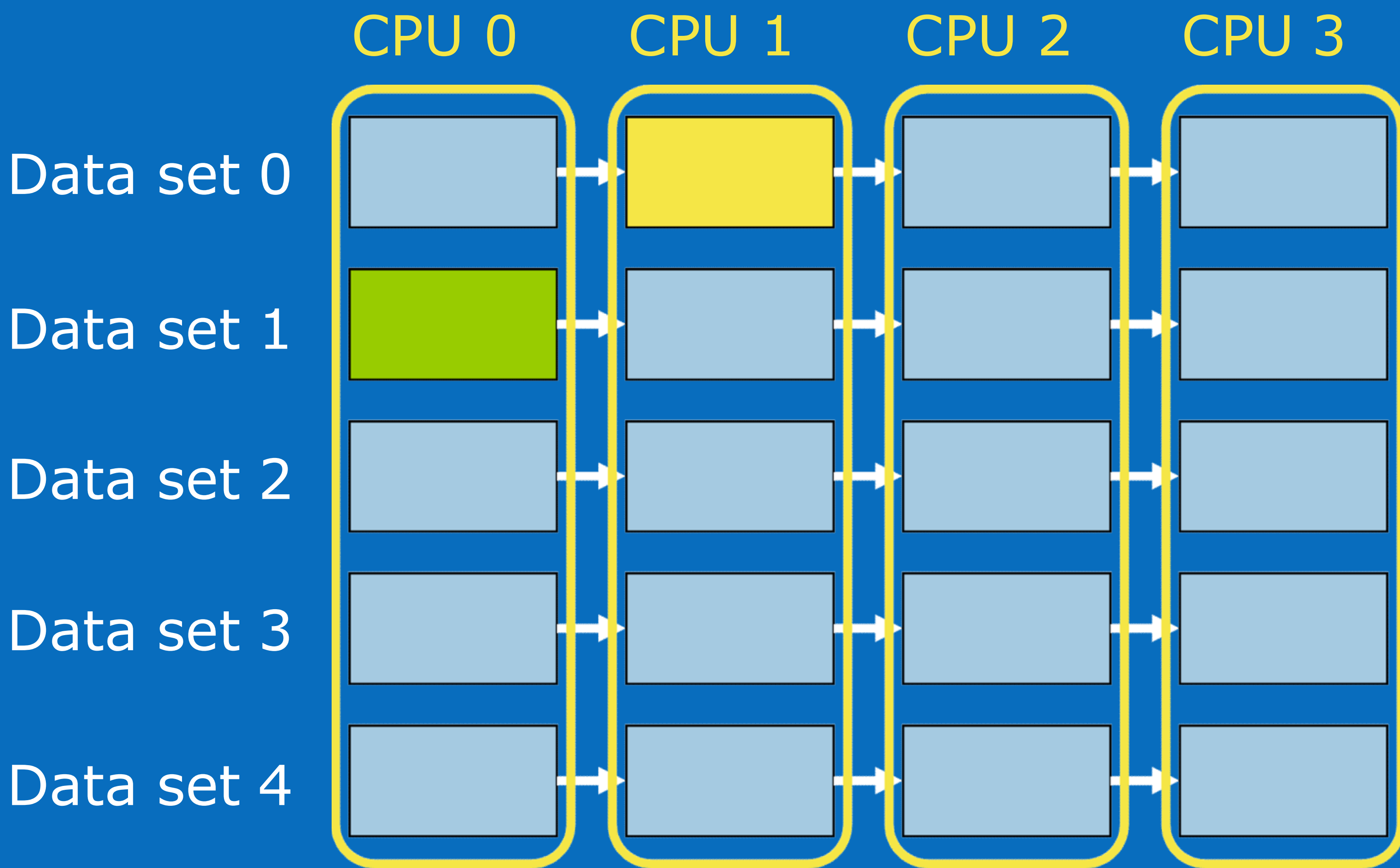


The pipeline processes 2 data sets in 5 steps

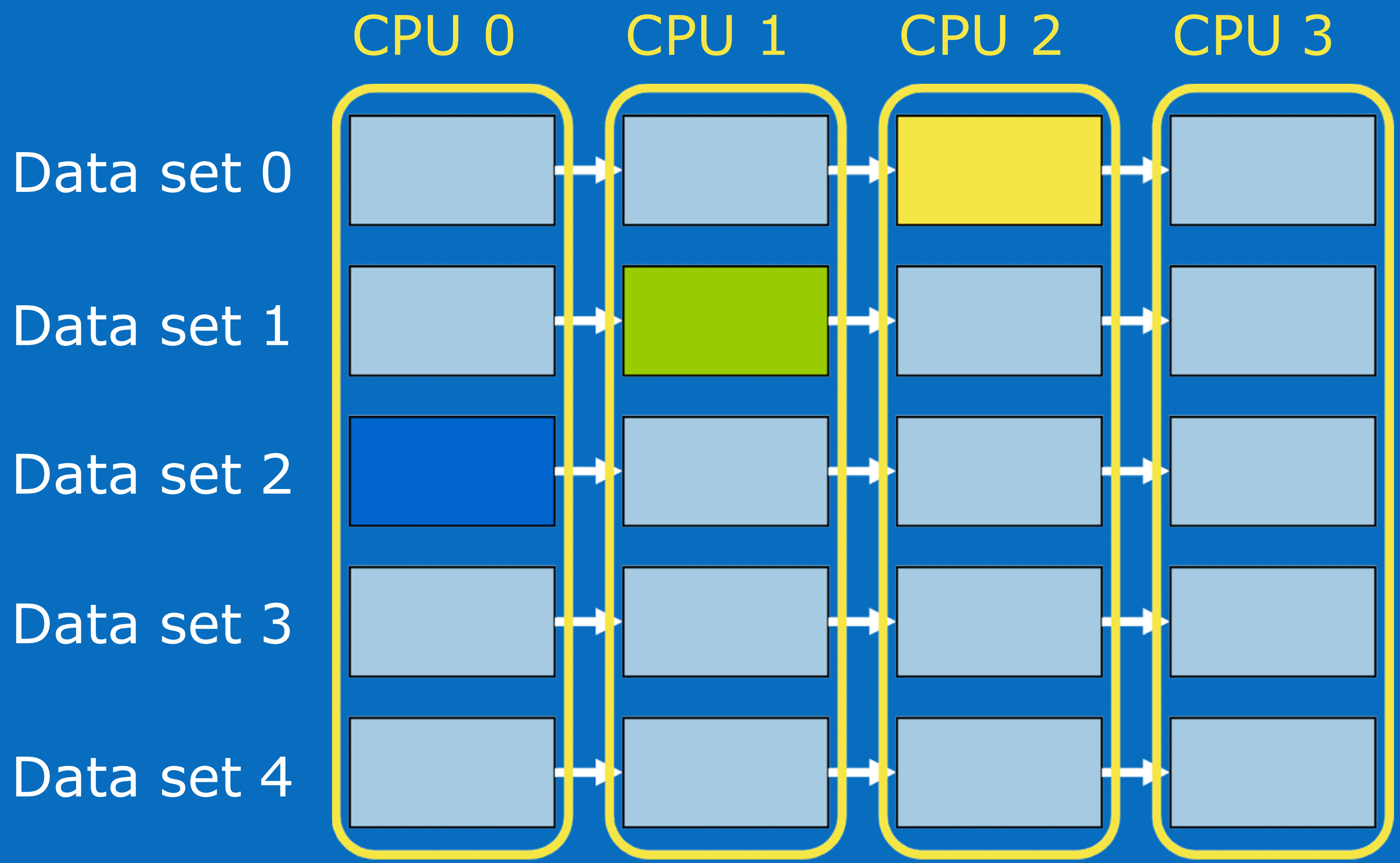
Pipelining Five Data Sets (Step 1)



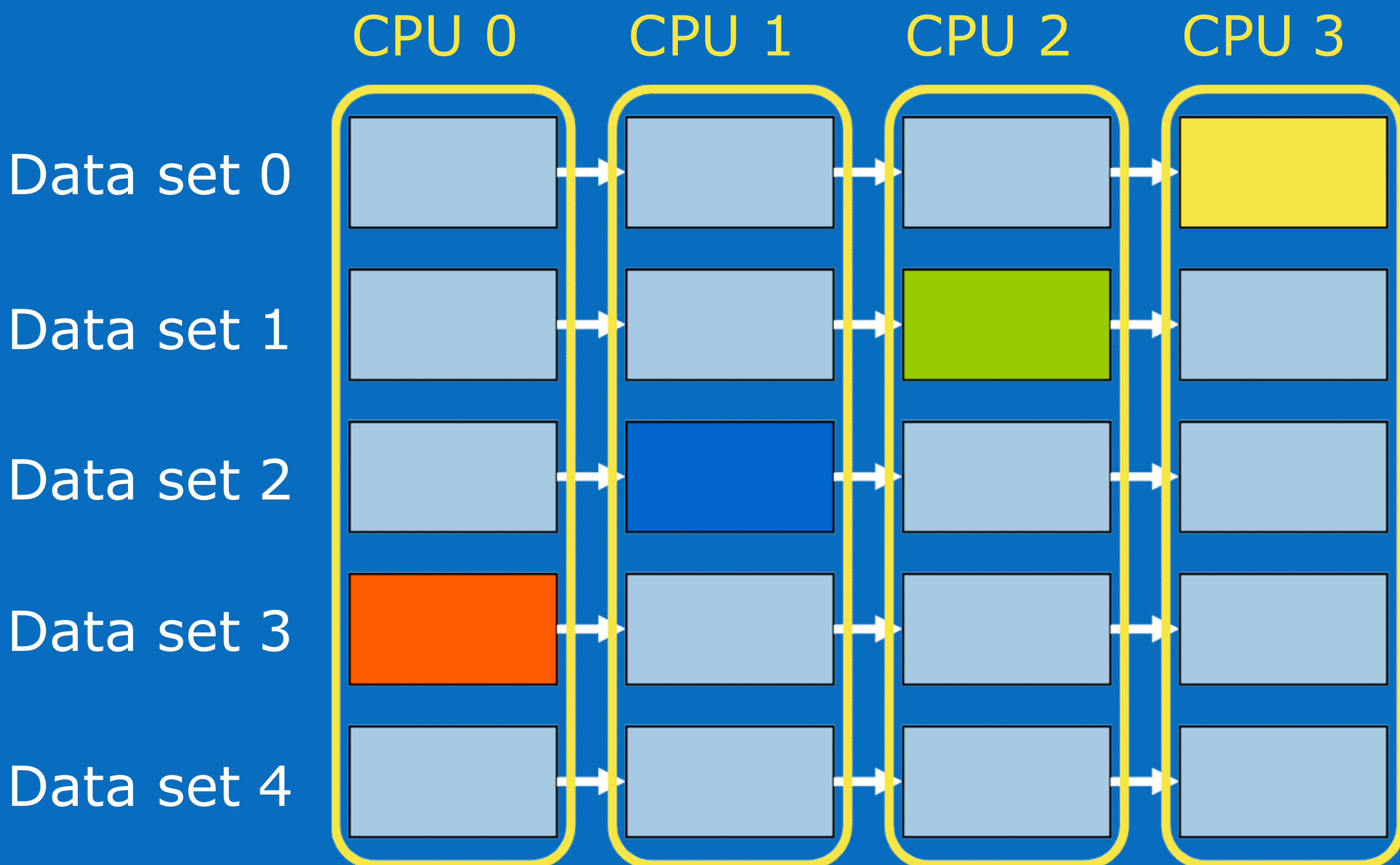
Pipelining Five Data Sets (Step 2)



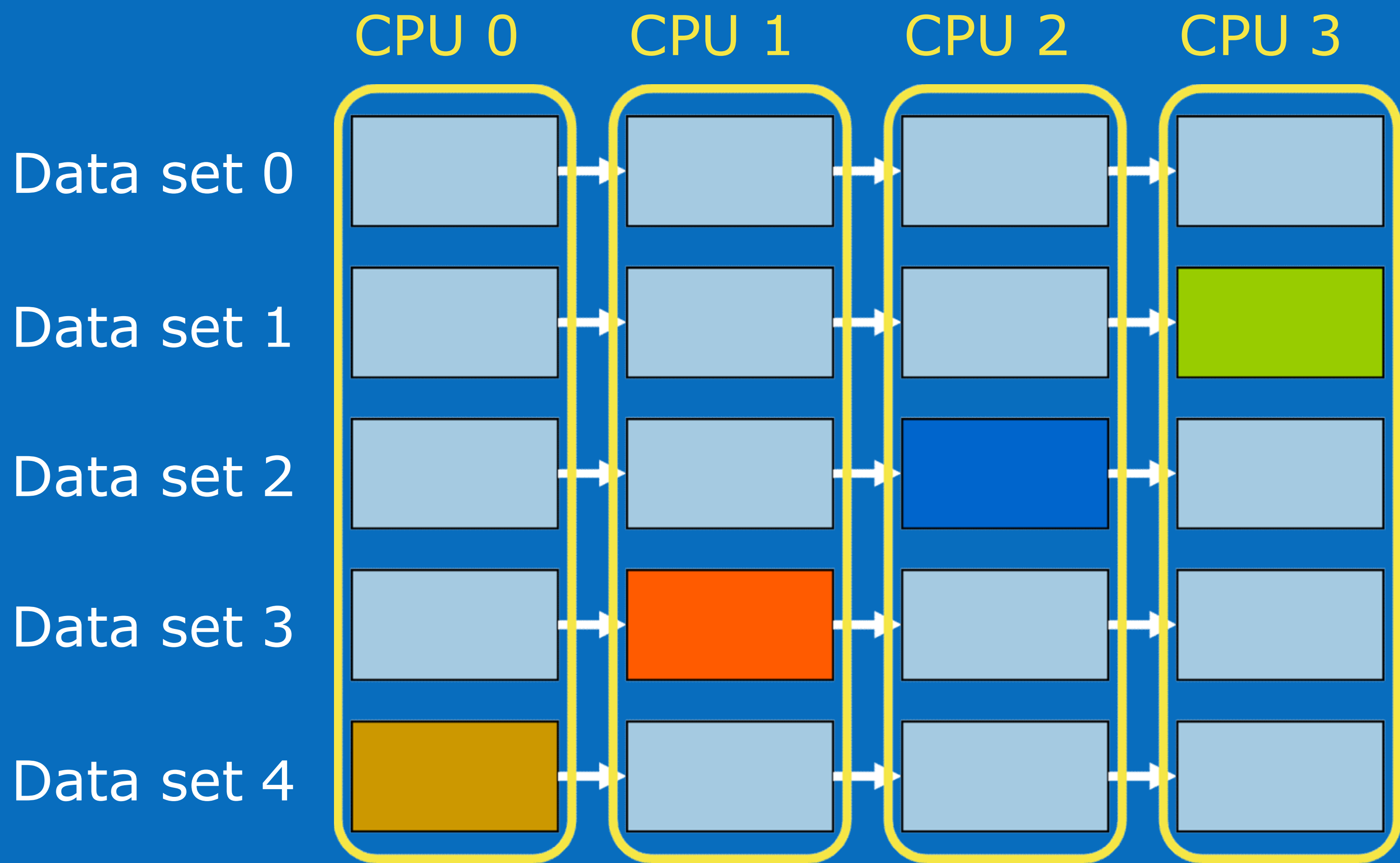
Pipelining Five Data Sets (Step 3)



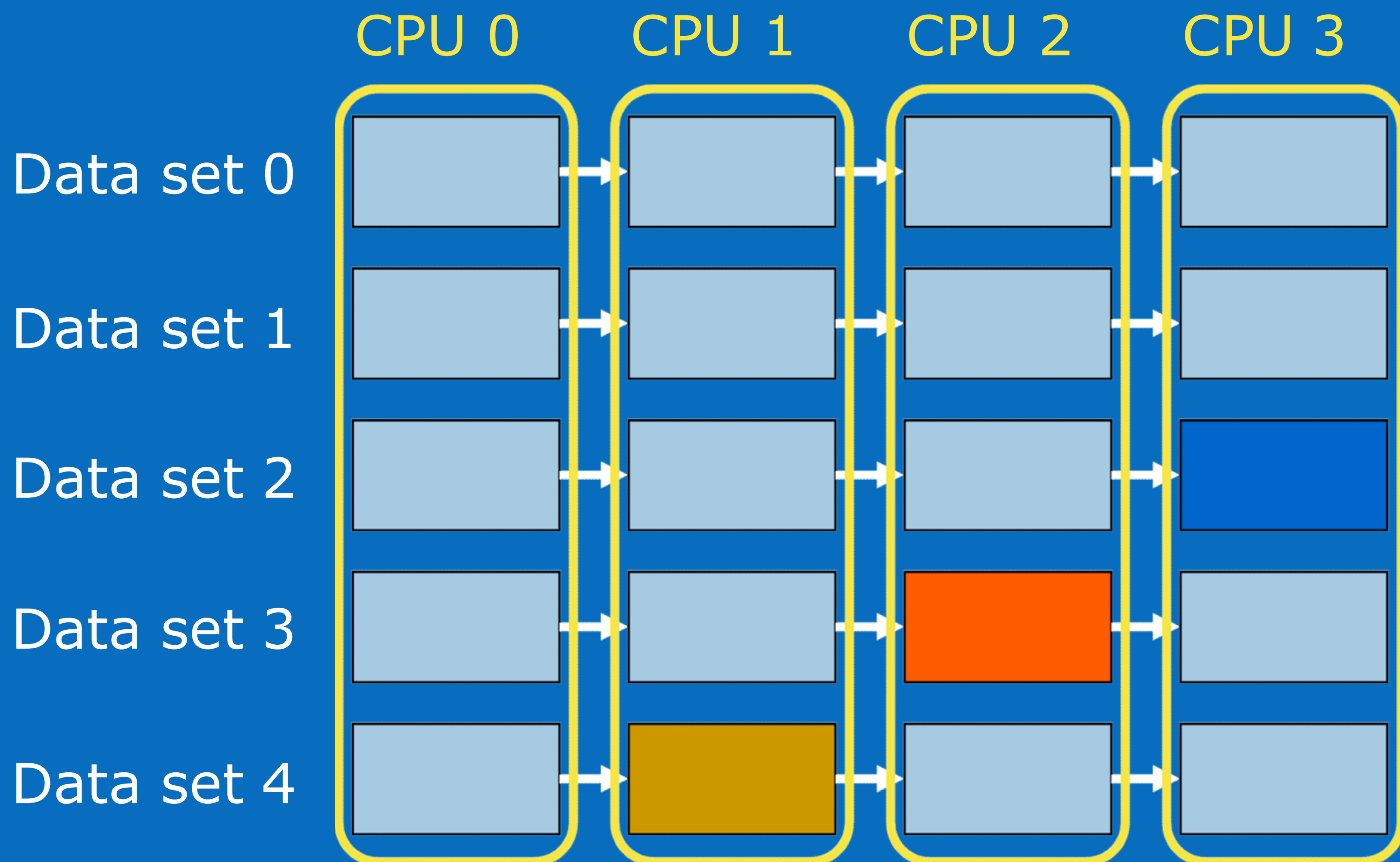
Pipelining Five Data Sets (Step 4)



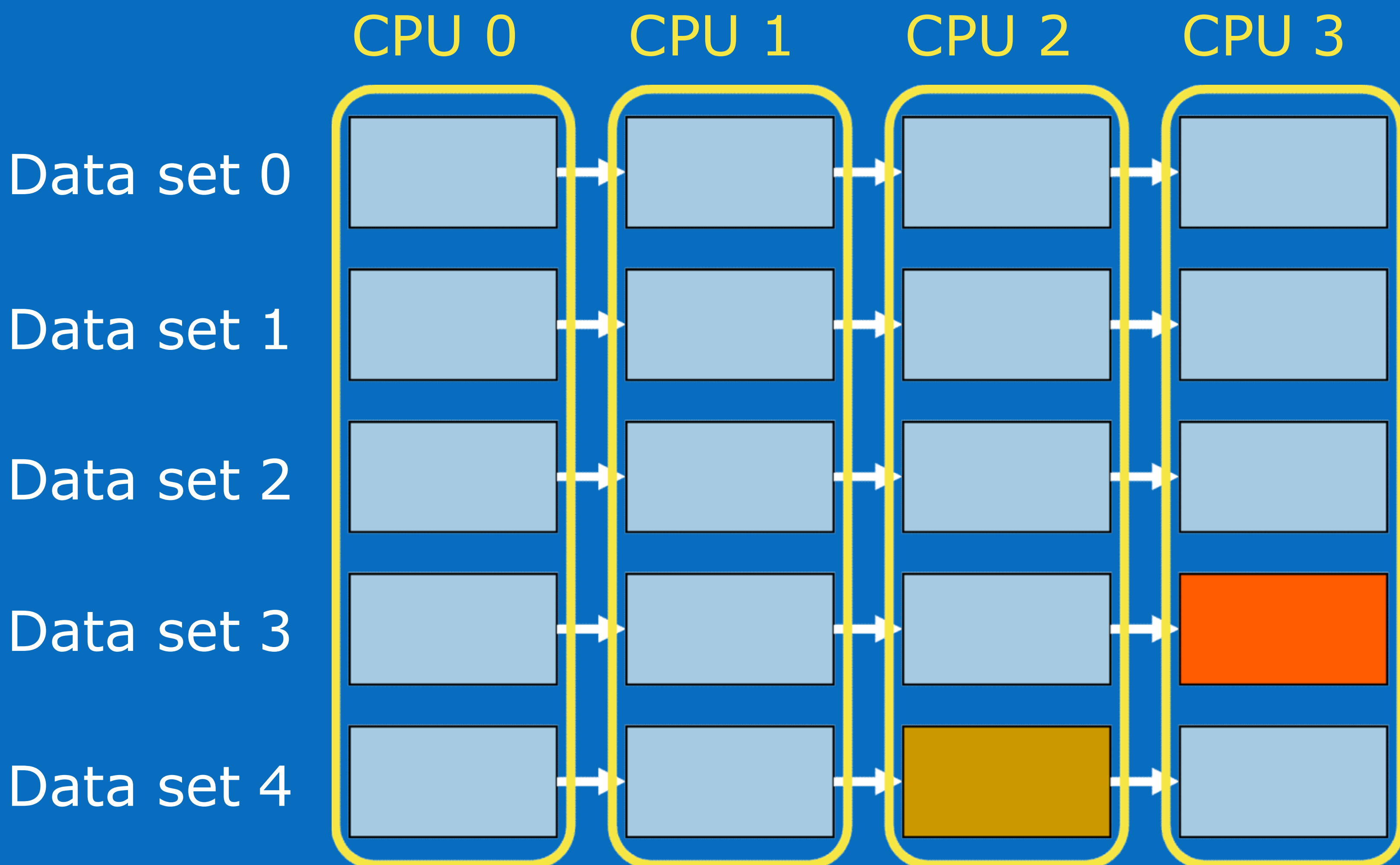
Pipelining Five Data Sets (Step 5)



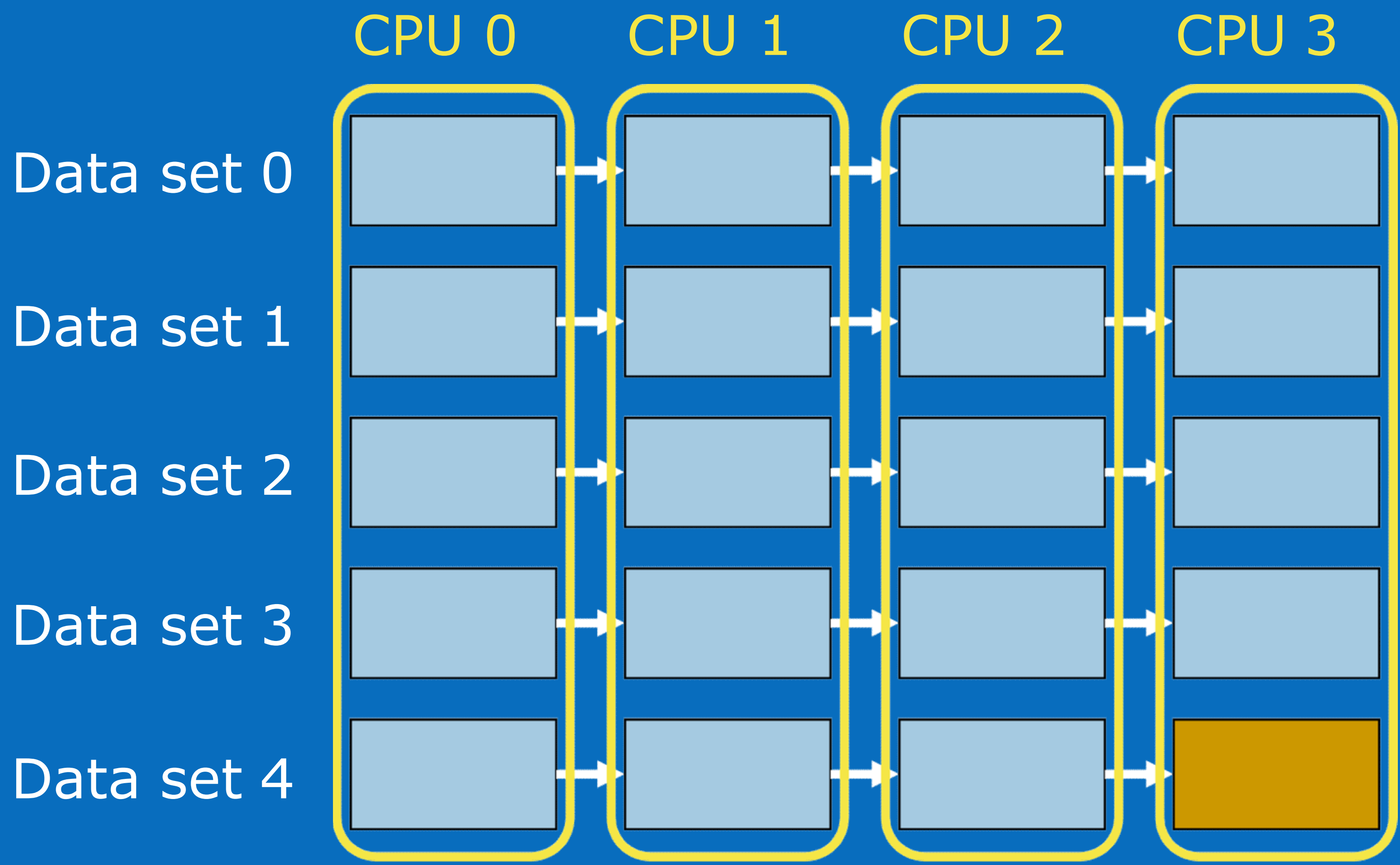
Pipelining Five Data Sets (Step 6)



Pipelining Five Data Sets (Step 7)



Pipelining Five Data Sets (Step 8)



Dependence Graph

Graph = (nodes, arrows)

Node for each

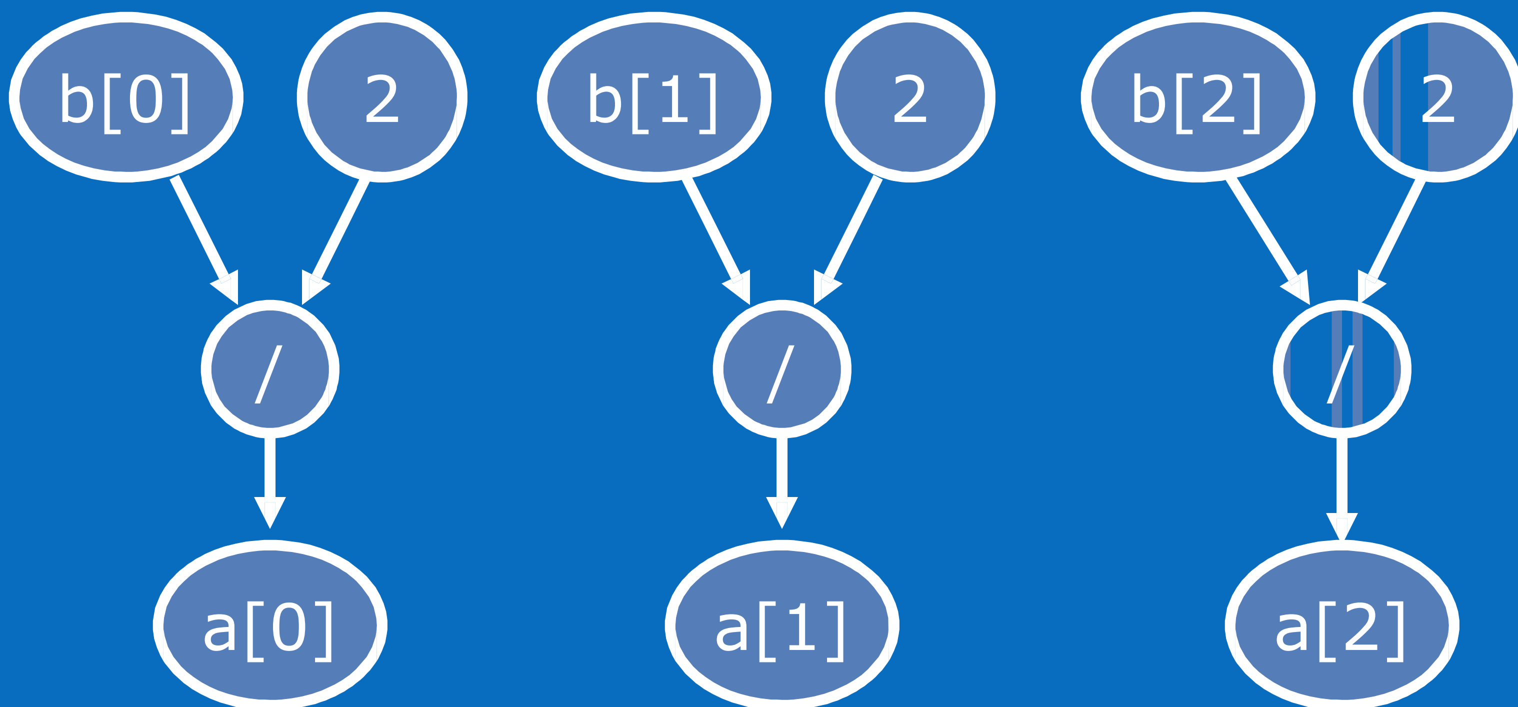
- Variable assignment (except index variables)
- Constant
- Operator or function call

Arrows indicate use of variables and constants

- Data flow
- Control flow

Dependence Graph Example #1

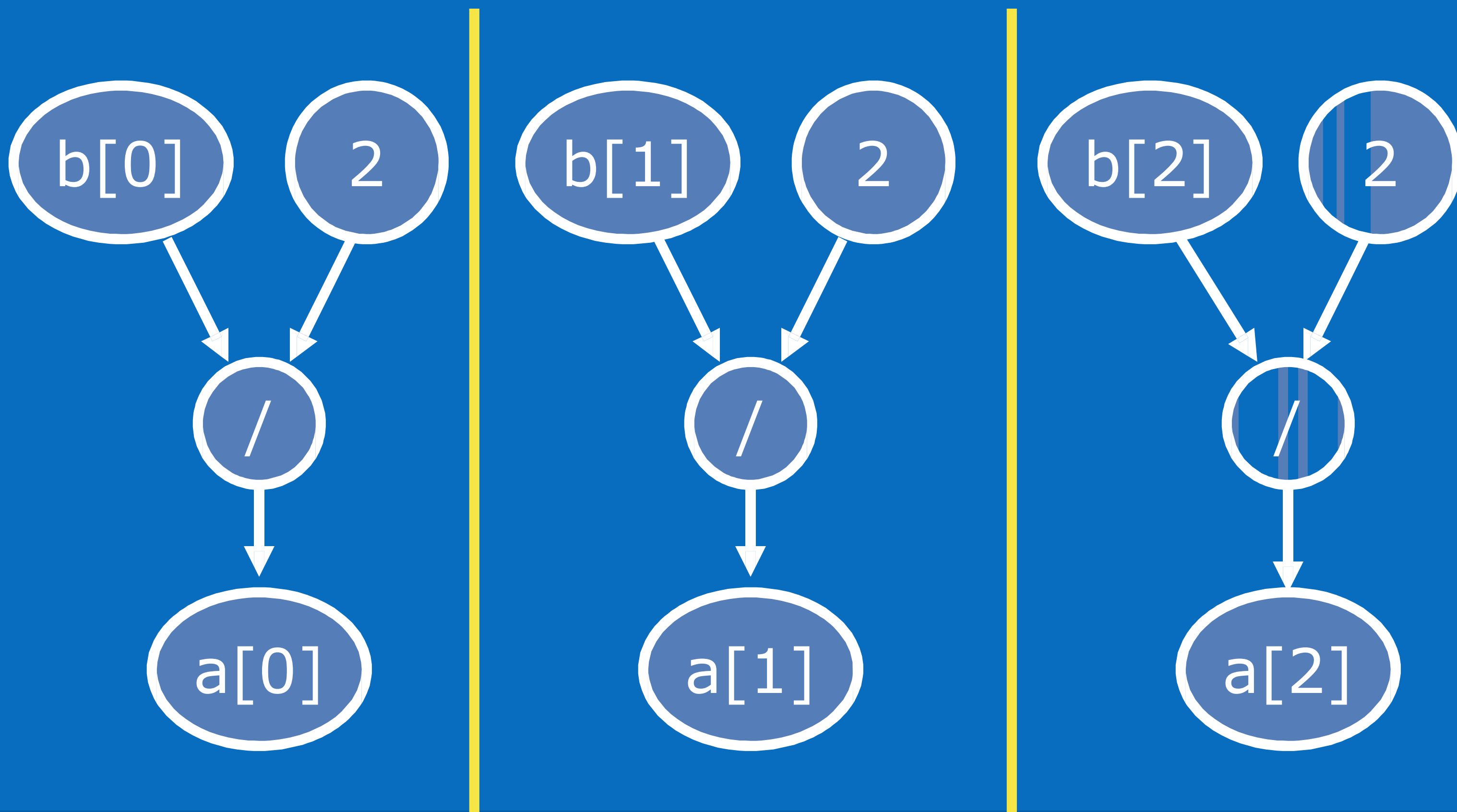
```
for (i = 0; i < 3; i++)
   a[i] = b[i] / 2.0;
```



Dependence Graph Example #1

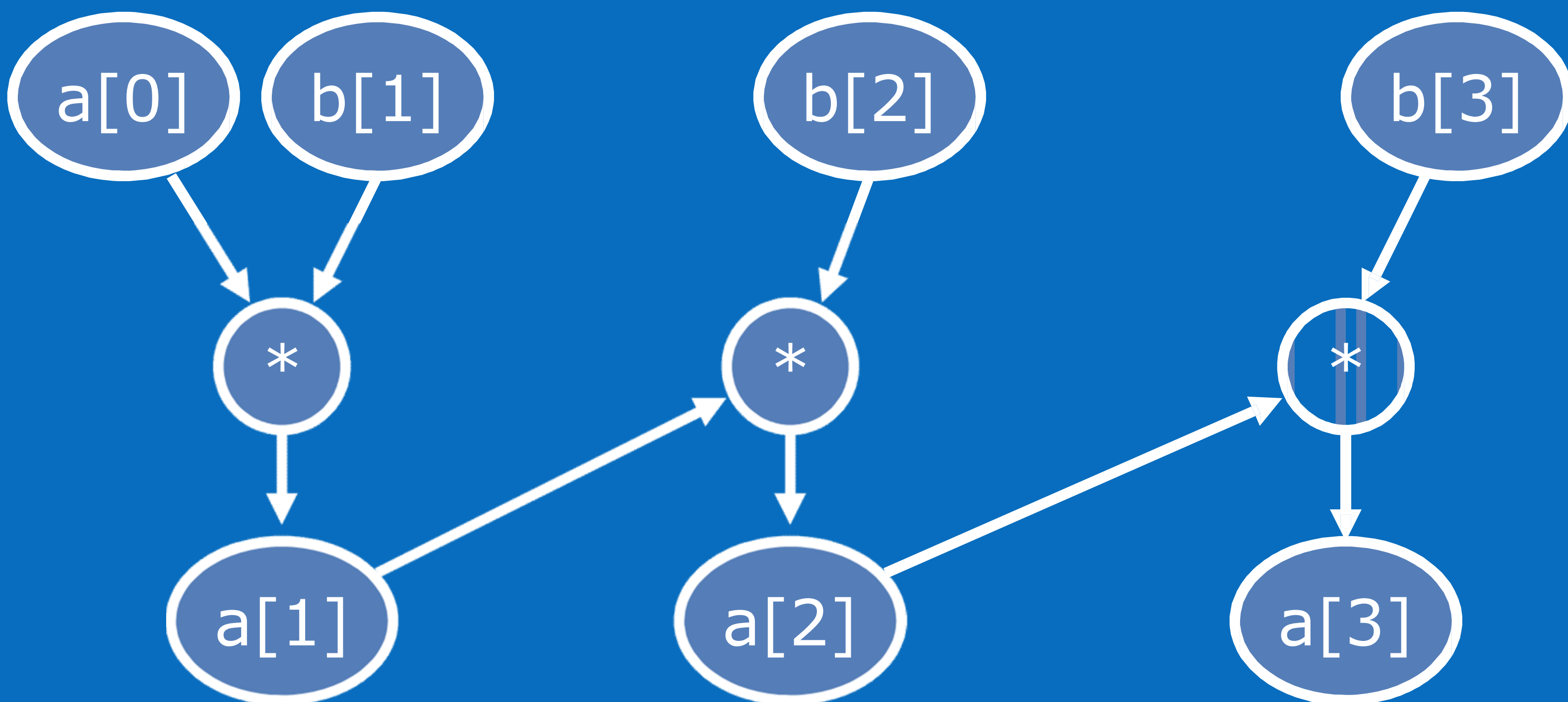
```
for (i = 0; i < 3; i++)
    a[i] = b[i] / 2.0;
```

Domain decomposition possible



Dependence Graph Example #2

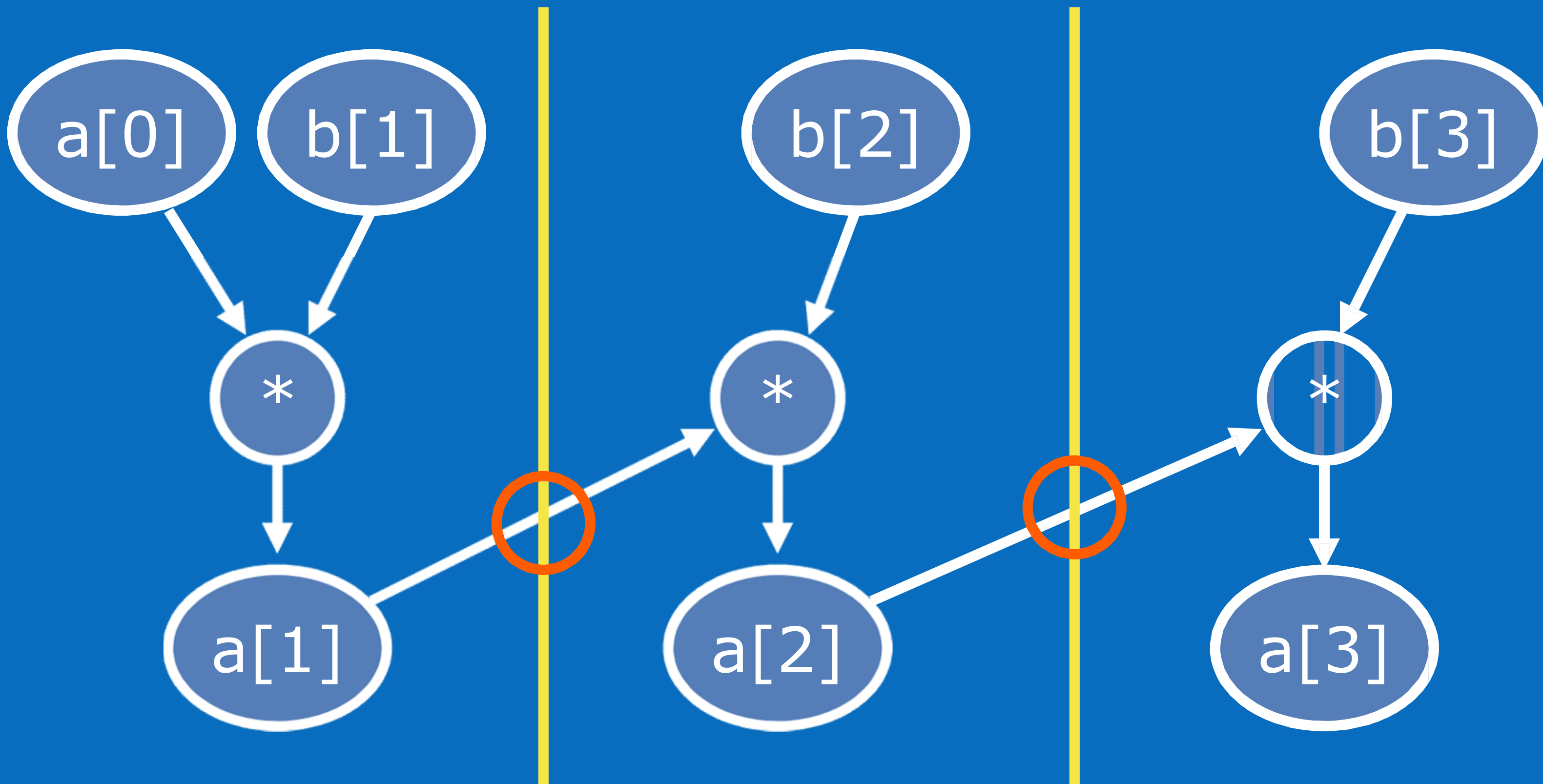
```
for (i = 1; i < 4; i++)
   a[i] = a[i-1] * b[i];
```



Dependence Graph Example #2

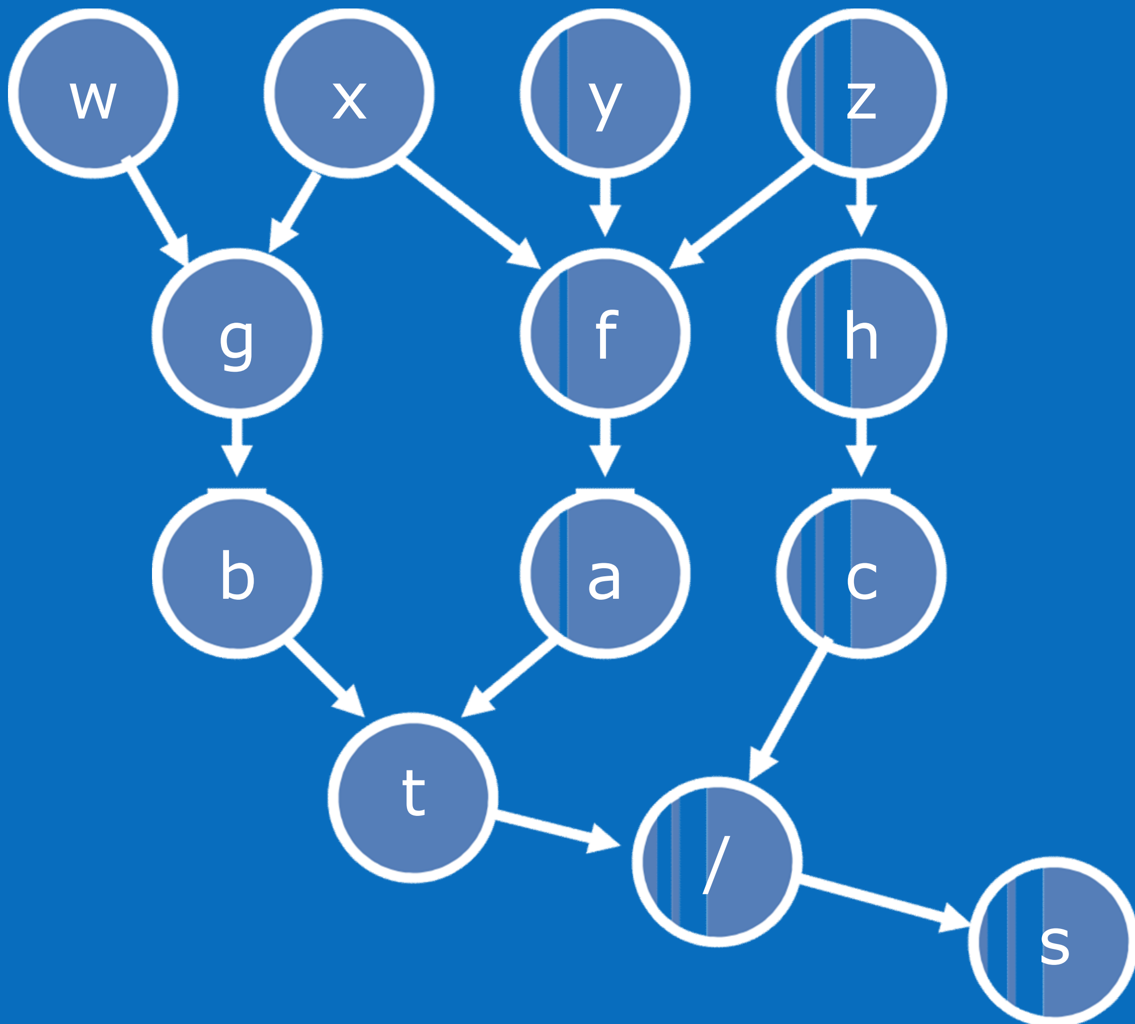
```
for (i = 1; i < 4; i++)
    a[i] = a[i-1] * b[i];
```

No domain decomposition



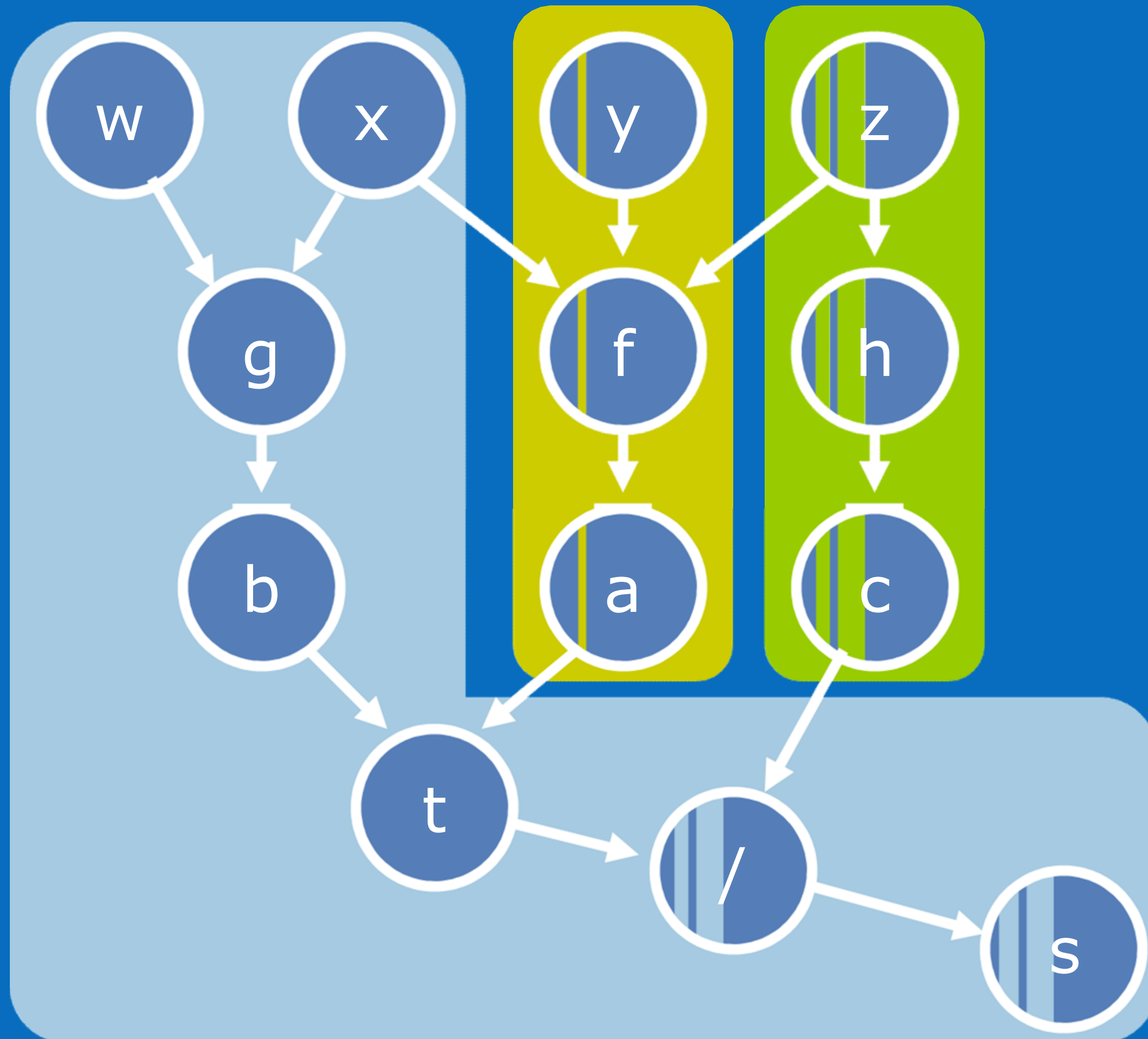
Dependence Graph Example #3

a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;



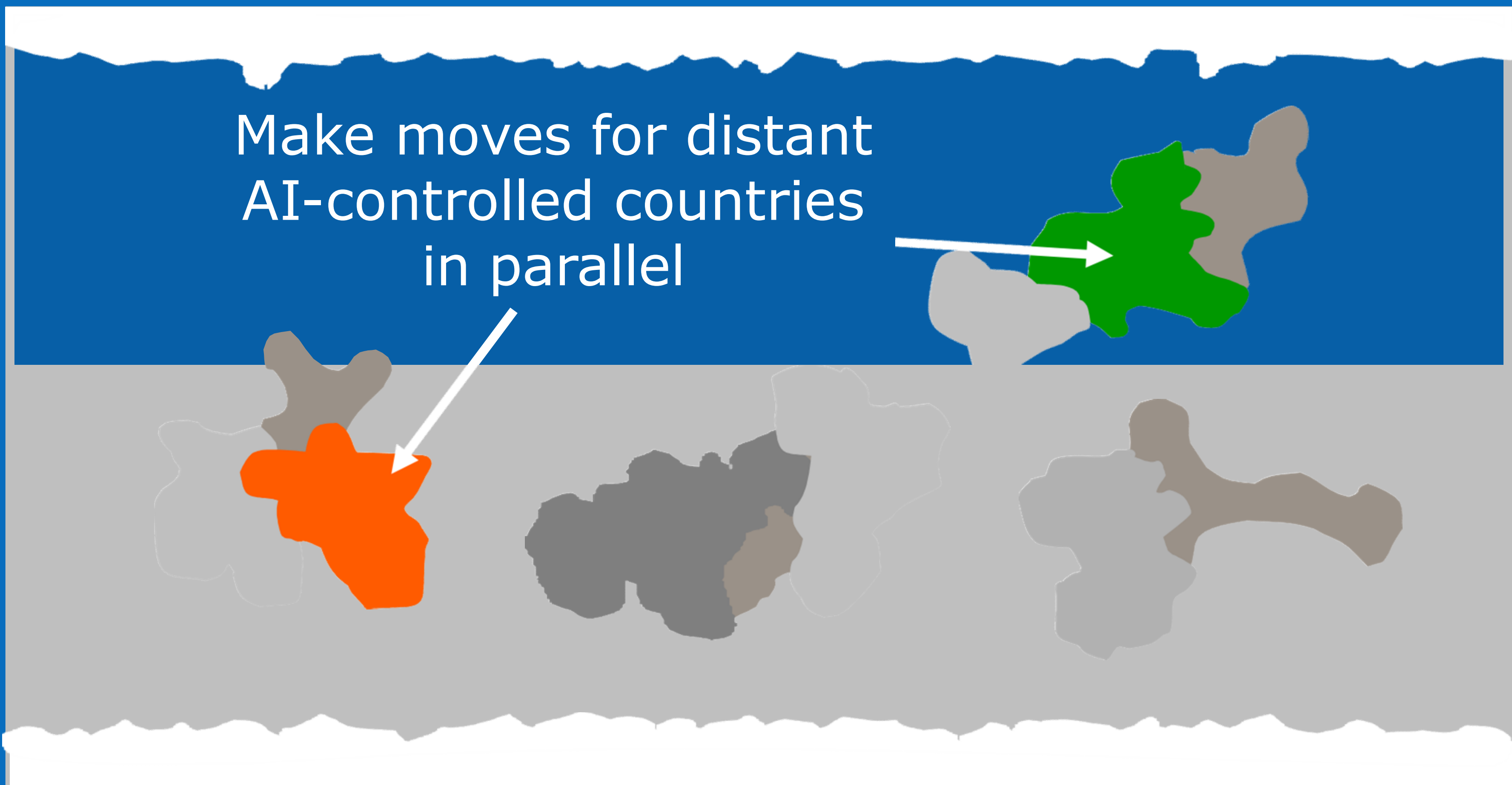
Dependence Graph Example #3

a = f(x, y, z);
b = g(w, x);
t = a + b;
c = h(z);
s = t / c;

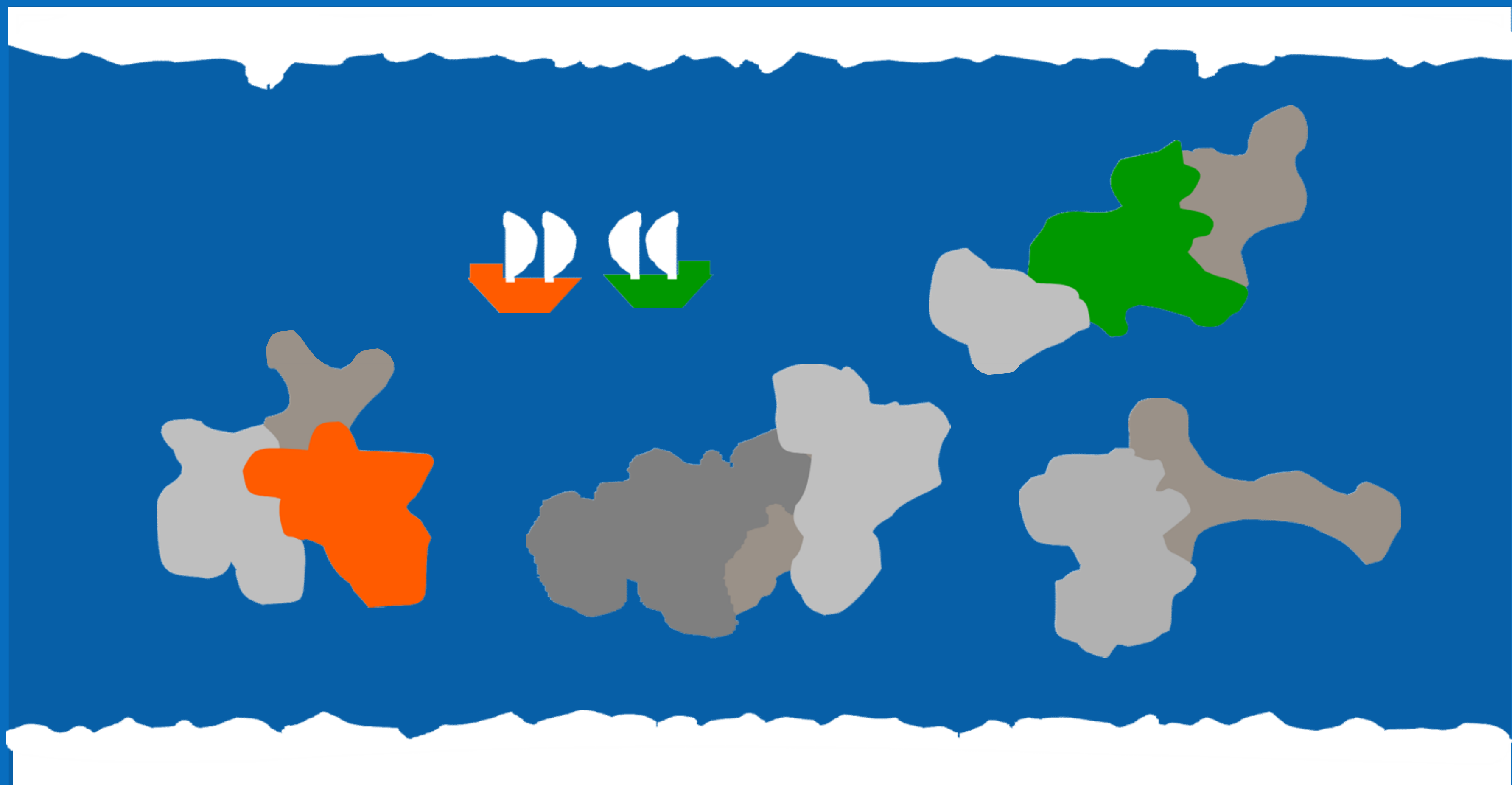


Task decomposition with 3 CPUs.

Speculative Computation in a Turn-Based Strategy Game



Risk: Unexpected Interaction



Orange Cannot Move a Ship that Has Already Been Sunk by Green

Solution: Reverse Time

Must be able to "undo" an erroneous, speculative computation

Analogous to what is done in hardware after incorrect branch prediction

Speculative computations typically do not have a big payoff in parallel computing

Fork/Join Programming Model

When program begins execution, only master thread active

Master thread executes sequential portions of program

For parallel portions of program, master thread *forks* (creates or awakens) additional threads

At *join* (end of parallel section of code), extra threads are suspended or die

Relating Fork/Join to Code

```
for {
}
for {
}
```

Sequential code

Parallel code

Sequential code

Parallel code

Sequential code

Incremental Parallelization

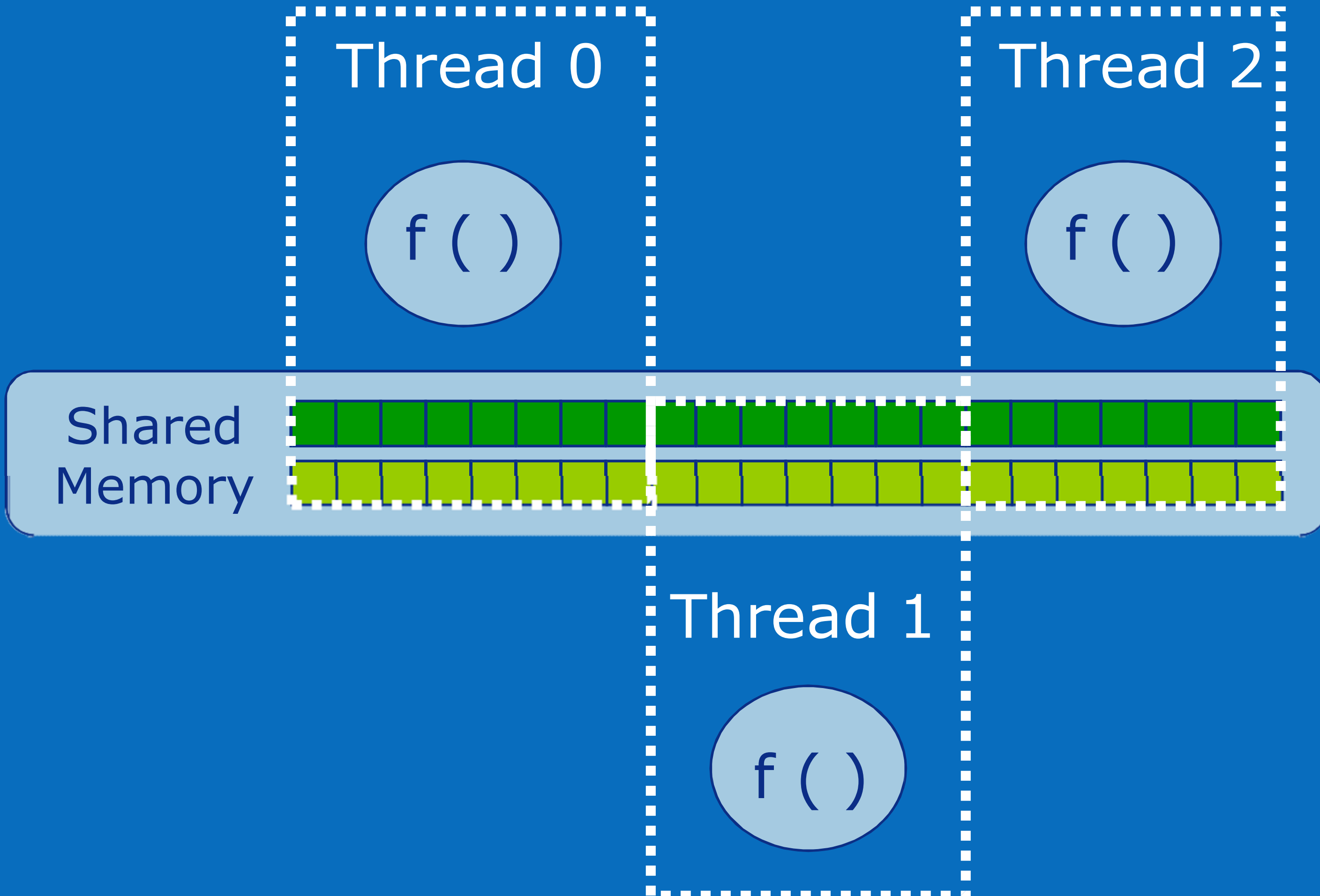
- Sequential program a special case of threaded program
- Programmers can add parallelism incrementally
- Profile program execution
- Repeat
  - Choose best opportunity for parallelization
  - Transform sequential code into parallel code
- Until further improvements not worth the effort

Utility of Threads

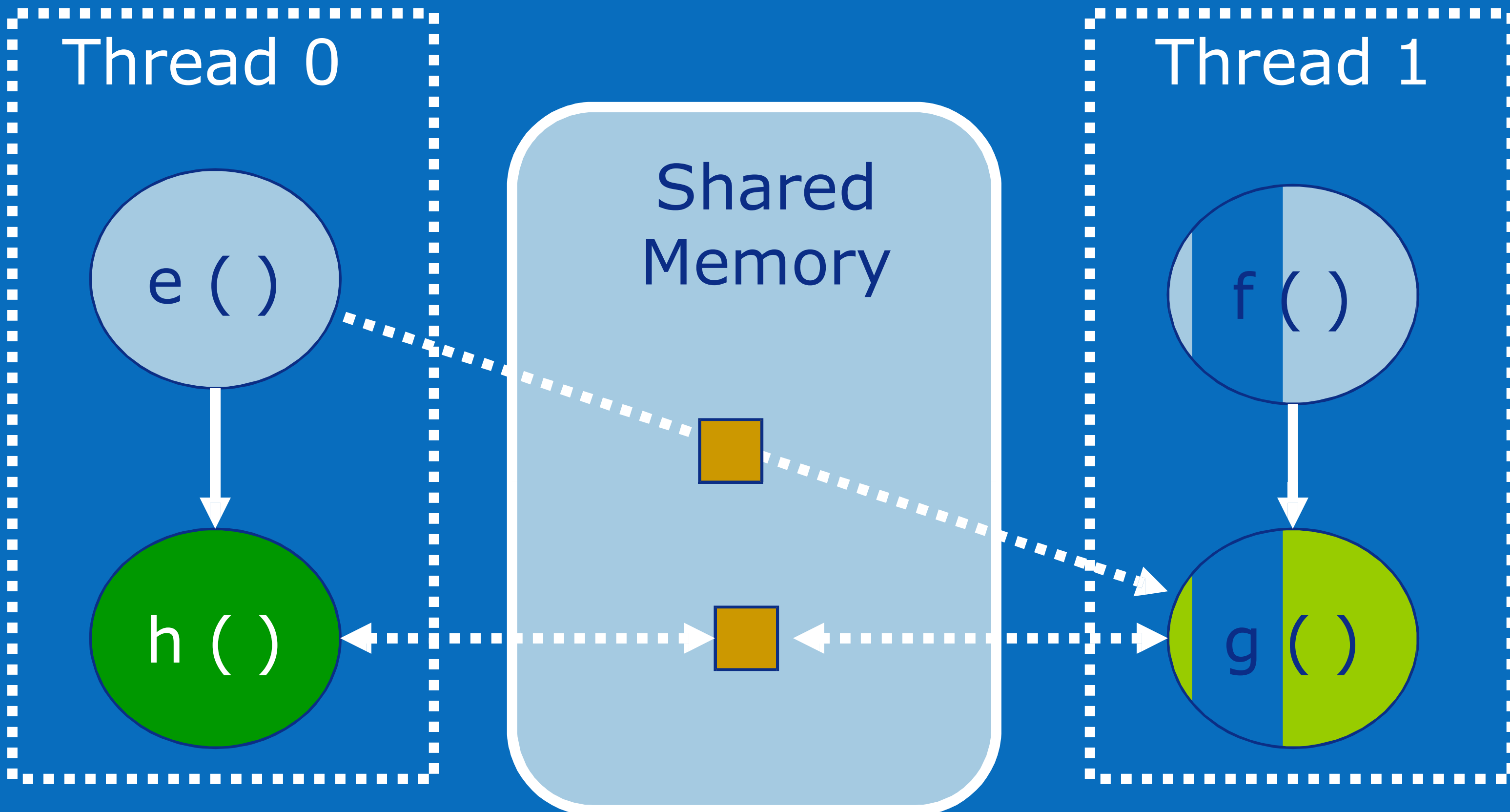
Threads are flexible enough to implement

- Domain decomposition
- Functional decomposition
- Pipelining

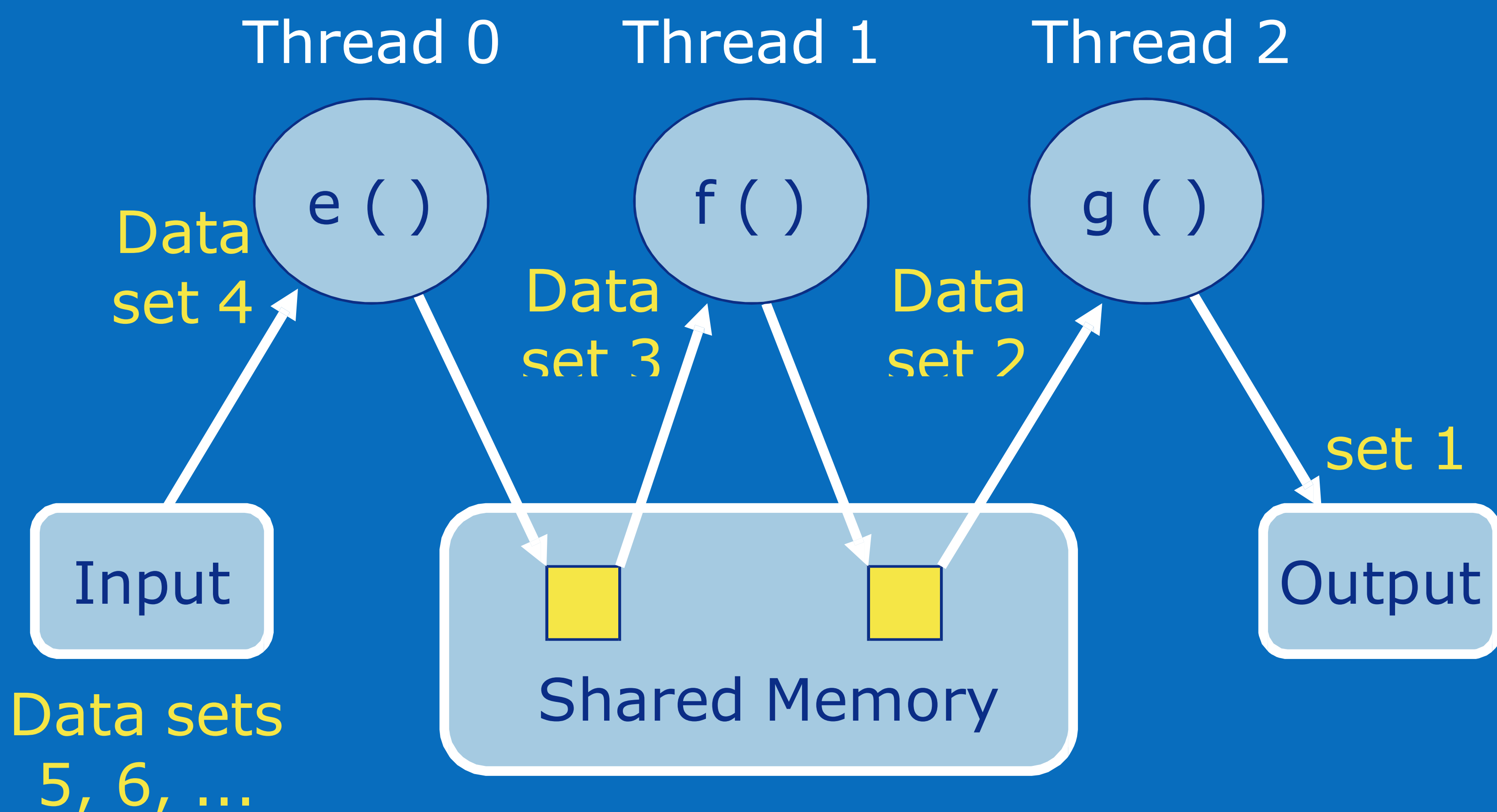
Domain Decomposition Using Threads



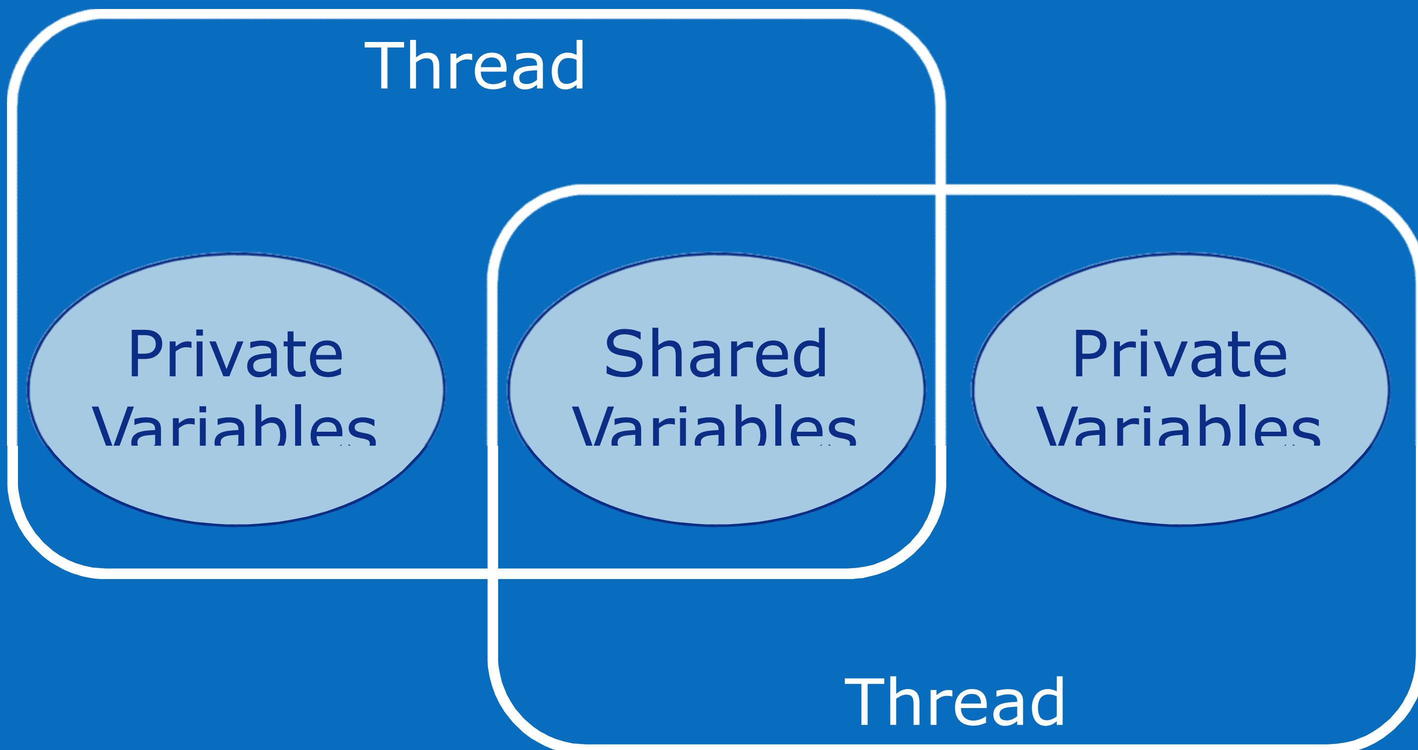
Functional Decomposition Using Threads



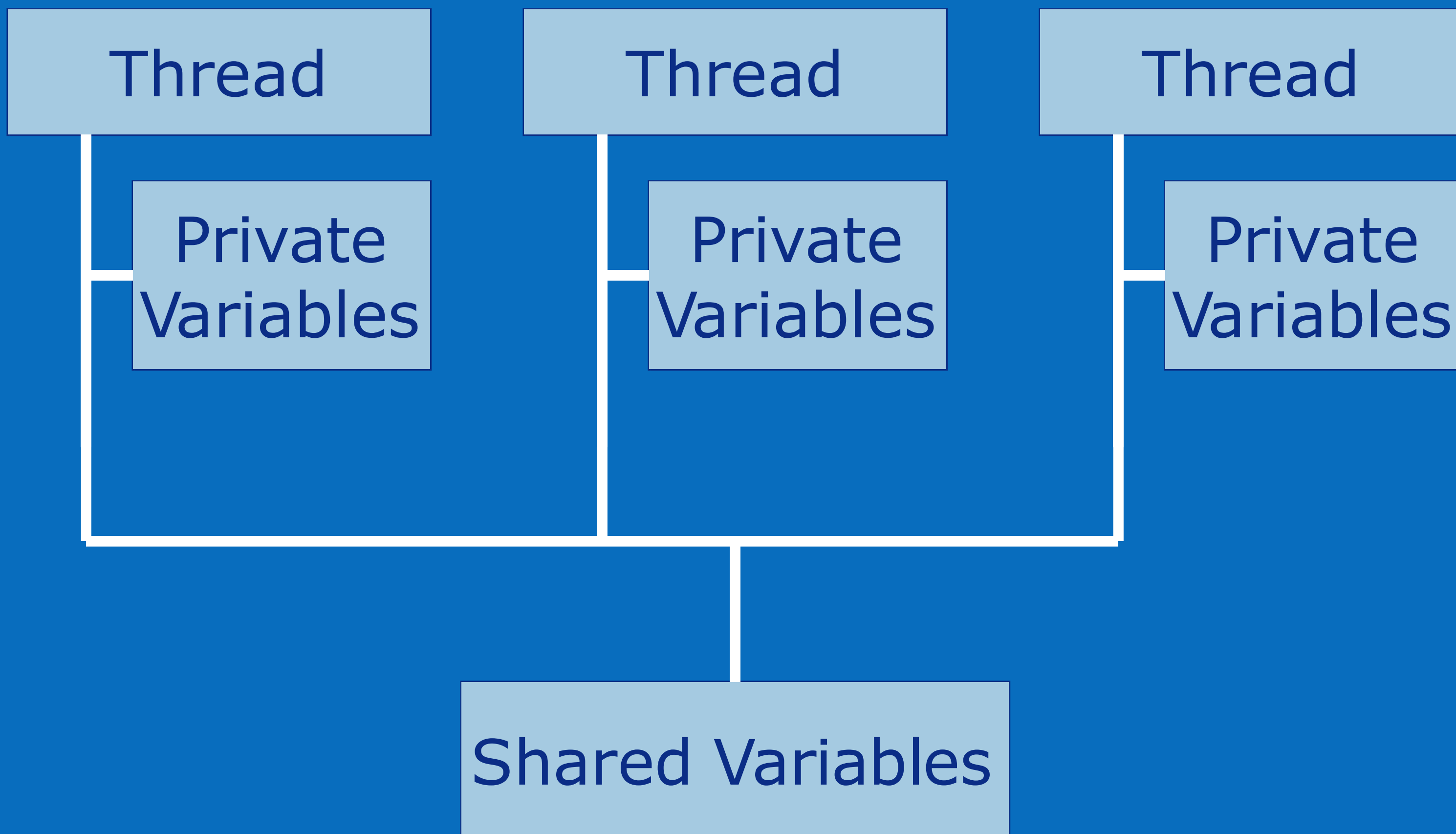
Pipelining Using Threads



Shared versus Private Variables



The Threads Model



What Is OpenMP?

- OpenMP is an API for parallel programming
- First developed by the OpenMP Architecture Review Board (1997), now a standard
- Designed for shared-memory multiprocessors
- Set of compiler directives, library functions, and environment variables, but not a language
- Can be used with C, C++, or Fortran
- Based on fork/join model of threads

Strengths and Weaknesses of OpenMP

Strengths

- Well-suited for domain decompositions
- Available on Unix and Windows NT

Weaknesses

- Not well-tailored for functional decompositions
- Compilers do not have to check for such errors as deadlocks and race conditions

Syntax of Compiler Directives

A C/C++ compiler directive is called a *pragma*

Pragmas are handled by the preprocessor

All OpenMP pragmas have the syntax:

`#pragma omp` <*rest of pragma*>

Pragmas appear immediately before relevant construct

Pragma: parallel for

The compiler directive

```
#pragma omp parallel for
```

tells the compiler that the `for` loop which immediately follows can be executed in parallel

The number of loop iterations must be computable at run time before loop executes

Loop must not contain a `break`, `return`, or `exit`

Loop must not contain a `goto` to a label outside loop

Example

```
int first, *marked, prime, size;
...
#pragma omp parallel for
for (i = first; i < size; i += prime)
      marked[i] = 1;
```

Matching Threads with CPUs

Function `omp_get_num_procs` returns the number of physical processors available to the parallel program

```
int omp_get_num_procs (void);
```

Example:

```
int t;
...
t = omp_get_num_procs();
```

Matching Threads with CPUs (cont.)

Function `omp_set_num_threads` allows you to set the number of threads that should be active in parallel sections of code

```
void omp_set_num_threads (int t);
```

The function can be called with different arguments at different points in the program

Example:

```
int t;

…

omp_set_num_threads (t);
```

Which Loop to Make Parallel?

```
main () {
int i, j, k;
float **a, **b;
...
for (k = 0; k < N; k++)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i][j] = MIN(a[i][j], a[i][k] + a[k][j]);
```

Loop-carried dependences

Can execute in parallel

Can execute in parallel

Grain Size

There is a fork/join for every instance of

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

Since fork/join is a source of overhead, we want to maximize the amount of work done for each fork/join; i.e., the *grain size*

Hence we choose to make the middle loop parallel

Almost Right, but Not Quite

```
main () {
int i, j, k;
float **a, **b;
...
for (k = 0; k < N; k++)
  #pragma omp parallel for
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i][j] = MIN(a[i][j], a[i][k] + a[k][j]);
```

Problem: j is a shared variable

Problem Solved with private Clause

```
main () {
int i, j, k;
float **a, **b;
...
for (k = 0; k < N; k++)
  #pragma omp parallel for private (j)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i][j] = MIN(a[i][j], a[i][k] + a[k][j]);
```

Tells compiler to make listed variables private

Example

```
int i;
float *a, *b, *c, tmp;
...
for (i = 0; i < N; i++) {
    tmp = a[i] / b[i];
    c[i] = tmp * tmp;
}
```

Loop is perfectly parallelizable except for shared variable “tmp”

Solution

```
int i;
float *a, *b, *c, tmp;
...
#pragma omp parallel for private (tmp)
for (i = 0; i < N; i++) {
   tmp = a[i] / b[i];
   c[i] = tmp * tmp;
}
```

More about Private Variables

Each thread has its own copy of the private variables

If `j` is declared private, then inside the `for` loop no thread can access the "other" `j` (the `j` in shared memory)

`j`

No thread can assign a new value to the shared `j`

Private variables are undefined at loop entry and loop exit, reducing execution time

Clause: firstprivate

The `firstprivate` clause tells the compiler that the private variable should inherit the value of the shared variable upon loop entry

The value is assigned once per thread, not once per loop iteration

Example

```
a[0] = 0.0;
for (i = 1; i < N; i++)
     a[i] = alpha (i, a[i-1]);
#pragma omp parallel for firstprivate (a)
for (i = 0; i < N; i++) {
   b[i] = beta (i, a[i]);
   a[i] = gamma (i);
   c[i] = delta (a[i], b[i]);
}
```

Clause: lastprivate

The `lastprivate` clause tells the compiler that the value of the private variable after the *sequentially last* loop iteration should be assigned to the shared variable upon loop exit

In other words, when the thread responsible for the sequentially last loop iteration exits the loop, its copy of the private variable is copied back to the shared variable

Example

```
#pragma omp parallel for lastprivate (x)
for (i = 0; i < N; i++) {
    x = foo (i);
    y[i] = bar(i, x);
}
last_x = x;
```

Pragma: parallel

In the effort to increase grain size, sometimes the code that should be executed in parallel goes beyond a single `for` loop

The `parallel` pragma is used when a block of code should be executed in parallel

Pragma: for

The `for` pragma is used inside a block of code already marked with the `parallel` pragma

It indicates a `for` loop whose iterations should be divided among the active threads

There is a *barrier synchronization* of the threads at the end of the `for` loop

Pragma: single

The `single` pragma is used inside a parallel block of code

It tells the compiler that only a single thread should execute the statement or block of code immediately following

Clause: nowait

The `nowait` clause tells the compiler that there is no need for a barrier synchronization at the end of a parallel `for` loop or `single` block of code

Case: parallel, for, single Pragmas

```
for (i = 0; i < N; i++)
    a[i] = alpha(i);
if (delta < 0.0) printf ("delta < 0.0\n");
for (i = 0; i < N; i++)
    b[i] = beta (i, delta);
```

Solution: parallel, for, single Pragma

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (i = 0; i < N; i++)
        a[i] = alpha(i);
    #pragma omp single nowait
    if (delta < 0.0) printf ("delta < 0.0\n");
    #pragma omp for
    for (i = 0; i < N; i++)
        b[i] = beta (i, delta);
}
```

Extended Example

```
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        printf ("Exiting during iteration %d\n", i);
        break;
    }
    for (j = low; j < high; j++)
        c[j] += alpha (i, j);
}
```

Extended Example

```
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
      printf ("Exiting during iteration %d\n", i);
      break;
   }
   #pragma omp parallel for
   for (j = low; j < high; j++)
      c[j] += alpha (i, j);
}
```

Extended Example

```
#pragma omp parallel private (i, j, low, high)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
      printf ("Exiting during iteration %d\n", i);
      break;
   }
   #pragma omp for nowait
   for (j = low; j < high; j++)
      c[j] += alpha (i, j);
}
```

Extended Example

```
#pragma omp parallel private (i, j, low, high)
for (i = 0; i < m; i++) {
    low = a[i];
    high = b[i];
    if (low > high) {
        #pragma omp single nowait
        printf ("Exiting during iteration %d\n", i);
        break;
    }
    #pragma omp for nowait
    for (j = low; j < high; j++)
        c[j] += alpha (i, j);
}
```

Potential Pitfall?

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
    x = (i + 0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

What happens when we make the `for` loop parallel?

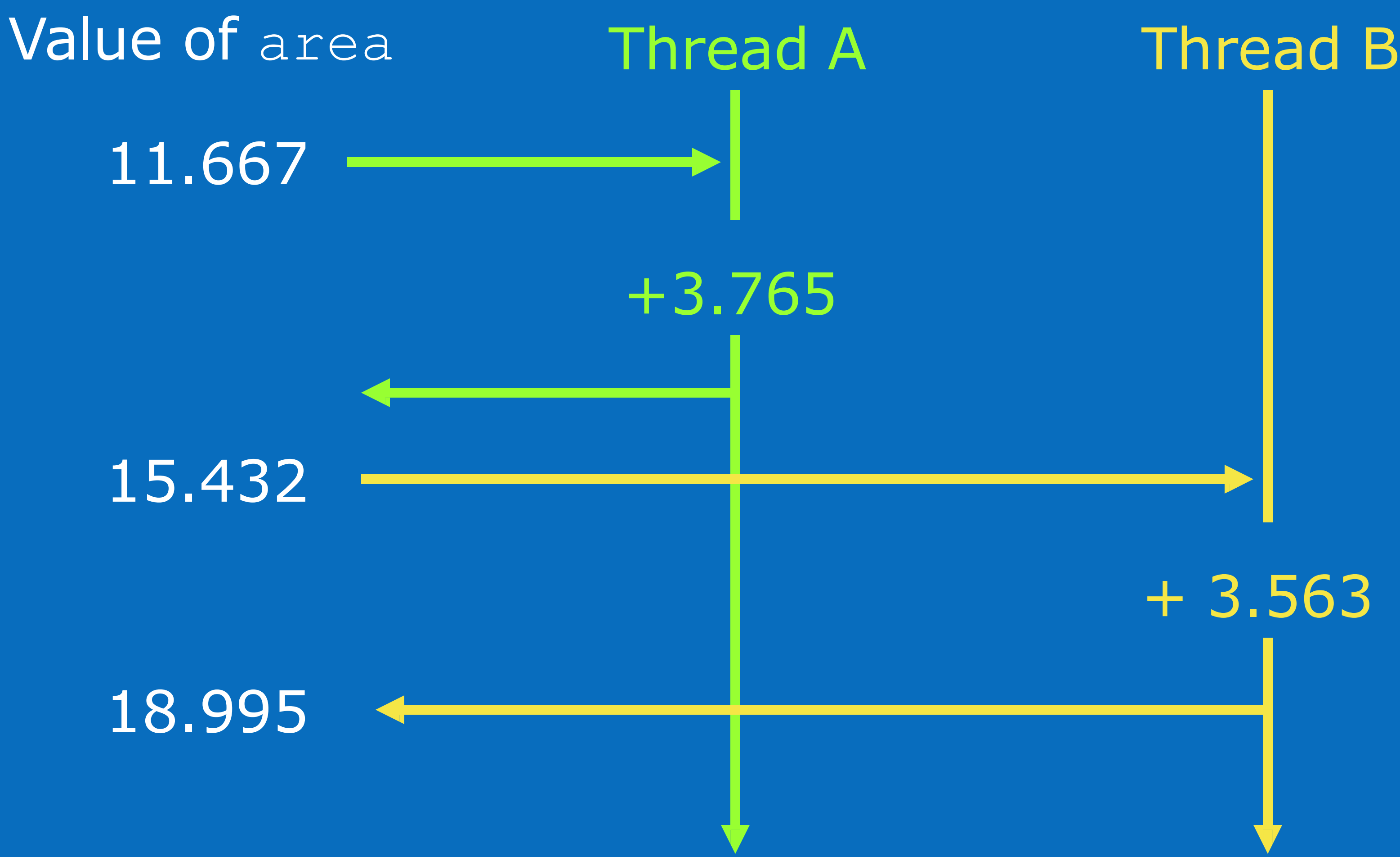
Race Condition

A *race condition* is nondeterministic behavior caused by the times at which two or more threads access a shared variable

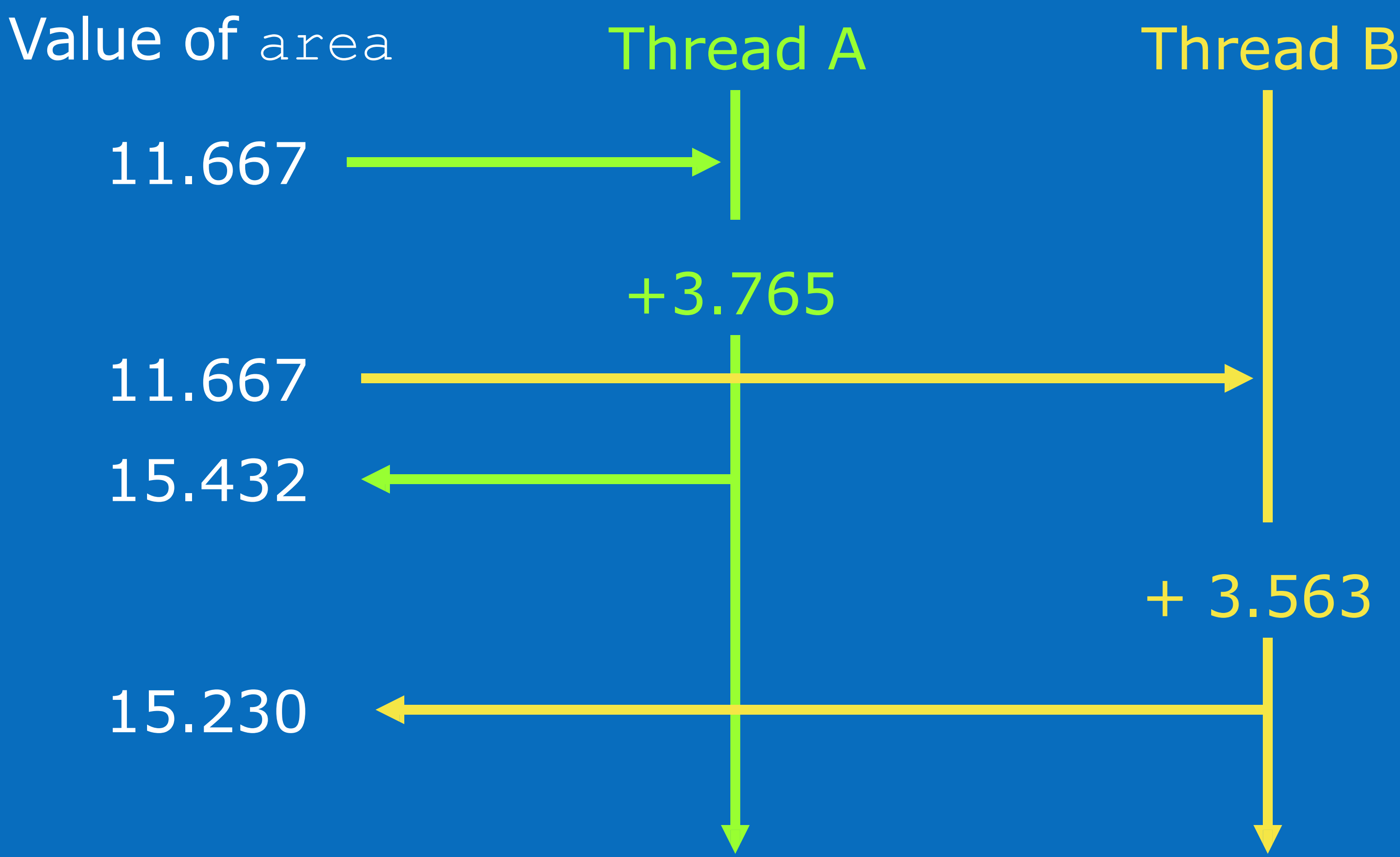
For example, suppose both Thread A and Thread B are executing the statement

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

One Timing ⇒ Correct Sum



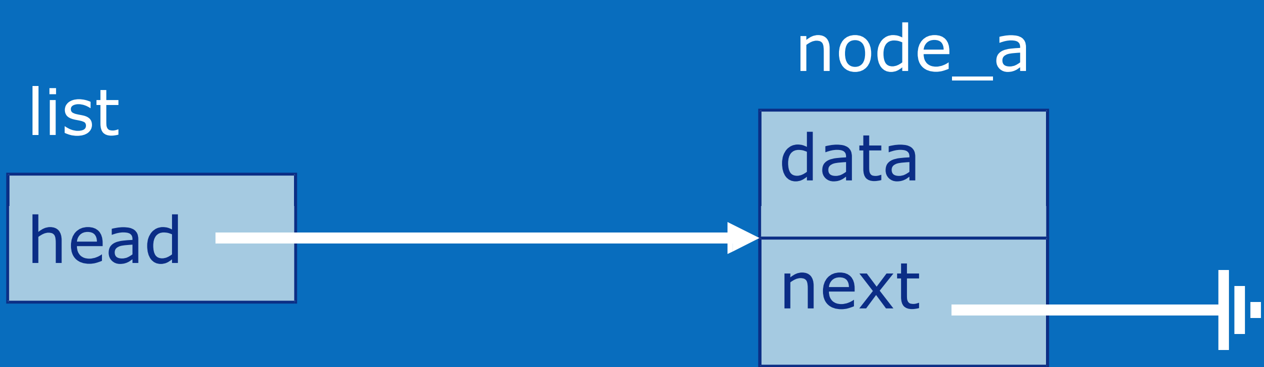
Another Timing ⇒ Incorrect Sum



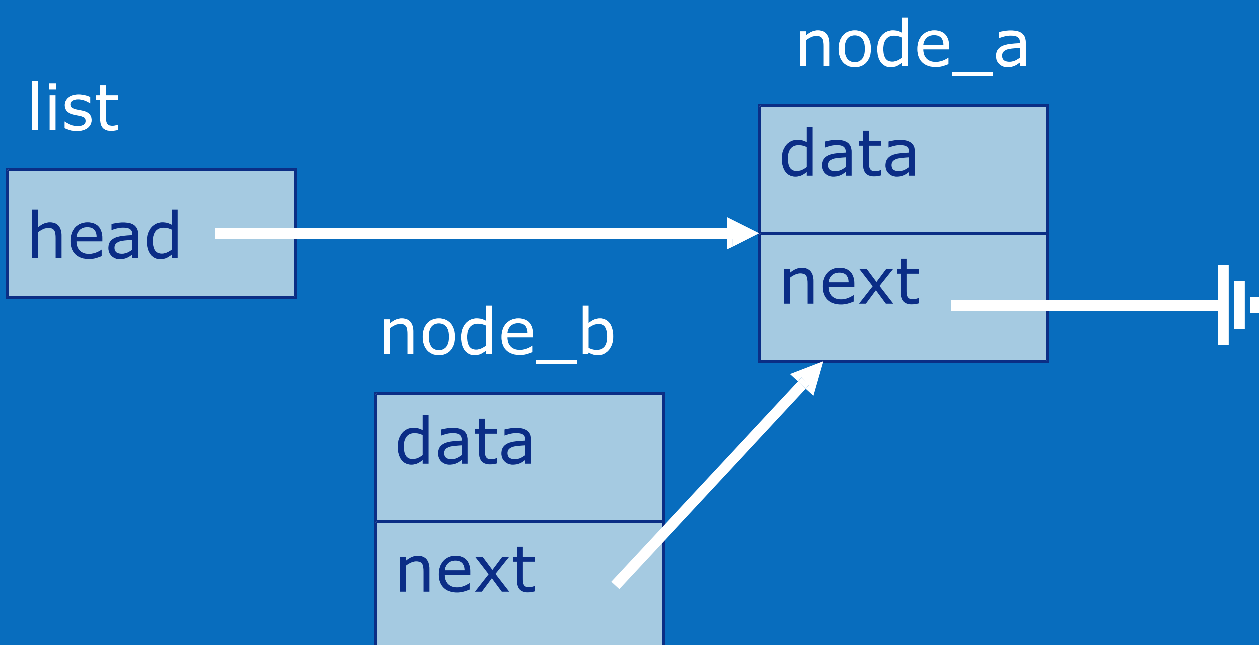
Another Race Condition Example

```
struct Node {
    struct Node *next;
    int data;   };

struct List {
    struct Node *head; }

void AddHead (struct List *list,
              struct Node *node) {
    node->next = list->head;
    list->head = node;
}
```

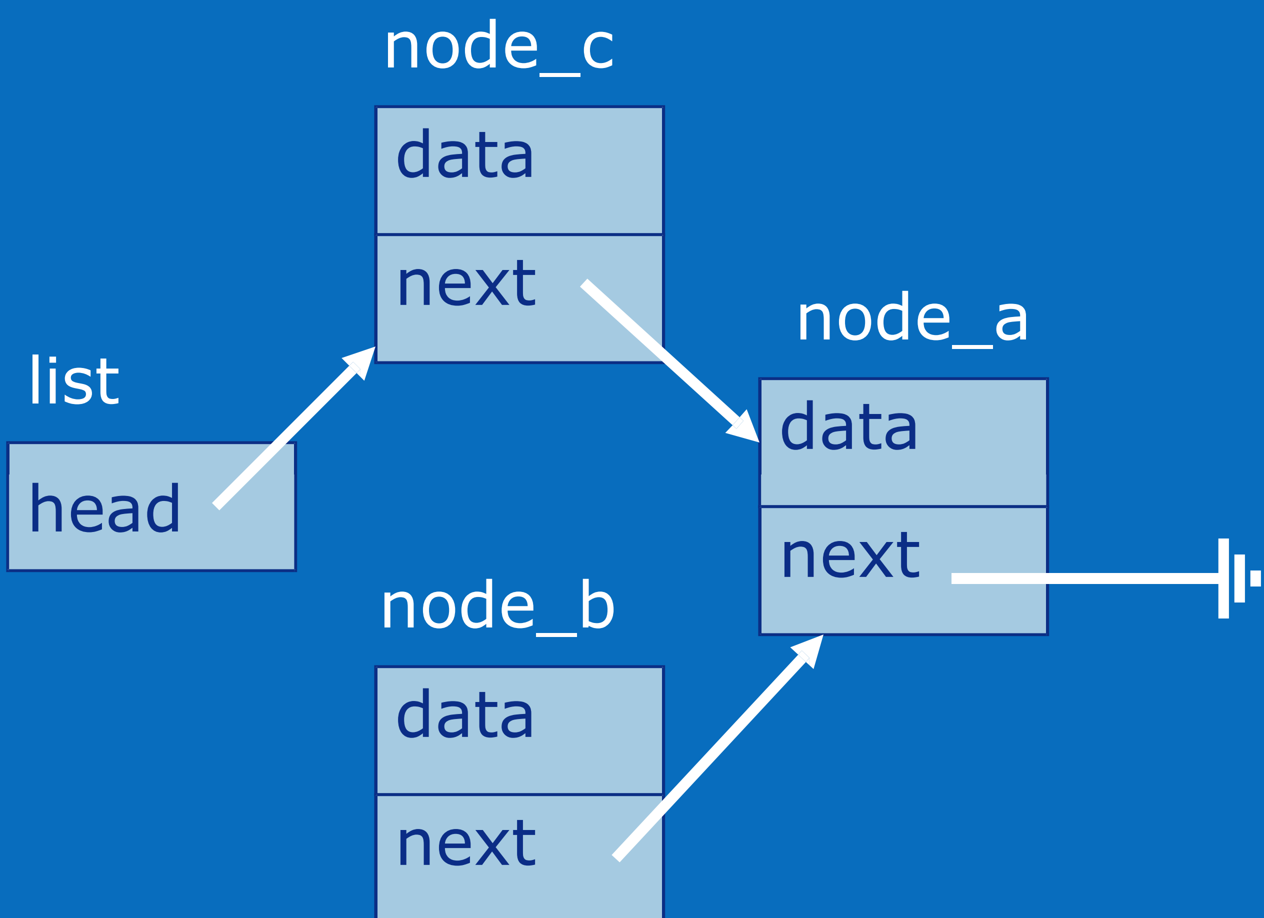
Original Singly-Linked List



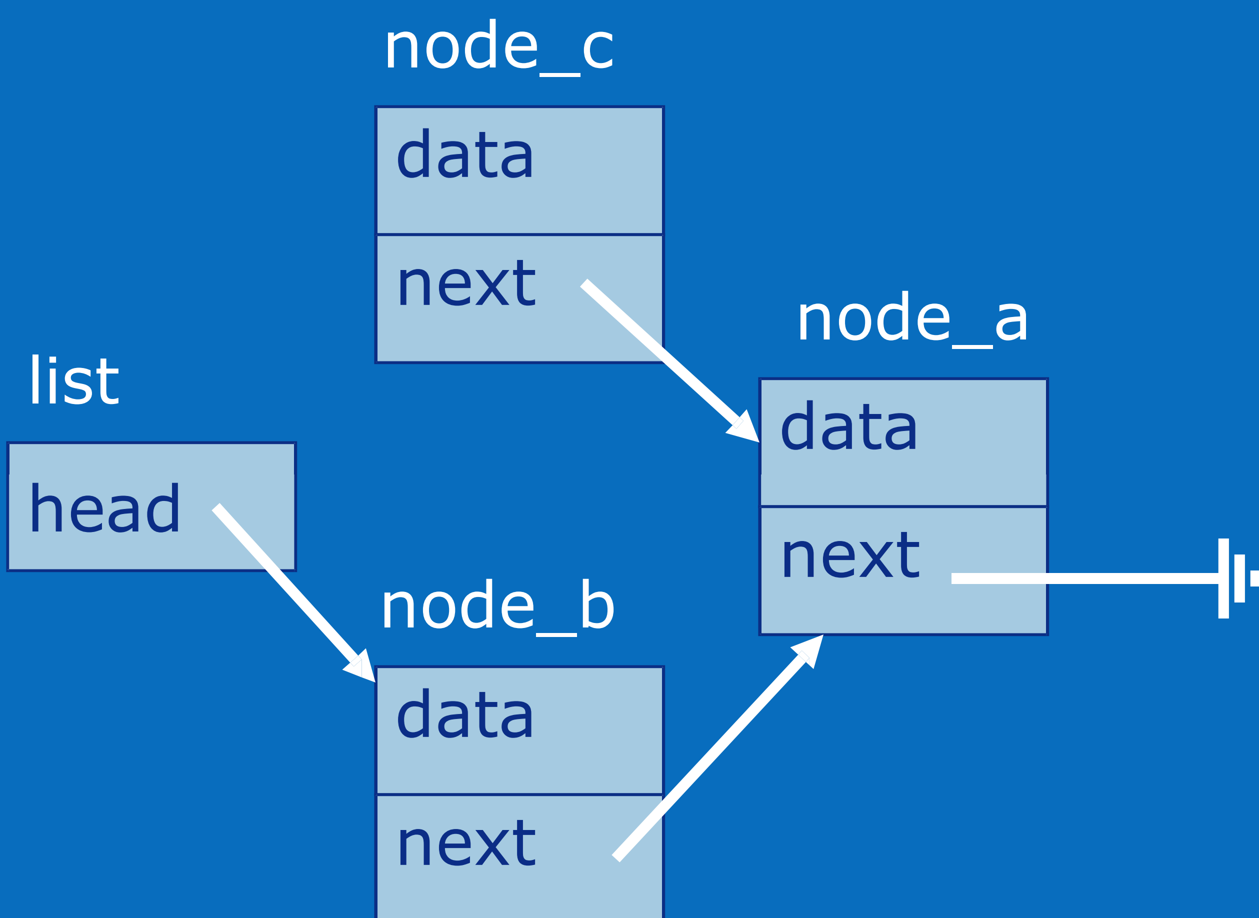
Thread 1 after Stmt. 1 of AddHead



Thread 2 Executes AddHead



Thread 1 After Stmt. 2 of AddHead



Why Race Conditions Are Nasty

- Programs with race conditions exhibit nondeterministic behavior
  - Sometimes give correct result
  - Sometimes give erroneous result
- Programs often work correctly on trivial data sets and small number of threads
- Errors more likely to occur when number of threads and/or execution time increases
- Hence debugging race conditions can be difficult

Mutual Exclusion

We can prevent the race conditions described earlier by ensuring that only one thread at a time references and updates shared variable or data structure

Mutual exclusion refers to a kind of synchronization that allows only a single thread or process at a time to have access to a shared resource

Mutual exclusion is implemented using some form of locking

Do Flags Guarantee Mutual Exclusion?

```
int flag = 0;

void AddHead (struct List *list,
              struct Node *node) {
   while (flag != 0) /* wait */ ;
   flag = 1;
   node->next = list->head;
   list->head = node;
   flag = 0;
}
```

Flags Don't Guarantee Mutual Exclusion

flag

0

Thread 1

```
int flag = 0;

void AddHead (struct List *list,
              struct Node *node) {
   while (flag != 0) /* wait */ ;
   flag = 1;
   node->next = list->head;
   list->head = node;
   flag = 0;
}
```

Flags Don’t Guarantee Mutual Exclusion

flag

0

Thread 1 Thread 2

```
int flag = 0;

void AddHead (struct List *list,
              struct Node *node) {
    while (flag != 0) /* wait */ ;
    flag = 1;
    node->next = list->head;
    list->head = node;
    flag = 0;
}
```

Flags Don’t Guarantee Mutual Exclusion

flag

1

Thread 1 Thread 2

```
int flag = 0;

void AddHead (struct List *list,
              struct Node *node) {
   while (flag != 0) /* wait */ ;
   flag = 1;
   node->next = list->head;
   list->head = node;
   flag = 0;
}
```

Flags Don’t Guarantee Mutual Exclusion


```
int flag = 0;

void AddHead (struct List *list,
              struct Node *node) {
   while (flag != 0) /* wait */ ;
   flag = 1;
   node->next = list->head;
   list->head = node;
   flag = 0;
}
```

Flags Don’t Guarantee Mutual Exclusion

flag

0

Thread 1 Thread 2

```
int flag = 0;

void AddHead (struct List *list,
              struct Node *node) {
  while (flag != 0) /* wait */ ;
  flag = 1;
  node->next = list->head;
  list->head = node;
  flag = 0;
}
```

Flags Don’t Guarantee Mutual Exclusion

flag | Thread 1 | Thread 2

0

```
int flag = 0;

void AddHead (struct List *list,
              struct Node *node) {
   while (flag != 0) /* wait */ ;
   flag = 1;
   node->next = list->head;
   list->head = node;
   flag = 0;
}
```

Locking Mechanism

The previous method failed because checking the value of `flag` and setting its value were two distinct operations

We need some sort of *atomic* test-and-set

Operating system provides functions to do this

The generic term “lock” refers to a synchronization mechanism used to control access to shared resources

Critical Sections

A *critical section* is a portion of code that threads execute in a mutually exclusive fashion

The `critical` pragma in OpenMP immediately precedes a statement or block representing a critical section

Good news: critical sections eliminate race conditions

Bad news: critical sections are executed sequentially

More bad news: you have to identify critical sections yourself

Reminder: Motivating Example

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
   x = (i + 0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Where is the critical section?

Solution #1

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
   x = (i + 0.5)/n;
#pragma omp critical
   area += 4.0 / (1.0 + x*x);
}
pi = area / n;
```

This ensures `area` will end up with the correct value.
How can we do better?

Solution #2

```
double area, pi, tmp, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x,tmp)
for (i = 0; i < n; i++) {
    x = (i + 0.5)/n;
    tmp = 4.0/(1.0 + x*x);
#pragma omp critical
    area += tmp;
}
pi = area / n;
```

This reduces amount of time spent in critical section.
How can we do better?

Solution #3

```
double area, pi, tmp, x;
int i, n;
...
area = 0.0;
#pragma omp parallel private(tmp)
{
   tmp = 0.0;
#pragma omp for private (x)
   for (i = 0; i < n; i++) {
      x = (i + 0.5)/n;
      tmp += 4.0/(1.0 + x*x);
   }
#pragma omp critical
   area += tmp;
}
pi = area / n;
```

Why is this better?

Reductions

Given associative binary operator $\oplus$ the expression

$a_1 \oplus a_2 \oplus a_3 \oplus \ldots \oplus a_n$

is called a *reduction*

The π-finding program performs a sum-reduction

OpenMP `reduction` Clause

Reductions are so common that OpenMP provides a `reduction` clause for the `parallel for` pragma

Eliminates need for

- Creating private variable
- Dividing computation into accumulation of local answers that contribute to global result

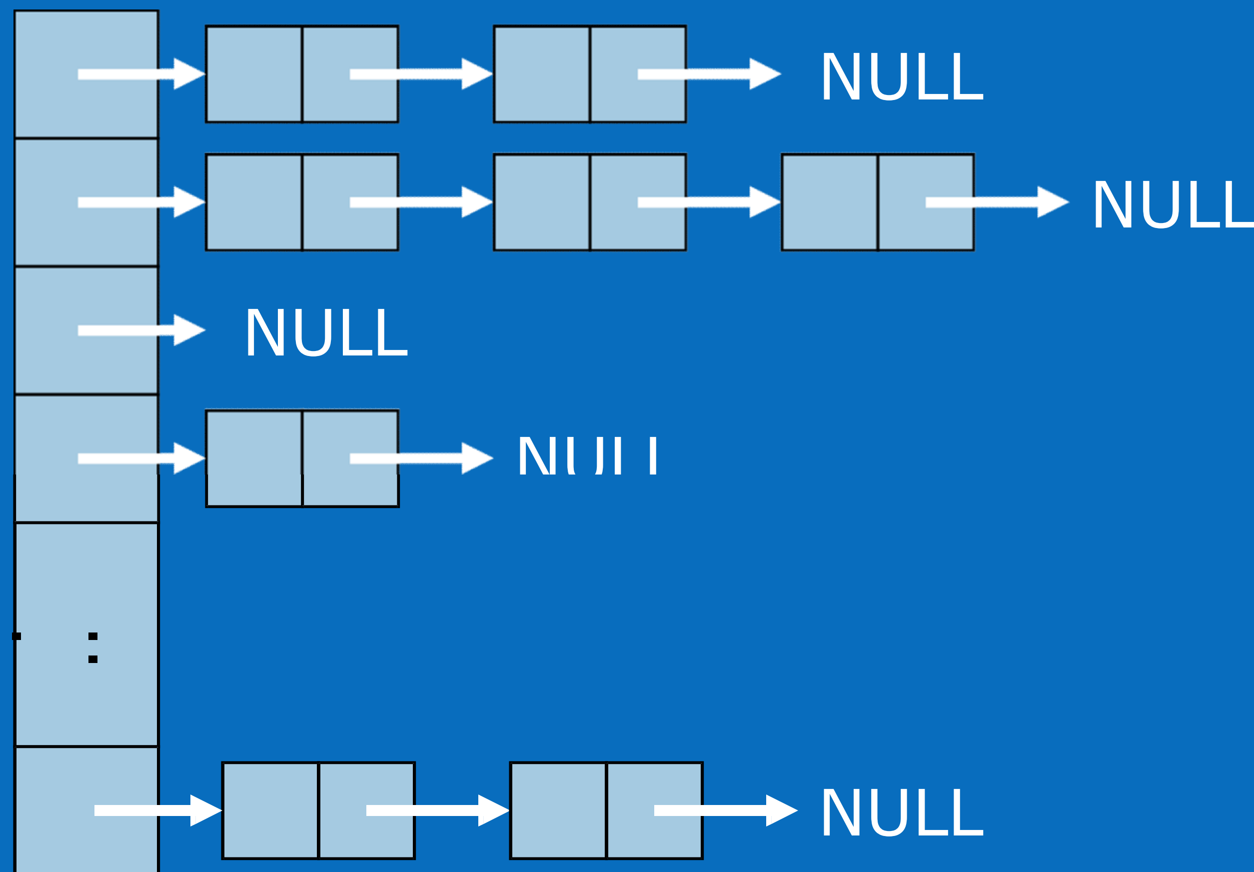
Solution #4

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x) \
                         reduction(+:area)
for (i = 0; i < n; i++) {
    x = (i + 0.5)/n;
    area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Important: Lock Data, Not Code

- Locks should be associated with data objects
- Different data objects should have different locks
- Suppose lock associated with critical section of code instead of data object
  - Mutual exclusion can be lost if same object manipulated by two different functions
  - Performance can be lost if two threads manipulating different objects attempt to execute same function

Example: Hash Table Creation



Locking Code: Inefficient

```
#pragma omp parallel for private (index)
for (i = 0; i < elements; i++) {
   index = hash(element[i]);
   #pragma omp critical
   insert_element (element[i], index);
}
```

Locking Data: Efficient

Declaration

```
/* Static variable */
omp_lock_t hash_lock[HASH_TABLE_SIZE];
```

Initialization

```
/* Inside function 'main' */
for (i = 0; i < HASH_TABLE_SIZE; i++)
   omp_init_lock(&hash_lock[i]);
```

Use

```
void insert_element (ELEMENT e, int i)
{
   omp_set_lock (&hash_lock[i]);
   /* Code to insert element e */
   omp_unset_lock (&hash_lock[i]);
}
```

Locks Are Dangerous

Suppose a lock is used to guarantee mutually exclusive access to a shared variable

Imagine two threads, each with its own critical section

Thread A	Thread B
`a += 5;`	`b += 5;`
`b += 7;`	`a += 7;`
`a += b;`	`a += b;`
`a += 11;`	`b += 11;`

Faulty Implementation

What happens if threads are at this point at the same time?

Thread A

```
lock (lock_a);
a += 5;
-----------------
lock (lock_b);
b += 7;
a += b;
unlock (lock_b);
a += 11;
unlock (lock_a);
```

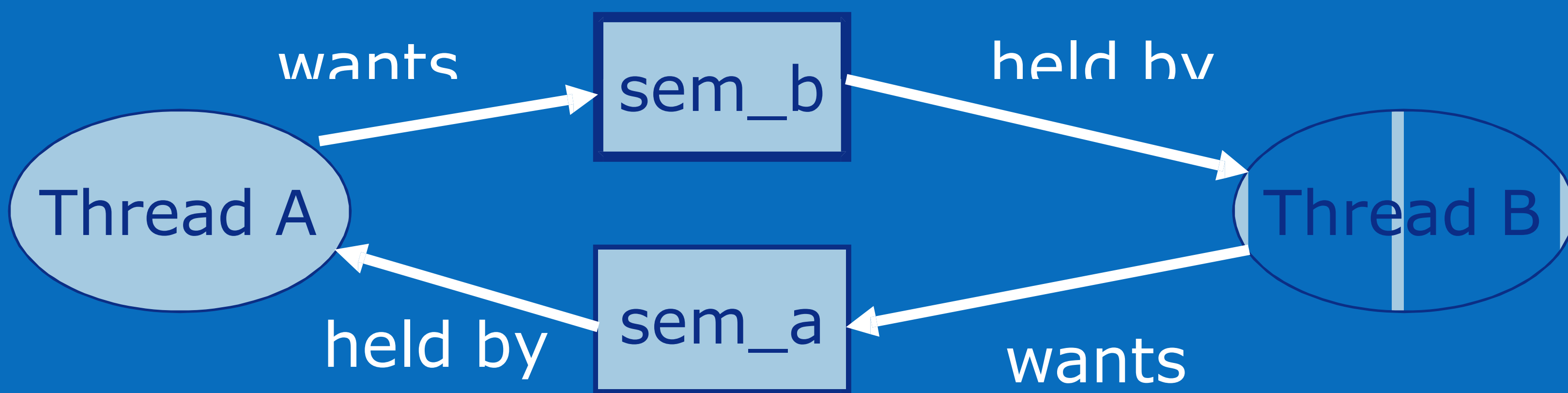
Thread B

```
lock (lock_b);
b += 5;
-----------------
lock (lock_a);
a += 7;
a += b;
unlock (lock_a);
b += 11;
unlock (lock_b);
```

Deadlock

A situation involving two or more threads (processes) in which no thread may proceed because each is waiting for a resource held by another

Can be represented by a resource allocation graph



A graph of deadlock contains a cycle

More on Deadlocks

A program exhibits a *global deadlock* if every thread is blocked

A program exhibits *local deadlock* if only some of the threads in the program are blocked

A deadlock is another example of a nondeterministic behavior exhibited by a parallel program

Adding debugging output to detect source of deadlock can change timing and reduce chance of deadlock occurring

Four Conditions for Deadlock

Mutually exclusive access to a resource

Threads hold onto resources they have while they wait for additional resources

Resources cannot be taken away from threads

Cycle in resource allocation graph

Deadlock Prevention Strategies

Don’t allow mutually exclusive access to resource	Make resource shareable
Don’t allow threads to wait while holding resources	Only request resources when have none. That means only hold one resource at a time or request all resources at once.
Allow resources to be taken away from threads.	Allow preemption. Works for CPU and memory. Doesn’t work for locks.
Ensure no cycle in request allocation graph.	Rank resources. Threads must acquire resources in order.

Correct Implementation

Threads must lock lock_a before lock_b

Thread A

```
lock (lock_a);
a += 5;
lock (lock_b);
b += 7;
a += b;
unlock (lock_b);
a += 11;
unlock (lock_a);
```

Thread B

```
lock (lock_a);
lock (lock_b);
b += 5;
a += 7;
a += b;
unlock (lock_a);
b += 11;
unlock (lock_b);
```

Another Problem with Locks

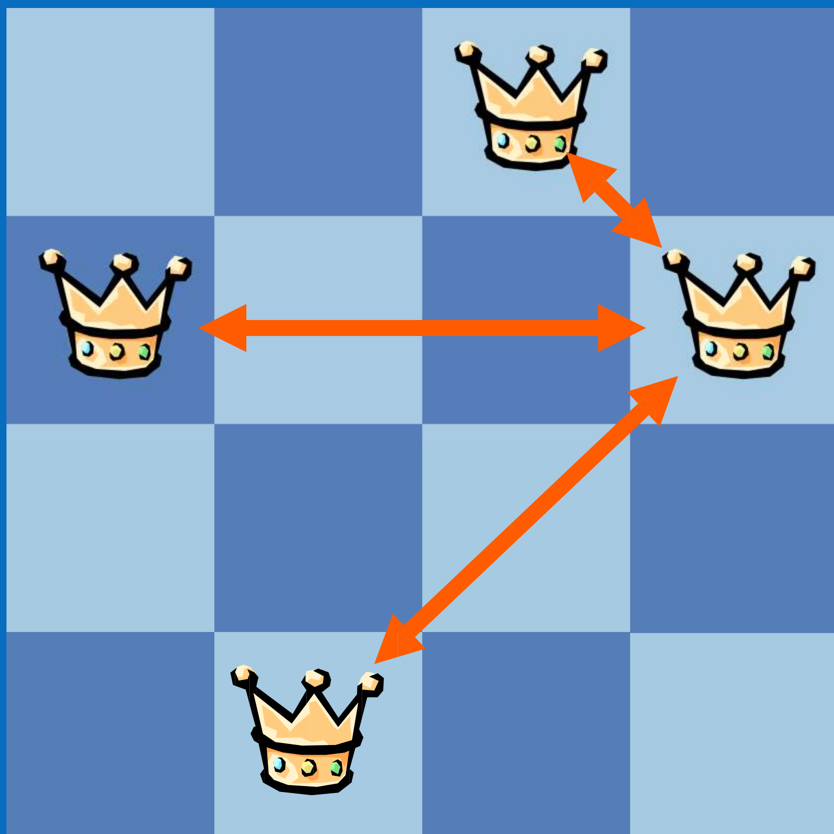
Every call to function `lock` should be matched with a call to `unlock`, representing the start and the end of the critical section

A program may be syntactically correct (i.e., may compile) without having matching calls

A programmer may forget the `unlock` call or may pass the wrong argument to `unlock`

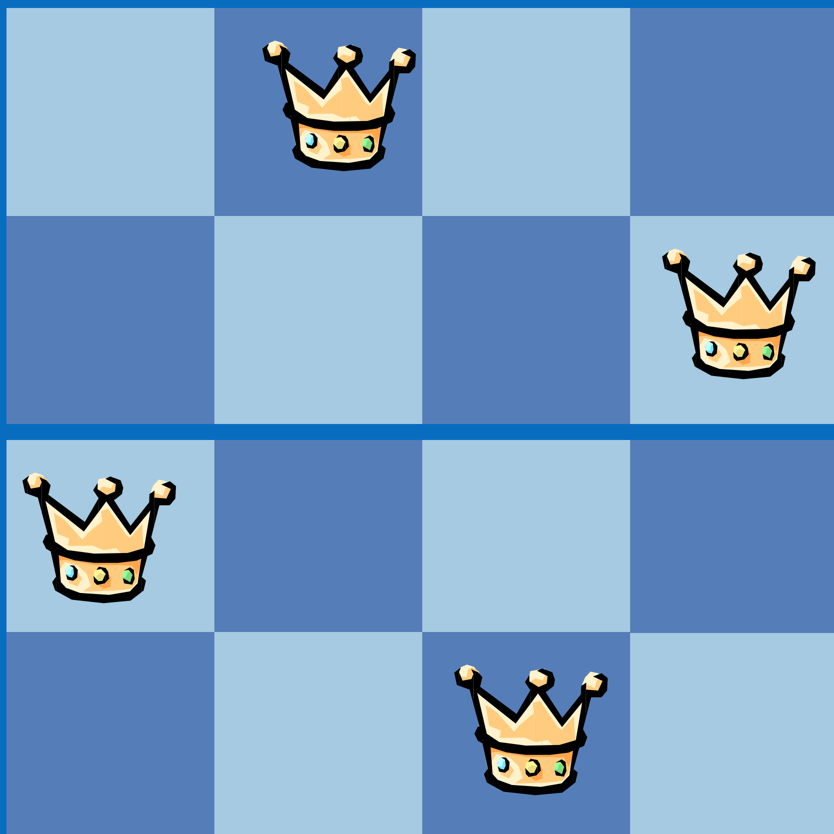
A thread that never releases a shared resource creates a deadlock

Case Study: The N Queens Problem

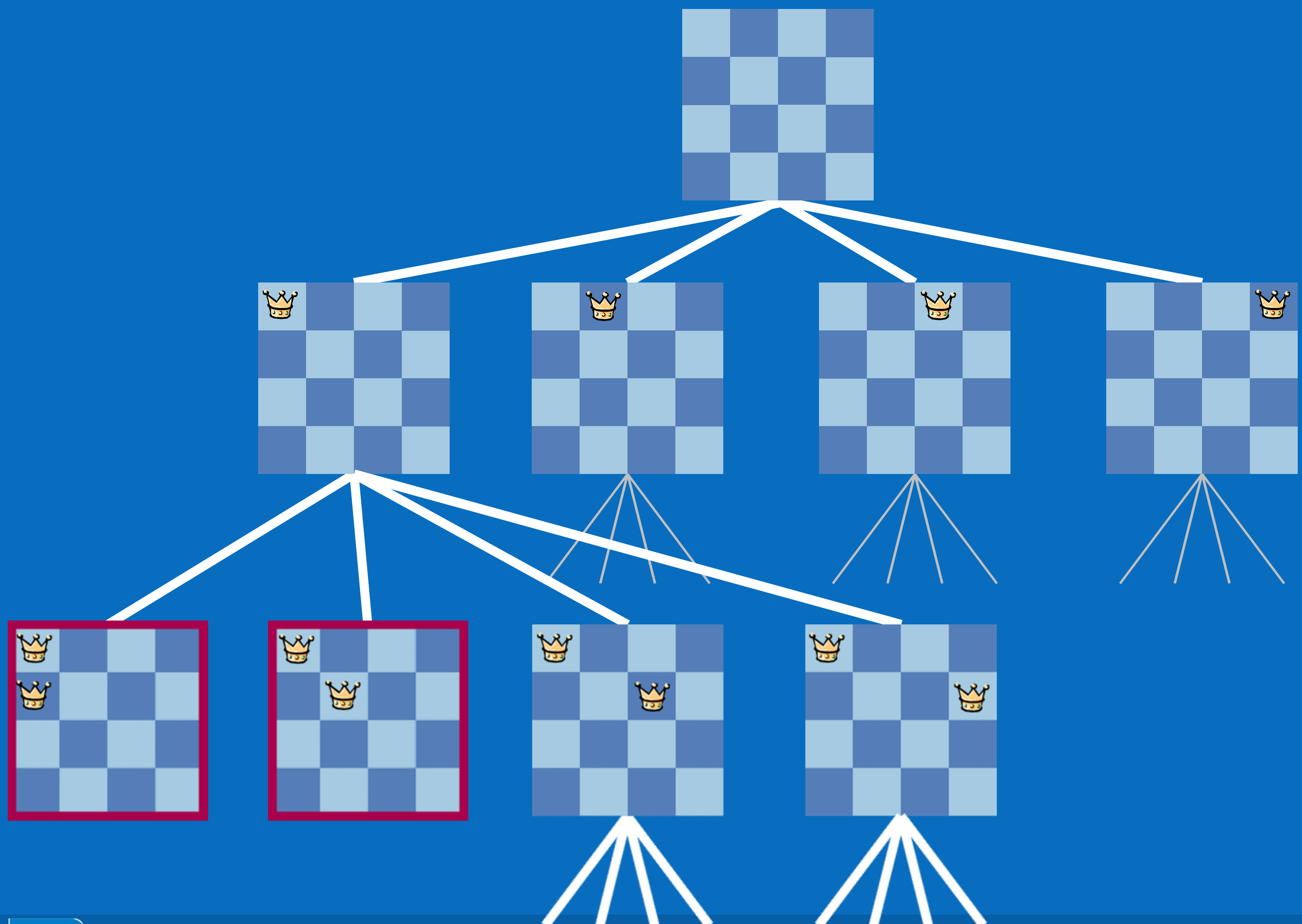


Is there a way to place N queens on an N-by-N chessboard such that no queen threatens another queen?

A Solution to the 4 Queens Problem



Exhaustive Search



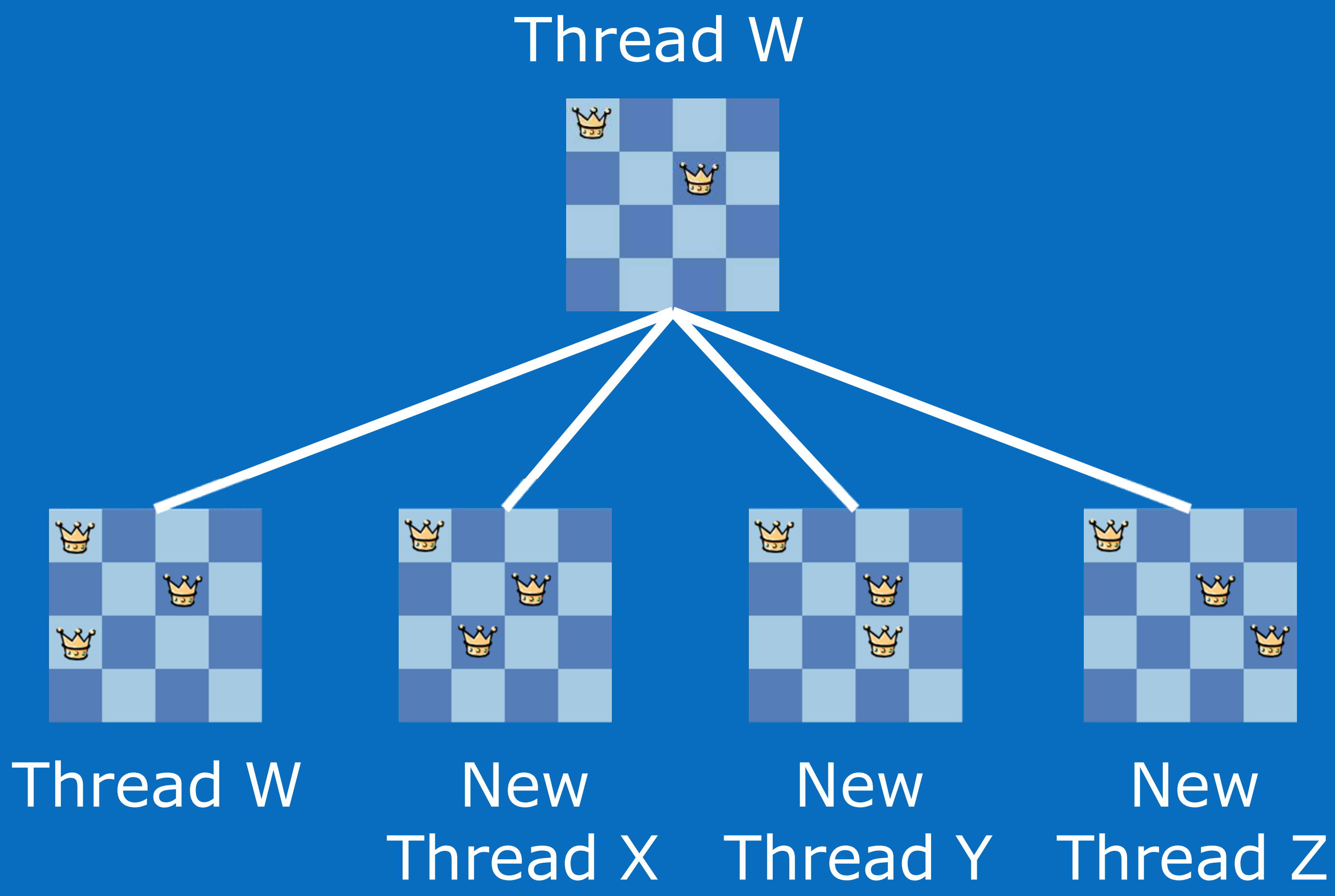
Design #1 for Parallel Search

Create threads to explore different parts of the search tree simultaneously

If a node has children

- The thread creates child nodes
- The thread explores one child node itself
- Thread creates a new thread for every other child node

Design #1 for Parallel Search



Pros and Cons of Design #1

Pros

- Simple design, easy to implement
- Balances work among threads

Cons

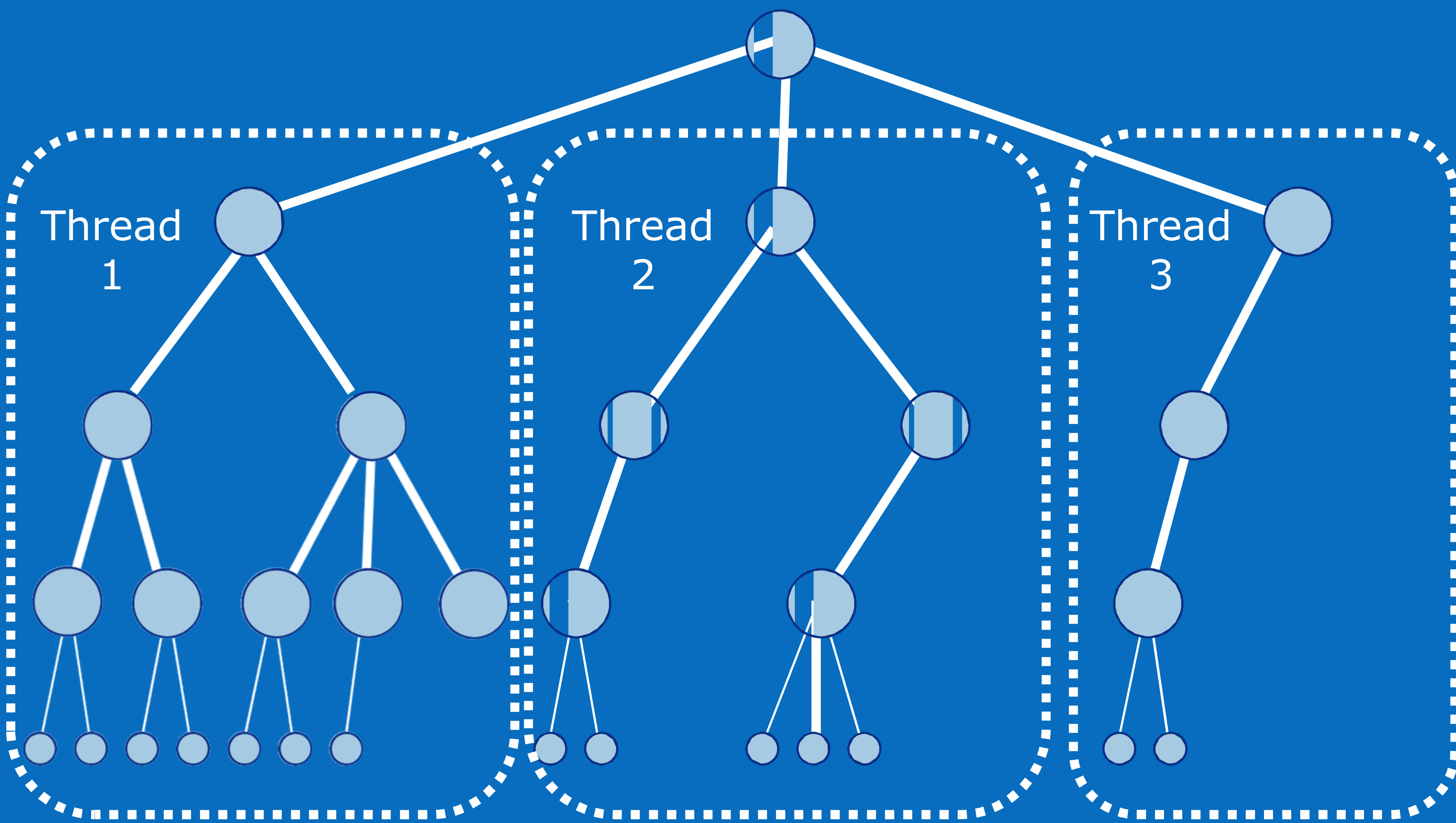
- Too many threads created
- Lifetime of threads too short
- Overhead costs too high

Design #2 for Parallel Search

One thread created for each subtree rooted at a particular depth

Each thread sequentially explores its subtree

Design #2 in Action



Pros and Cons of Design #2

Pros

- Thread creation/termination time minimized

Cons

- Subtree sizes may vary dramatically
- Some threads may finish long before others
- Imbalanced workloads lower efficiency

Design #3 for Parallel Search

- Main thread creates work pool—list of subtrees to explore
- Main thread creates finite number of co-worker threads
- Each subtree exploration is done by a single thread
- Inactive threads go to pool to get more work

Work Pool Analogy

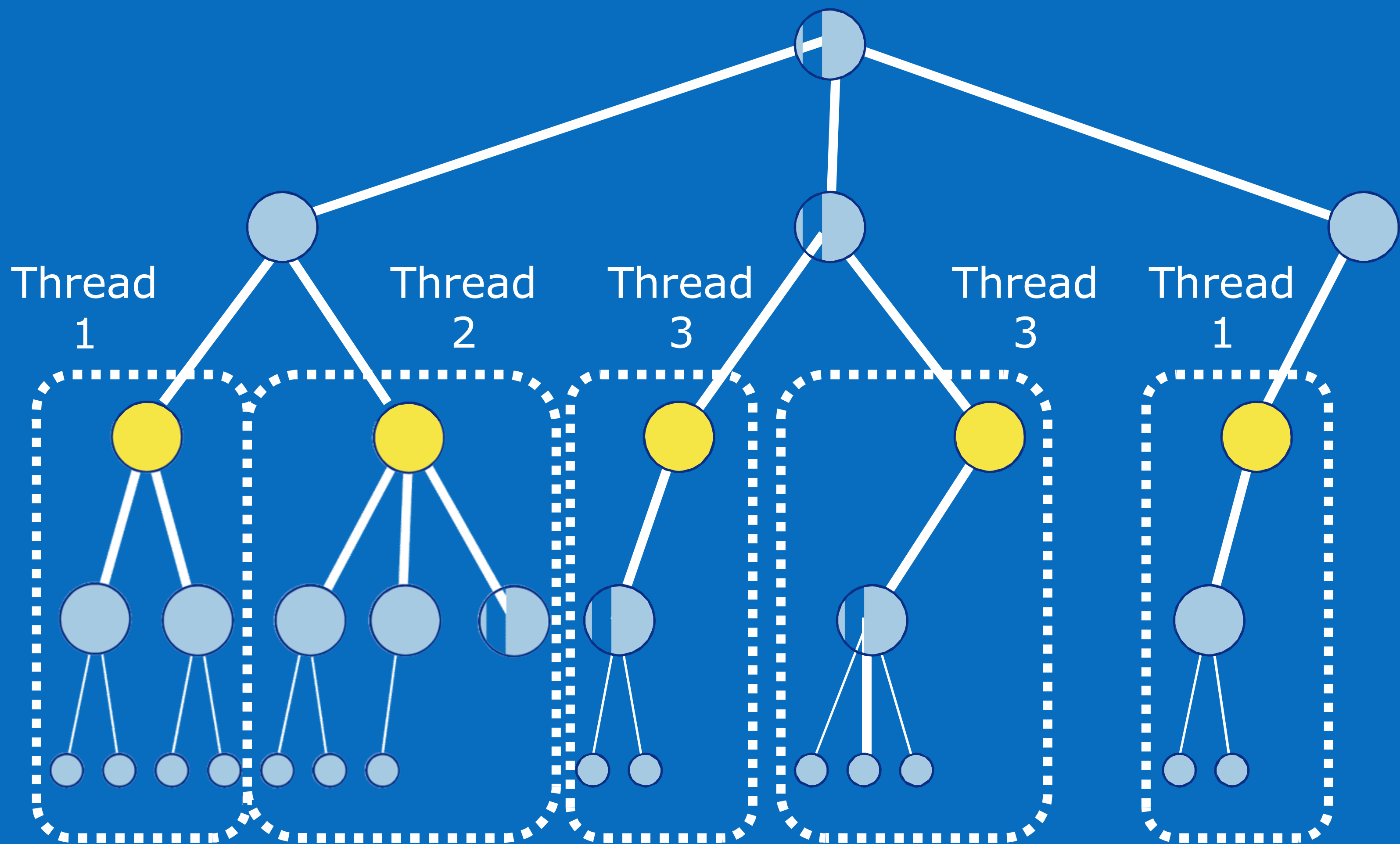
More rows than workers

Each worker takes an unpicked row and picks the crop

After completing a row, the worker takes another unpicked row

Process continues until all rows have been harvested

Design #3 in Action



Pros and Cons of Strategy #3

Pros

- Thread creation/termination time minimized
- Workload balance better than strategy #2

Cons

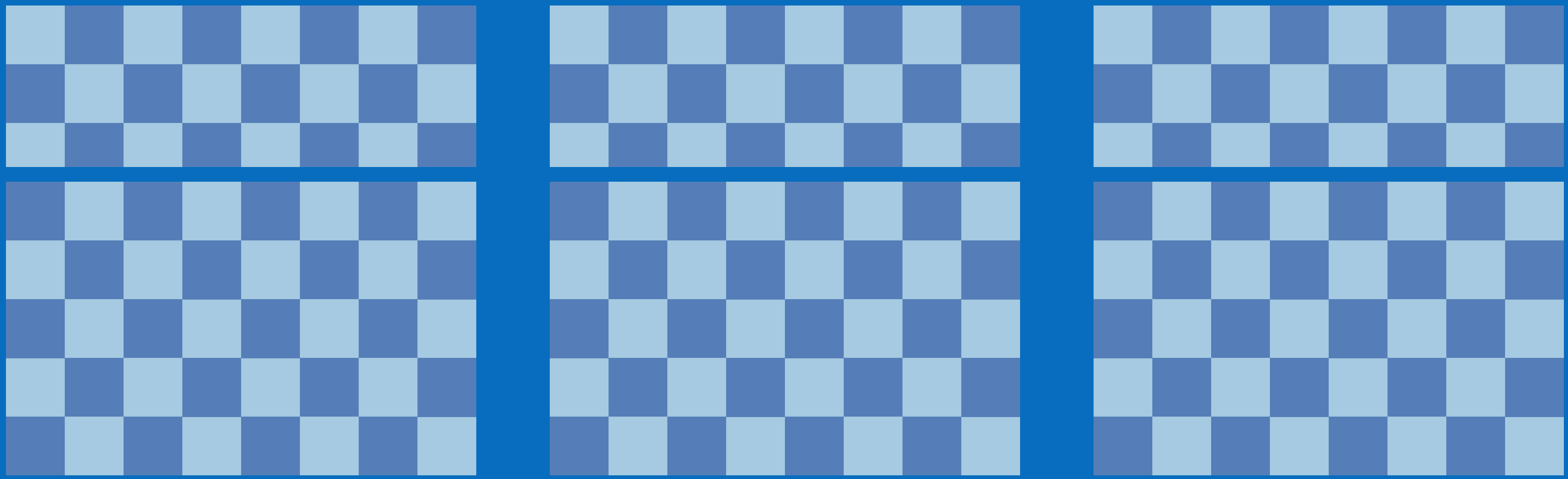
- Threads need exclusive access to data structure containing work to be done, a sequential component
- Workload balance worse than strategy #1

Conclusion

- Good compromise between designs 1 and 2

Implementing Strategy #3 for N Queens

Work pool consists of N boards representing N possible placements of queen on first row



Parallel Program Design

One thread creates list of partially filled-in boards

Fork: Create one thread per CPU

Each thread repeatedly gets board from list, searches for solutions, and adds to solution count, until no more board on list

Join: Occurs when list is empty

One thread prints number of solutions found

Search Tree Node Structure

```
/*    The 'board' struct contains information about a
      node in the search tree; i.e., partially filled-
      in board. The work pool is a singly linked
      list of 'board' structs. */

struct board {

   int pieces;            /* # of queens on board*/

   int places[MAX_N];     /* Queen's pos in each row */

   struct board *next;    /* Next search tree node */

};
```

Key Code in **main** Function

```
struct board *stack;
...
stack = NULL;
for (i = 0; i < n; i++) {
  initial=(struct board *)malloc(sizeof(struct board));
  initial->pieces = 1;
  initial->places[0] = i;
  initial->next = stack;
  stack = initial;
}
num_solutions = 0;
search_for_solutions (n, stack, &num_solutions);
printf ("The %d-queens puzzle has %d solutions\n", n,
            num_solutions);
```

Insertion of OpenMP Code

```
struct board *stack;
...
stack = NULL;
for (i = 0; i < n; i++) {
   initial=(struct board *)malloc(sizeof(struct board));
   initial->pieces = 1;
   initial->places[0] = i;
   initial->next = stack;
   stack = initial
}
num_solutions = 0;
omp_set_num_threads (omp_get_num_procs());
#pragma omp parallel
search_for_solutions (n, stack, &num_solutions);
printf ("The %d-queens puzzle has %d solutions\n", n,
            num_solutions);
```

Original C Function to Get Work

```
void search_for_solutions (int n,
   struct board *stack, int *num_solutions)
{
   struct board *ptr;
   void search (int, struct board *, int *);

   while (stack != NULL) {
      ptr = stack;
      stack = stack->next;
      search (n, ptr, num_solutions);
      free (ptr);
   }
}
```

C/OpenMP Function to Get Work

```
void search_for_solutions (int n,
   struct board *stack, int *num_solutions)
{
   struct board *ptr;
   void search (int, struct board *, int *);

   while (stack != NULL) {
     #pragma omp critical
     { ptr = stack; stack = stack->next; }
     search (n, ptr, num_solutions);
     free (ptr);
   }
}
```

Original C Search Function

```
void search (int n, struct board *ptr,
                  int *num_solutions)
{
   int i;
   int no_threats (struct board *);

   if (ptr->pieces == n) {
      (*num_solutions)++;
   } else
      ptr->pieces++;
      for (i = 0; i < n; i++) {
         ptr->places[ptr->pieces-1] = i;
         if (no_threats(ptr))
            search (n, ptr, num_solutions);
      }
      ptr->pieces--;
   }

}
```

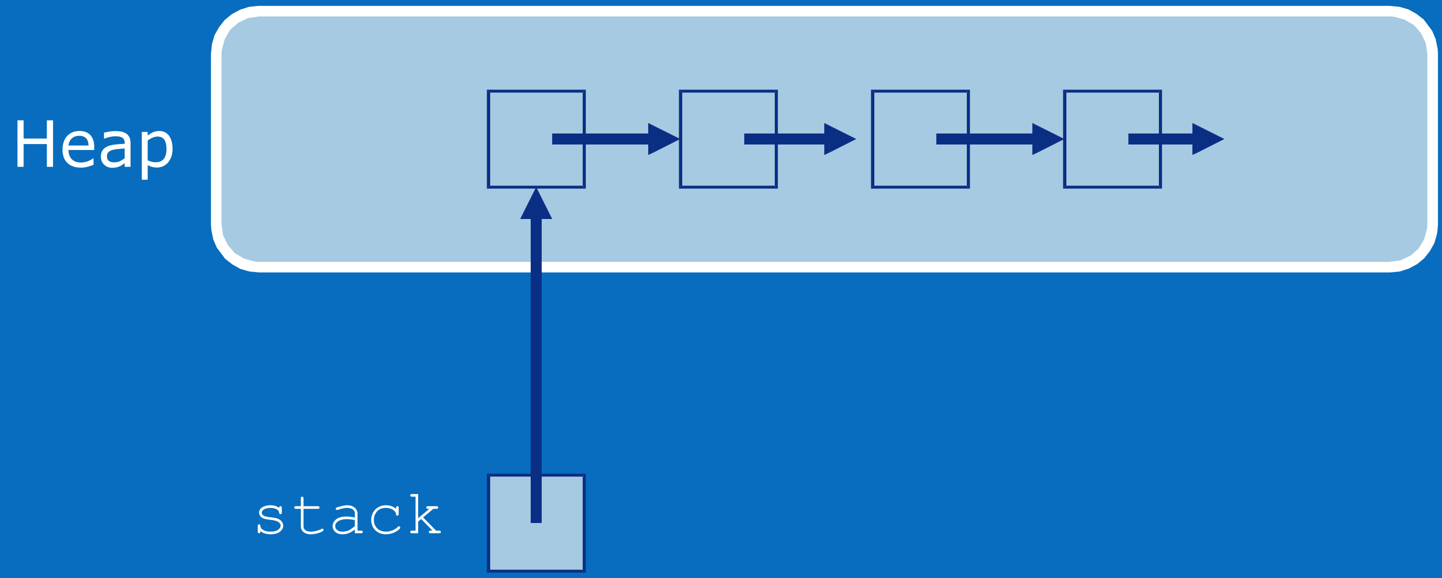
C/OpenMP Search Function

```
void search (int n, struct board *ptr,
                          int *num_solutions)
{
    int i;
    int no_threats (struct board *);

    if (ptr->pieces == n) {
        #pragma omp critical
        (*num_solutions)++;
    } else {
        ptr->pieces++;
        for (i = 0; i < n; i++) {
            ptr->places[ptr->pieces-1] = i;
            if (no_threats(ptr))
                search (n, ptr, num_solutions);
        }
        ptr->pieces--;
    }
}
```

Only One Problem: It Doesn't Work!

OpenMP program throws an exception

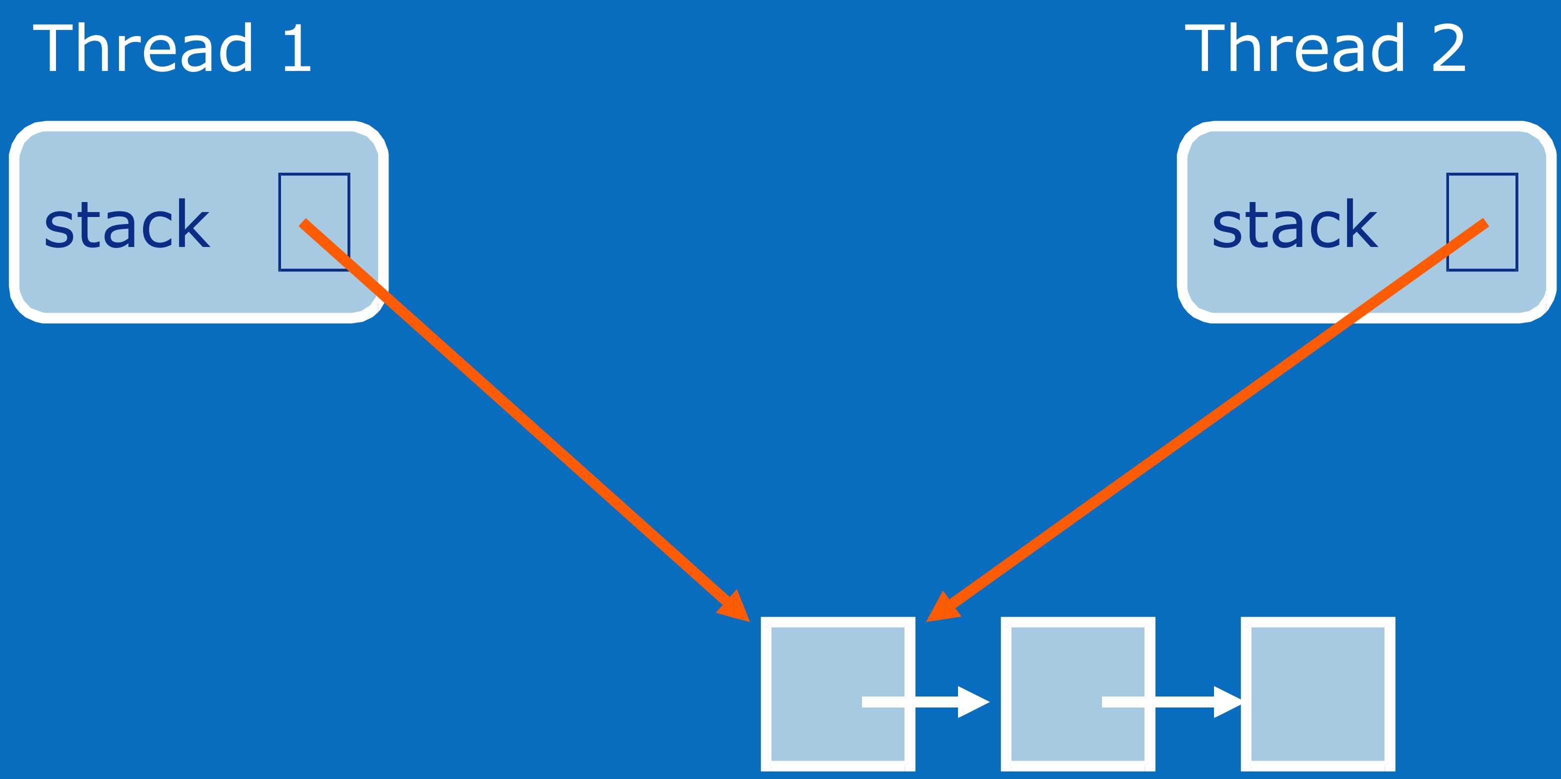
Culprit: Variable `stack`



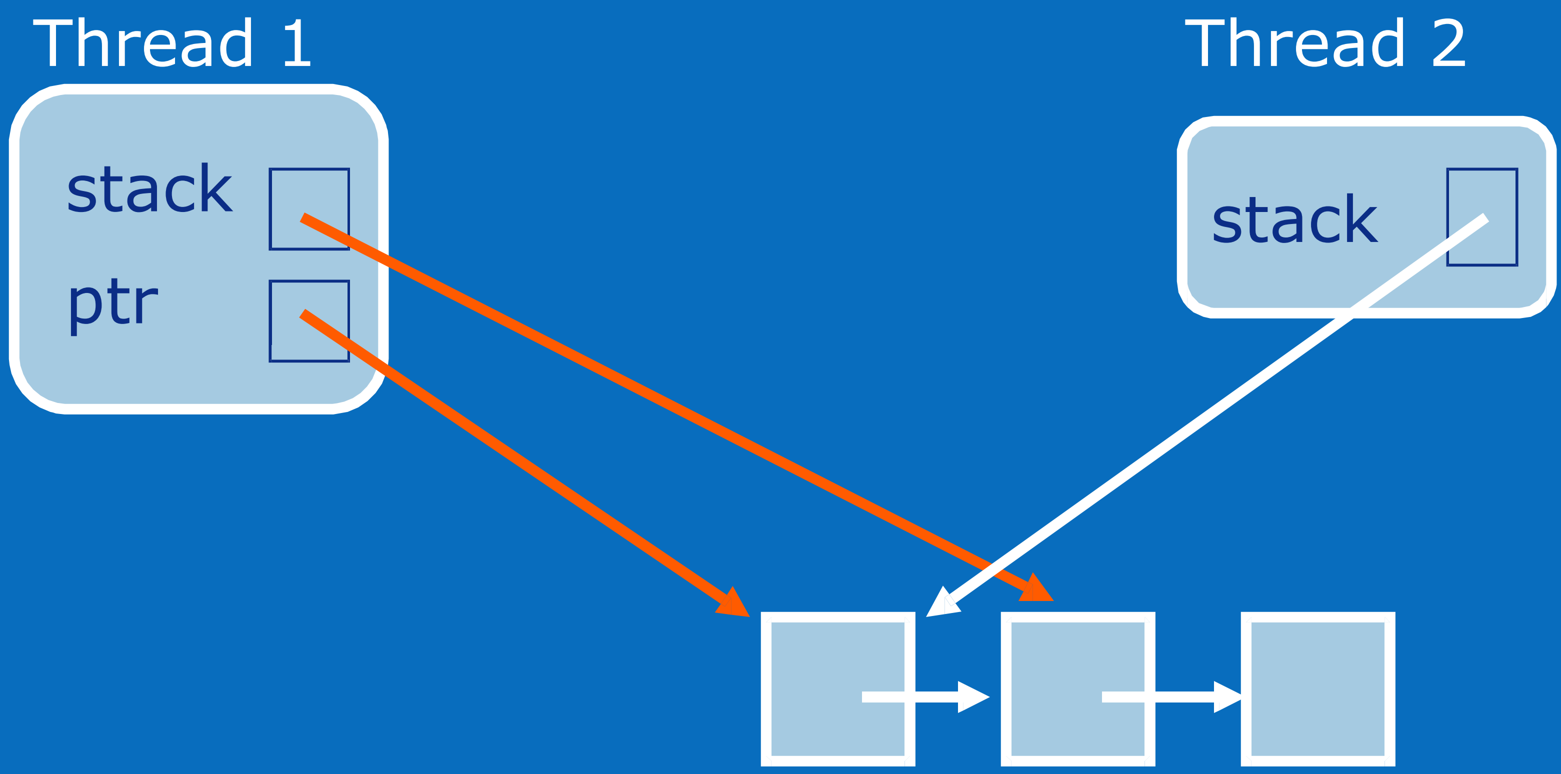
Problem Site

```
int main ()
{
   struct board *stack;
   ...
   #pragma omp parallel
   search_for_solutions
      (n, stack, &num_solutions);
   ...
}

void search_for_solutions (int n,
   struct board *stack, int *num_solutions)
{
   ...
   while (stack != NULL) ...
```

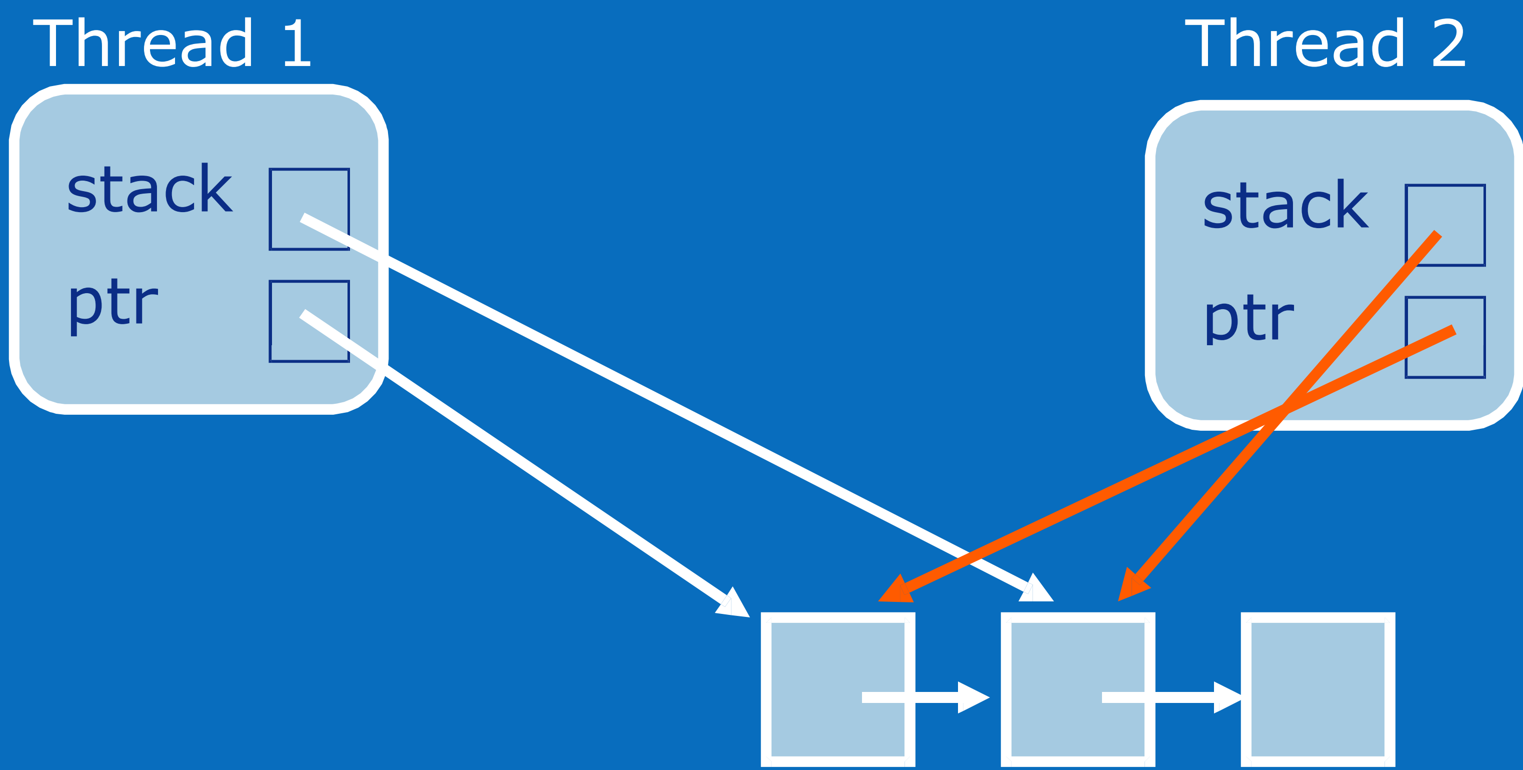
1. Both Threads Point to Top



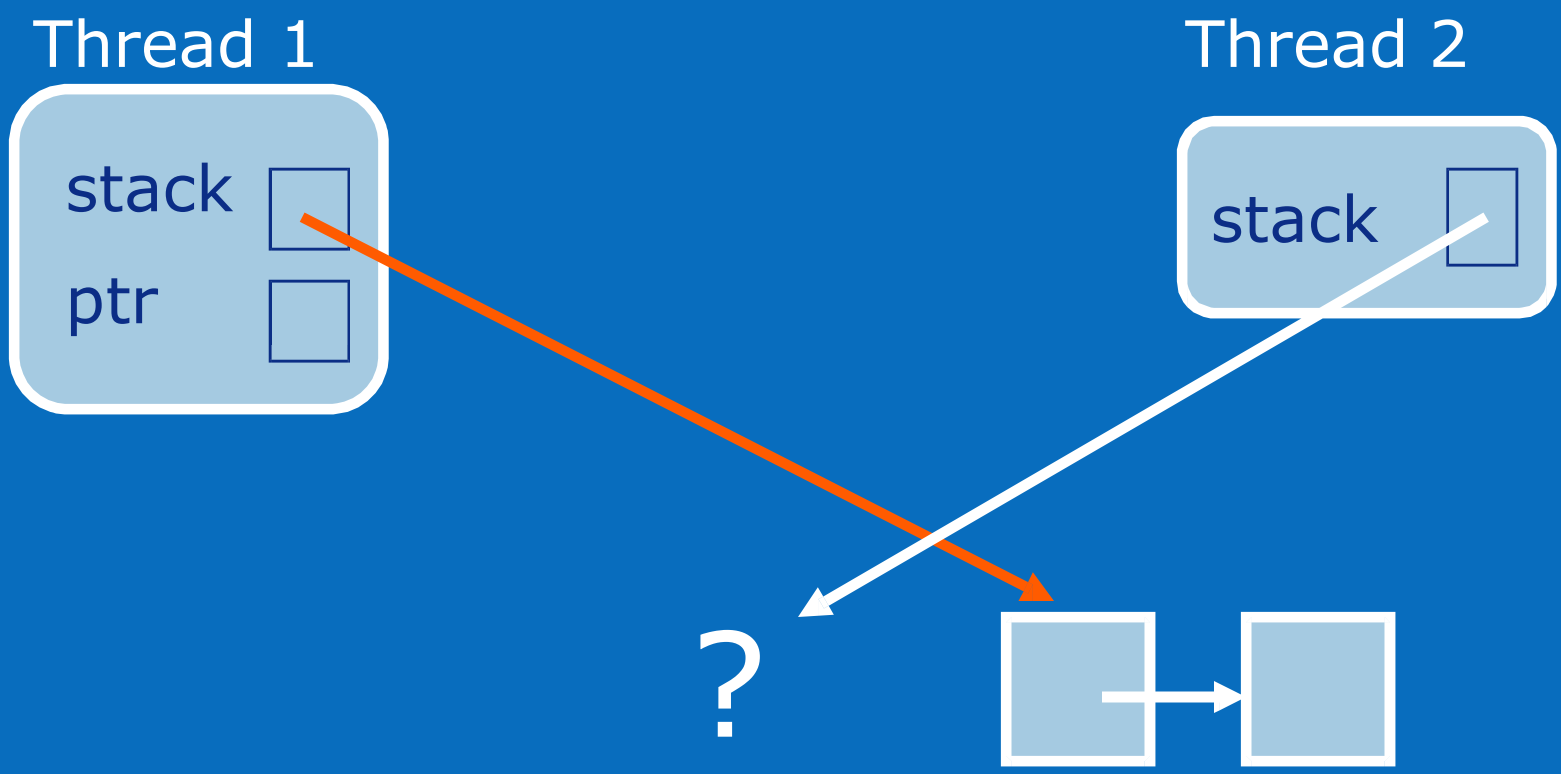
2. Thread 1 Grabs First Element



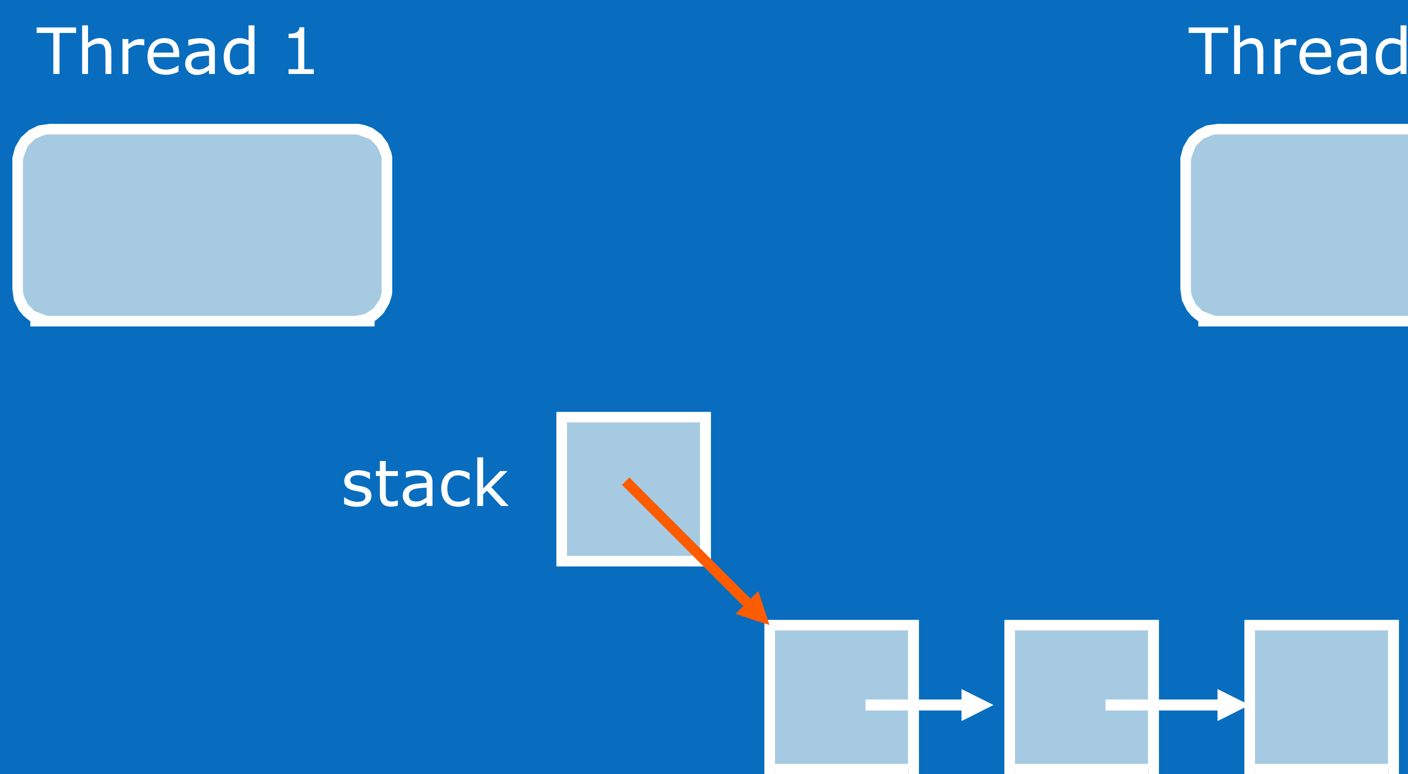
3. Error #1: Thread 2 grabs same element



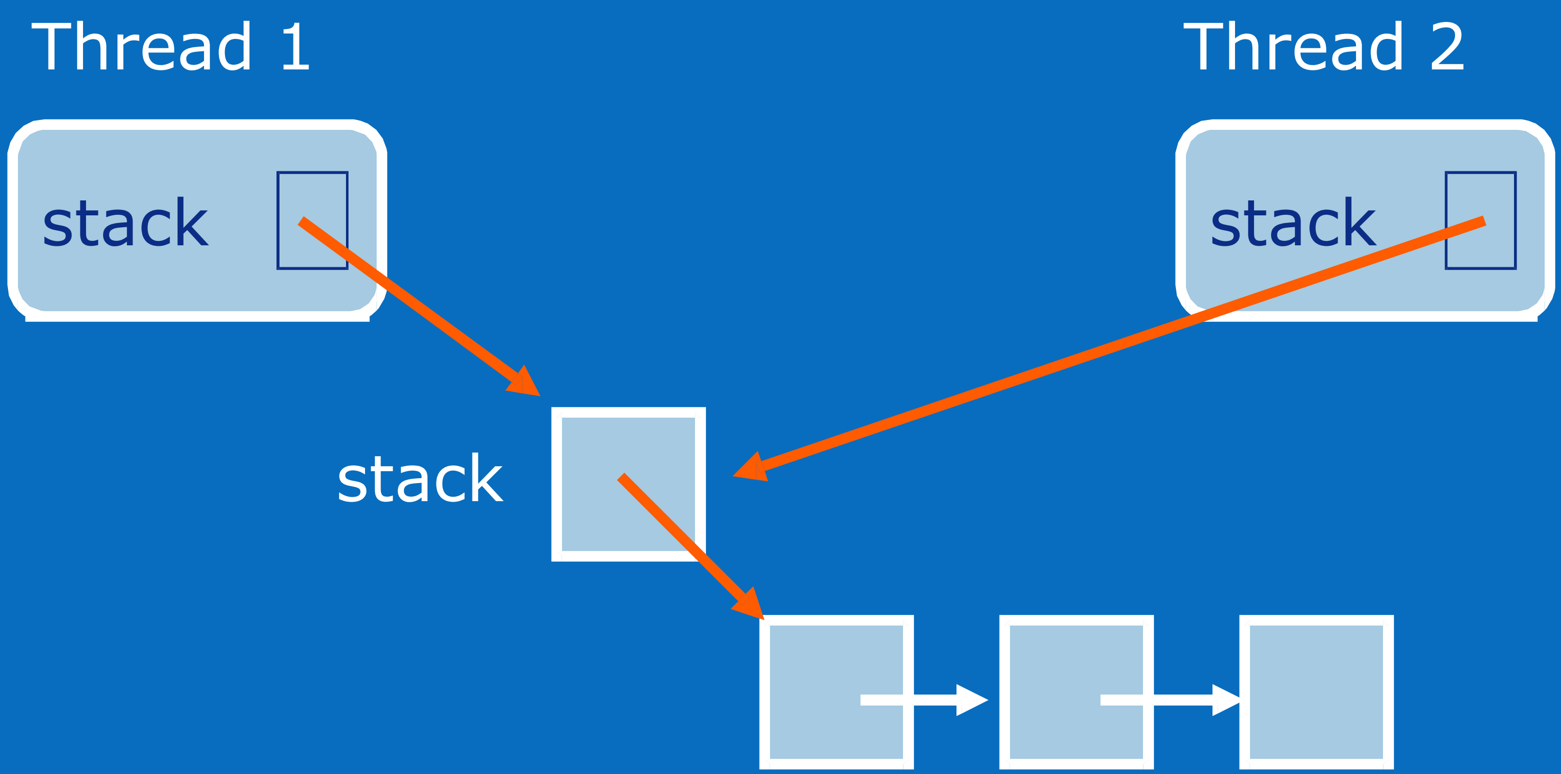
4. Error #2:
Thread 1 deletes element and then Thread 2's stack ptr dangles



Remedy 1: Make **stack** Static



Remedy 2: Use Indirection



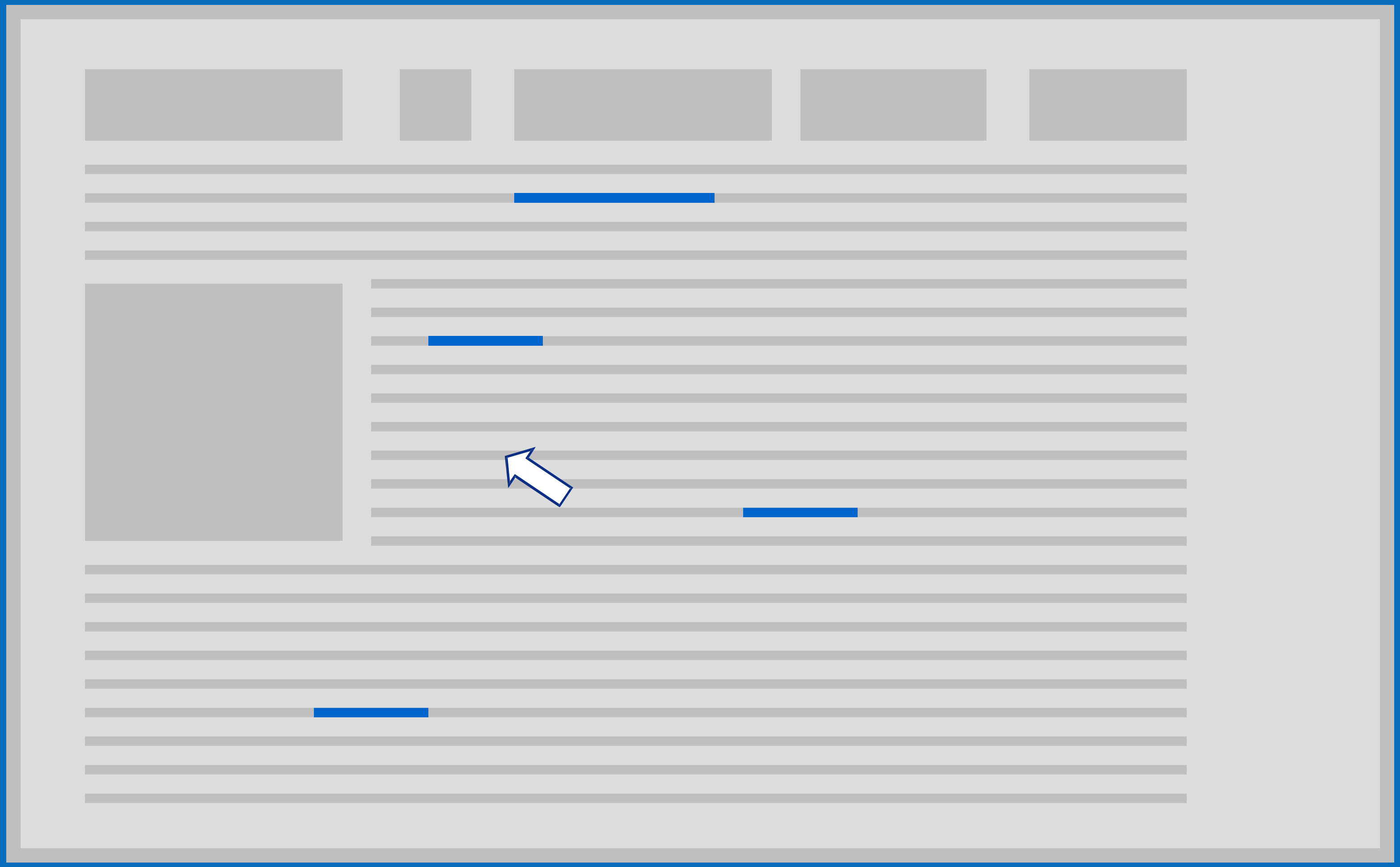
Corrected **main** Function

```
struct board *stack;
...
stack = NULL;
for (i = 0; i < n; i++) {
   initial=(struct board *)malloc(sizeof(struct board));
   initial->pieces = 1;
   initial->places[0] = i;
   initial->next = stack;
   stack = initial
}
num_solutions = 0;
omp_set_num_threads (omp_get_num_procs());
#pragma omp parallel
search_for_solutions (n, &stack, &num_solutions);
printf ("The %d-queens puzzle has %d solutions\n", n,
            num_solutions);
```

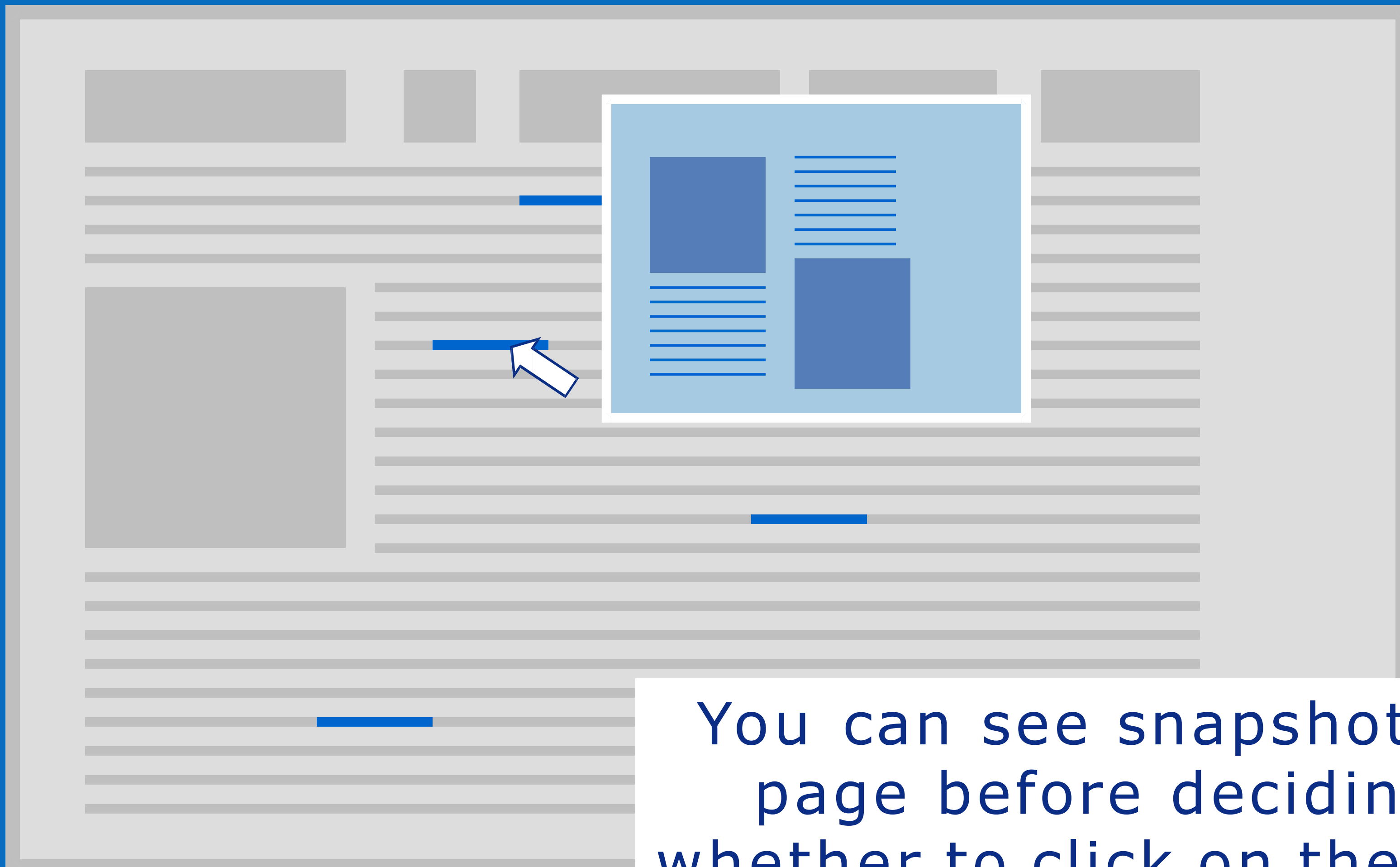
Corrected Stack Access Function

```
void search_for_solutions (int n,
   struct board **stack, int *num_solutions)
{
   struct board *ptr;
   void search (int, struct board *, int *);

   while (*stack != NULL) {
     #pragma omp critical
     { ptr = *stack;
       *stack = (*stack)->next; }
     search (n, ptr, num_solutions);
     free (ptr);
   }
}
```

Case Study: Fancy Web Browser



Case Study: Fancy Web Browser



You can see snapshot of page before deciding whether to click on the link

C Code

```
page = retrieve_page (url);
find_links (page, &num_links, &link_url);
for (i = 0; i < num_links; i++)
   snapshots[i].image = NULL;
for (i = 0; i < num_links; i++)
   generate_preview (&snapshots[i]);
display_page (page);
```

Pseudocode, Option A

Retrieve page

Identify links

Enter parallel region

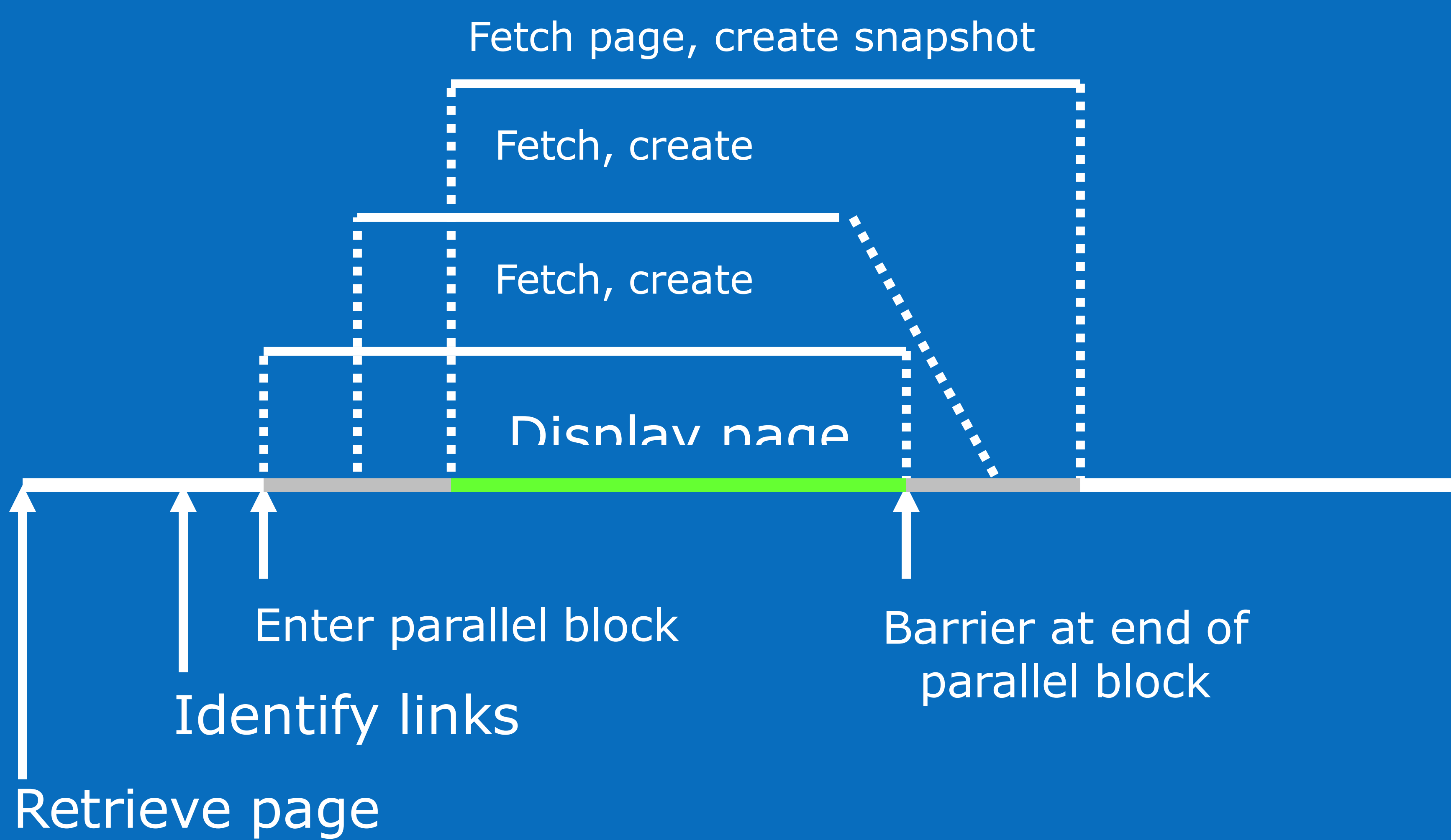
Thread gets ID number (*id*)

If *id* = 0 draw page

else fetch page & build snapshot image (*id*-1)

Exit parallel region

Timeline of Option A



C/OpenMP Code, Option A

```
page = retrieve_page (url);
find_links (page, &num_links, &link_url);
for (i = 0; i < num_links; i++)
   snapshots[i].image = NULL;
omp_set_num_threads (num_links + 1);
#pragma omp parallel private (id)
{
id = omp_get_thread_num();
if (id == 0) display_page (page);
else generate_preview (&snapshots[id-1]);
}
```

Pseudocode, Option B

Retrieve page

Identify links

Two activities happen in parallel

1. Draw page
2. For all links do in parallel

    Fetch page and build snapshot image

Parallel Sections

```
#pragma omp parallel sections
{
<code block A>
#pragma omp section
<code block B>
#pragma omp section
<code block C>
}
```

Meaning: The following block contains sub-blocks that may execute in parallel

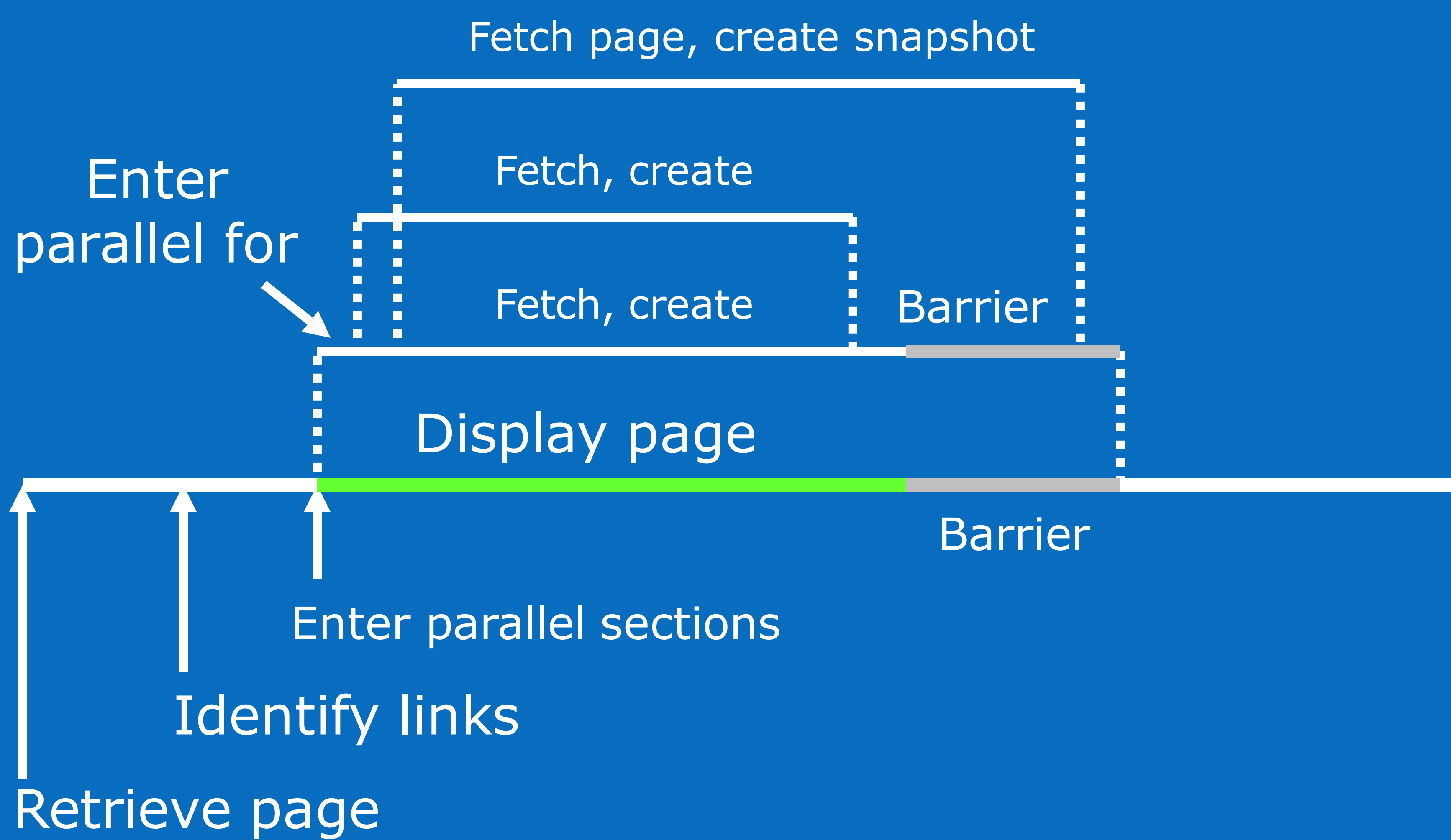
Each block executed by one thread

Dividers between sections

Nested Parallelism

- We can use `parallel sections` to specify two different concurrent activities: drawing the Web page and creating the snapshots
- We are using a `for` loop to create multiple snapshots; number of iterations is known only at run time
- We would like to make `for` loop parallel
- OpenMP allows nested parallelism: a parallel region inside another parallel region
- A thread entering a parallel region creates a new team of threads to execute it

Timeline of Option B



C/OpenMP Code, Option B

```
page = retrieve_page (url);
find_links (page, &num_links, &link_url);
omp_set_num_threads (2);
#pragma omp parallel sections
{
display_page (page);
#pragma omp section
omp_set_num_threads (num_links);
#pragma omp parallel for
for (i = 0; i < num_links; i++)
   generate_preview (&snapshots[i]);
}
```