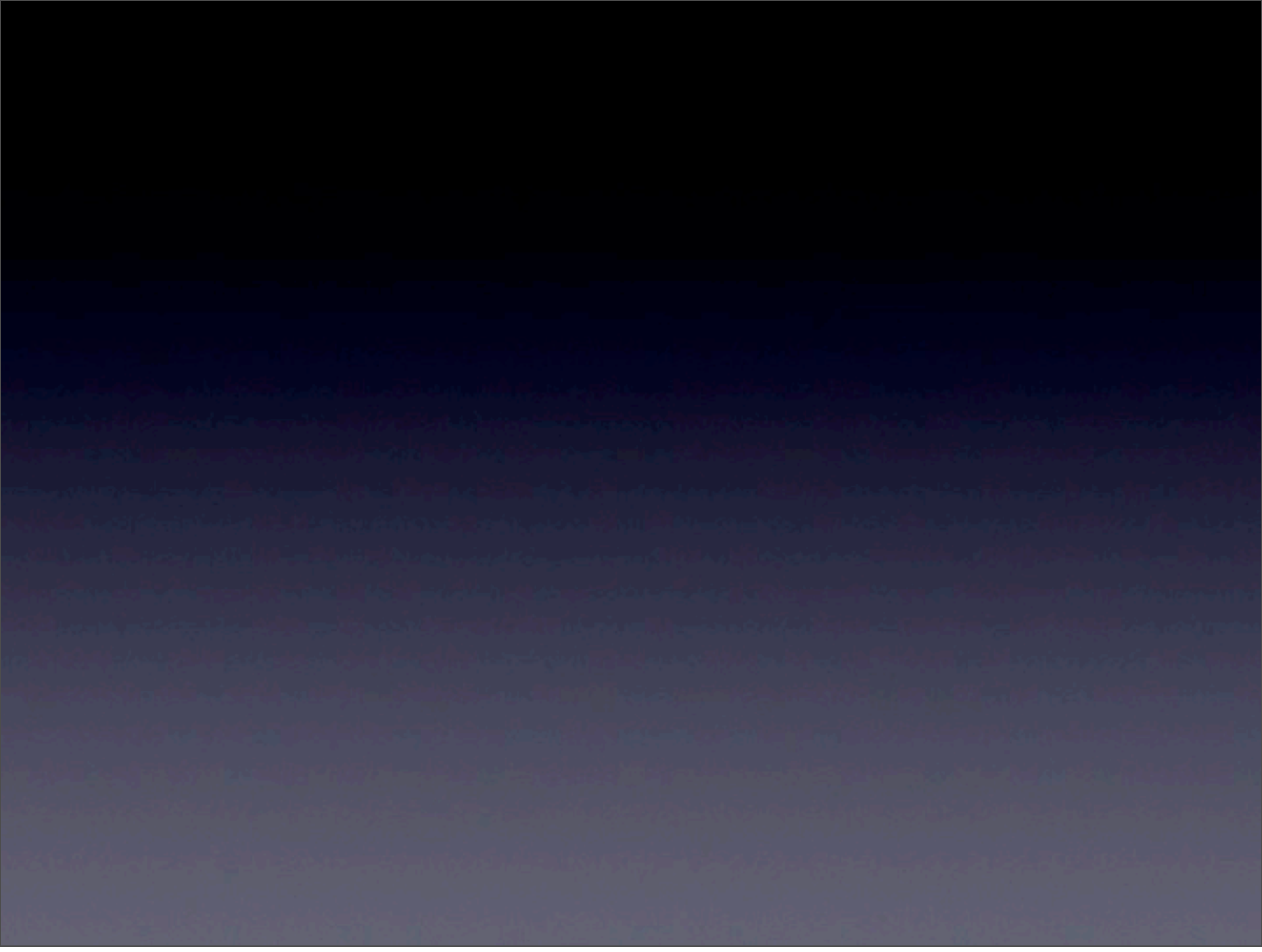
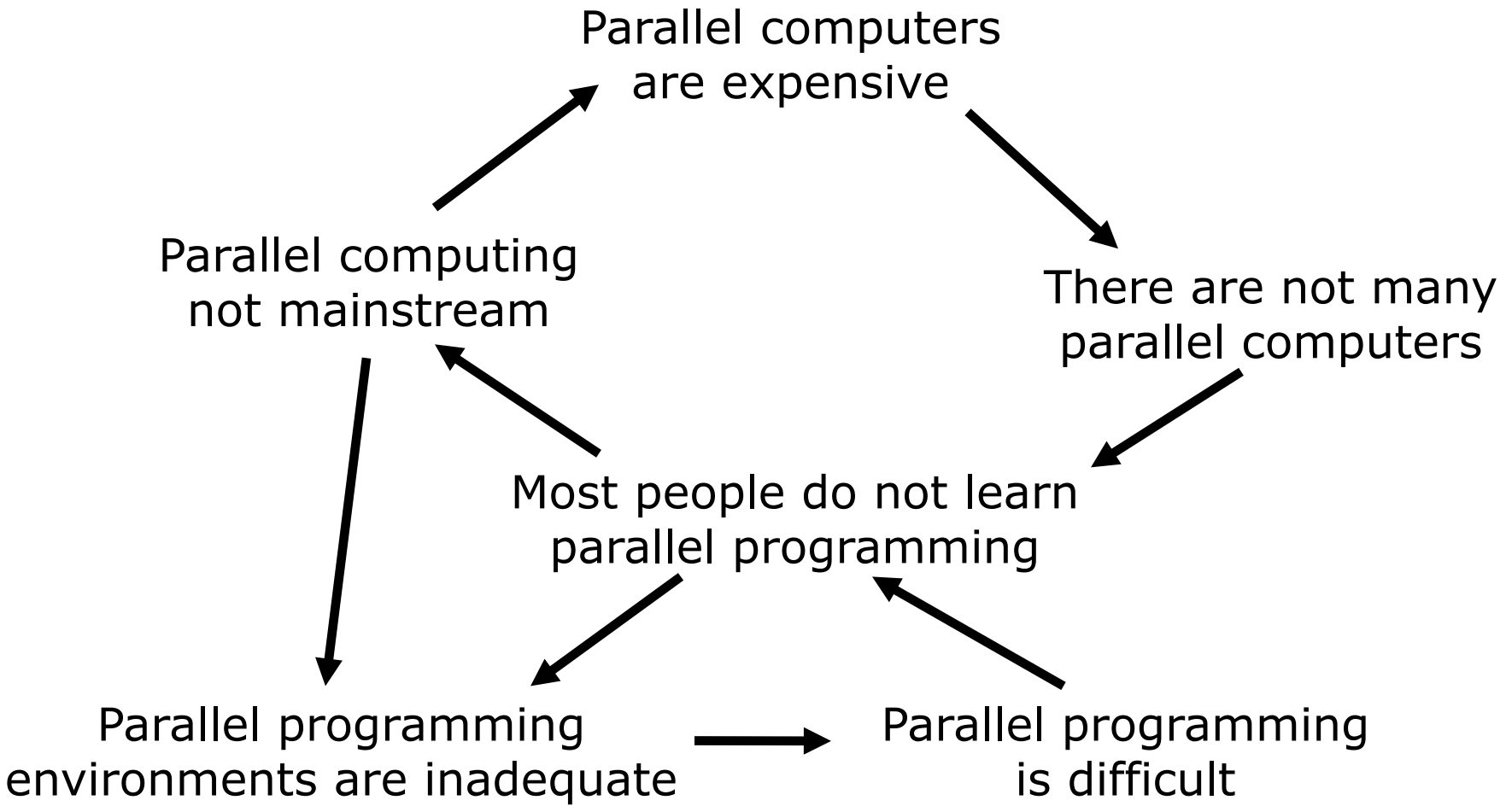


One more thing…
OpenMP has tasks

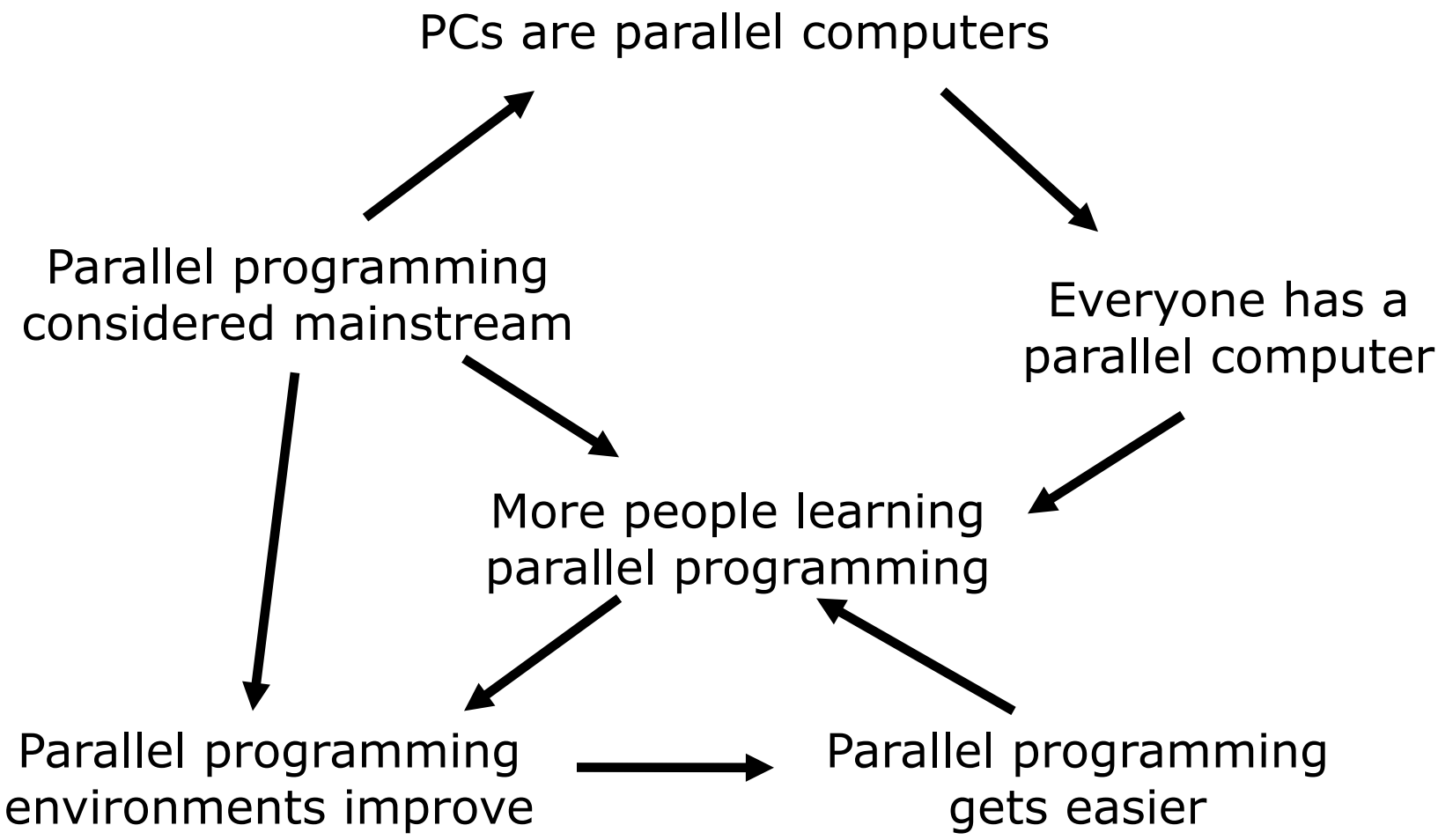
Multicore Programming and Architecture

Task-Based Parallelism

Old Dynamic of Parallel Computing



New Dynamic of Parallel Computing



Disclaimer

Disclaimer

- I'm a big fan of task-based parallelism
  - Seems like most attractive multicore programming paradigm
  - Captures loop-level parallelism of OpenMP + more irregular parallelism

Disclaimer

- I'm a big fan of task-based parallelism
  - Seems like most attractive multicore programming paradigm
  - Captures loop-level parallelism of OpenMP + more irregular parallelism
- Several variants of task-based parallelism
  - Cilk (language extension)
  - Intel's Threaded Building Blocks (C++ library)
  - Java's JSR-166y (for potential inclusion in Java 7)
  - Microsoft's Parallel Patterns Library (PPL)

Disclaimer

- I'm a big fan of task-based parallelism
  - Seems like most attractive multicore programming paradigm
  - Captures loop-level parallelism of OpenMP + more irregular parallelism
- Several variants of task-based parallelism
  - Cilk (language extension)
  - Intel's Threaded Building Blocks (C++ library)
  - Java's JSR-166y (for potential inclusion in Java 7)
  - Microsoft's Parallel Patterns Library (PPL)
- All variants have some basic idea…
  - But many differences in interface, implementation, etc.

Acknowledgments

- Includes slides from:
  - "Shared Memory Control Programming Intel Threading Building Blocks"
    - Clay Breshears (Intel)
    - Presented at UIUC's UPCRC 2009 Summer School
    - With permission, **includes some modifications by me**
  - Various "Intel Software College" slides **Blue background slides**
    - "Threading for Performance with Intel Threading Building Blocks"
    - "Recognizing Potential Parallelism"
    - "Implementing Task Decompositions"
- Other sources and references:
  - "Intel Threading Building Blocks" by James Reinders, O'Reilly
  - TBB documentation

Irregular Parallelism

Irregular Parallelism

- Beyond OpenMP's "Loop-level" parallelism
  - Primary focus on parallel loops
  - With **known** iteration counts

Irregular Parallelism

- Beyond OpenMP's "Loop-level" parallelism
  - Primary focus on parallel loops
  - With **known** iteration counts
- What if the parallelism is more irregular?
  - Walking a tree or graph

Irregular Parallelism

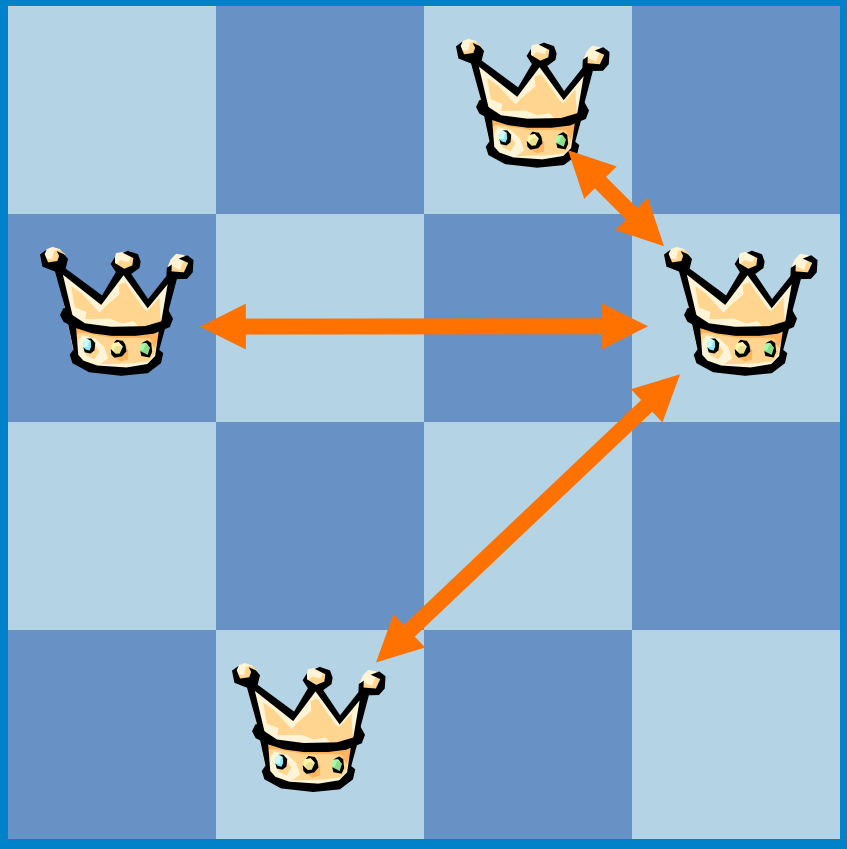
- Beyond OpenMP's "Loop-level" parallelism
  - Primary focus on parallel loops
  - With **known** iteration counts
- What if the parallelism is more irregular?
  - Walking a tree or graph
- Can often express this irregular parallelism as "tasks"
  - Tasks are bundles of (mostly) independent computation

Irregular Parallelism

- Beyond OpenMP's "Loop-level" parallelism
  - Primary focus on parallel loops
  - With **known** iteration counts
- What if the parallelism is more irregular?
  - Walking a tree or graph
- Can often express this irregular parallelism as "tasks"
  - Tasks are bundles of (mostly) independent computation
- Generalization of work-list algorithm
  - Task may or may not have dependencies

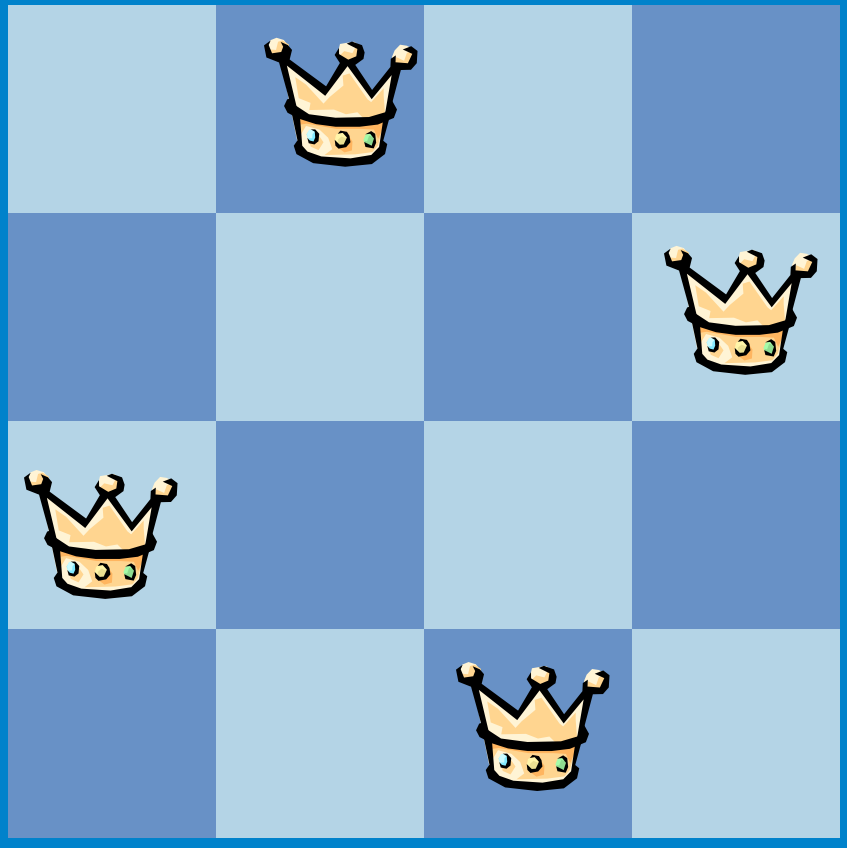
Case Study: N Queens

Case Study: The N Queens Problem

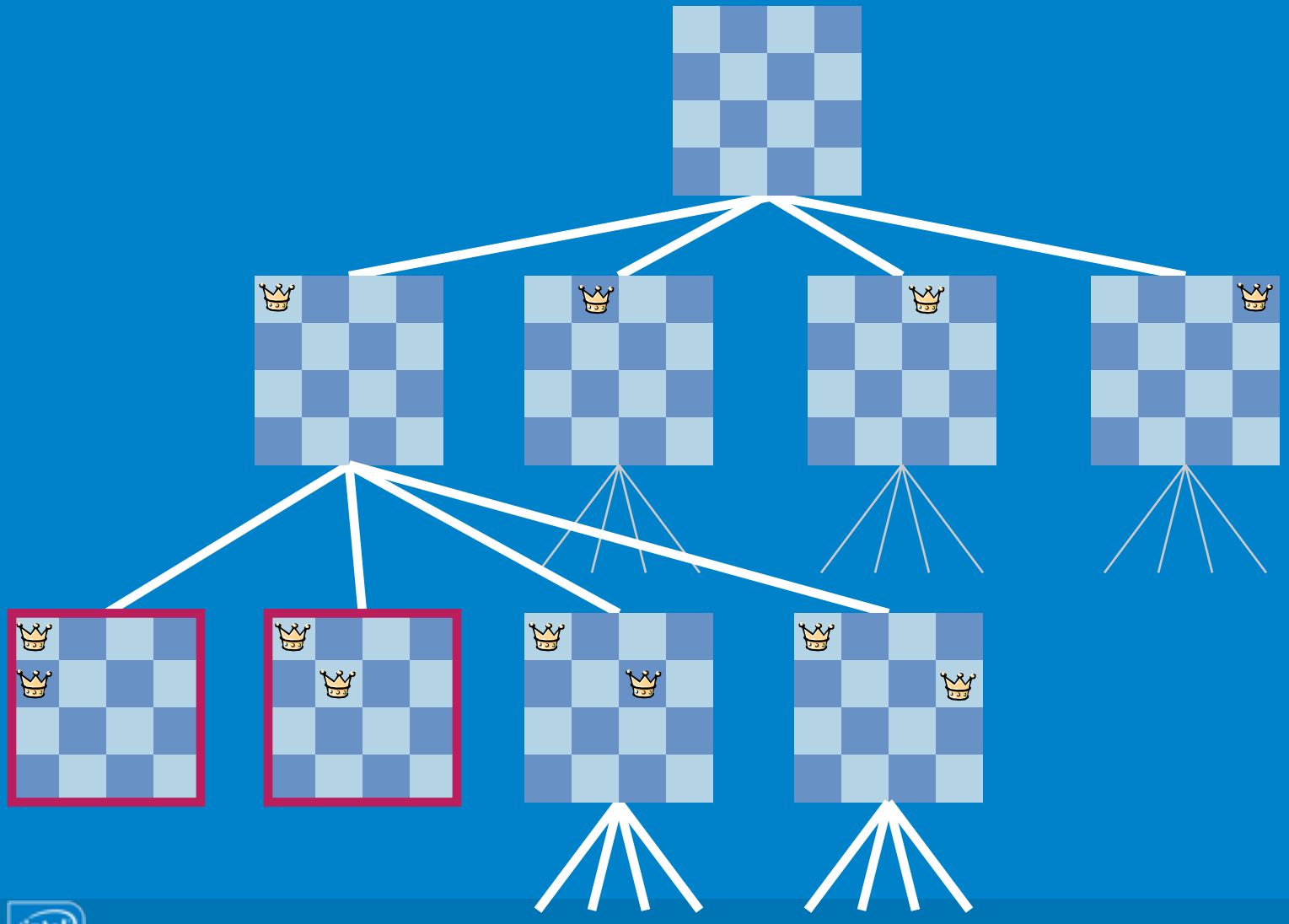


Is there a way to place N queens on an N-by-N chessboard such that no queen threatens another queen?

A Solution to the 4 Queens Problem



Exhaustive Search



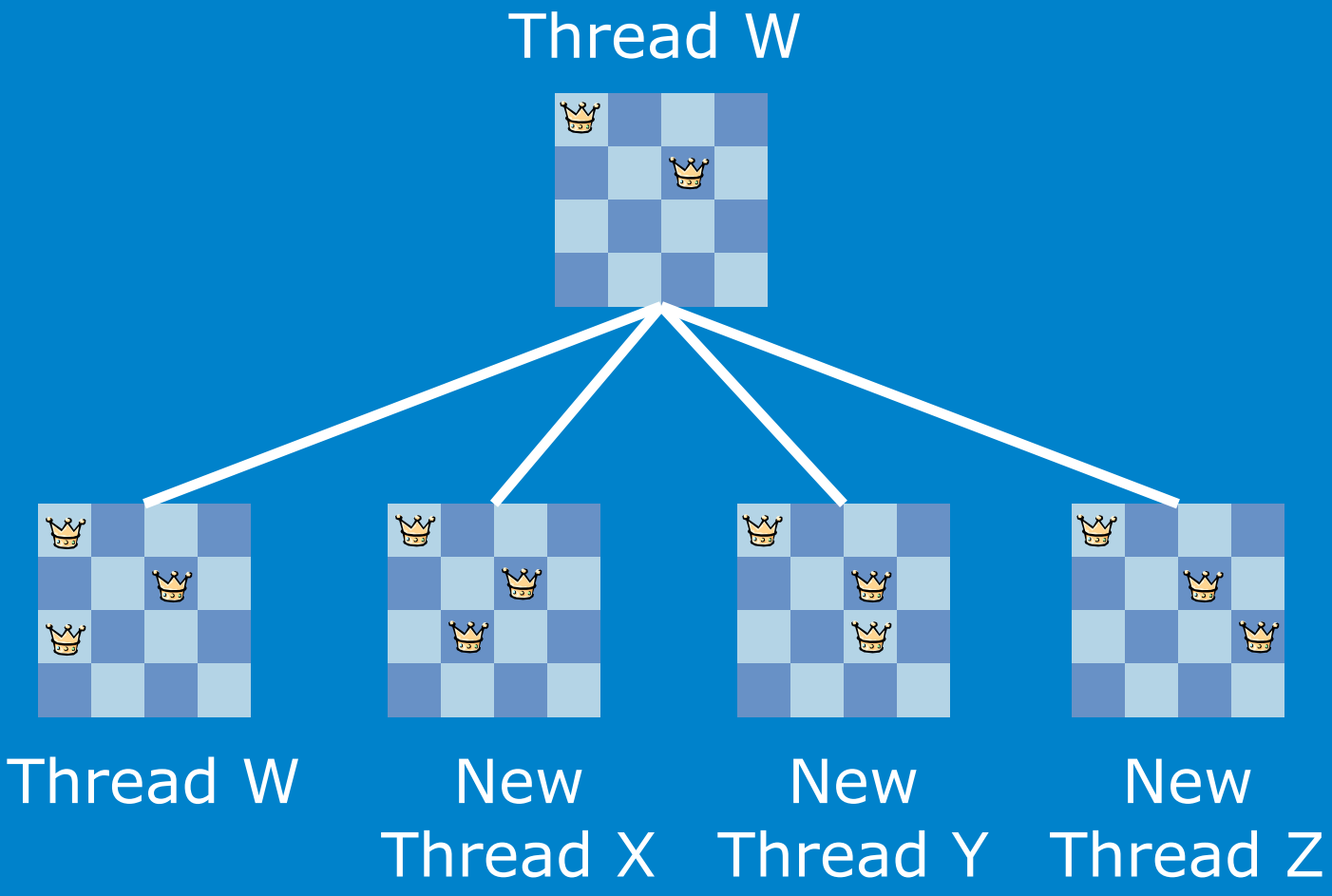
Design #1 for Parallel Search

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

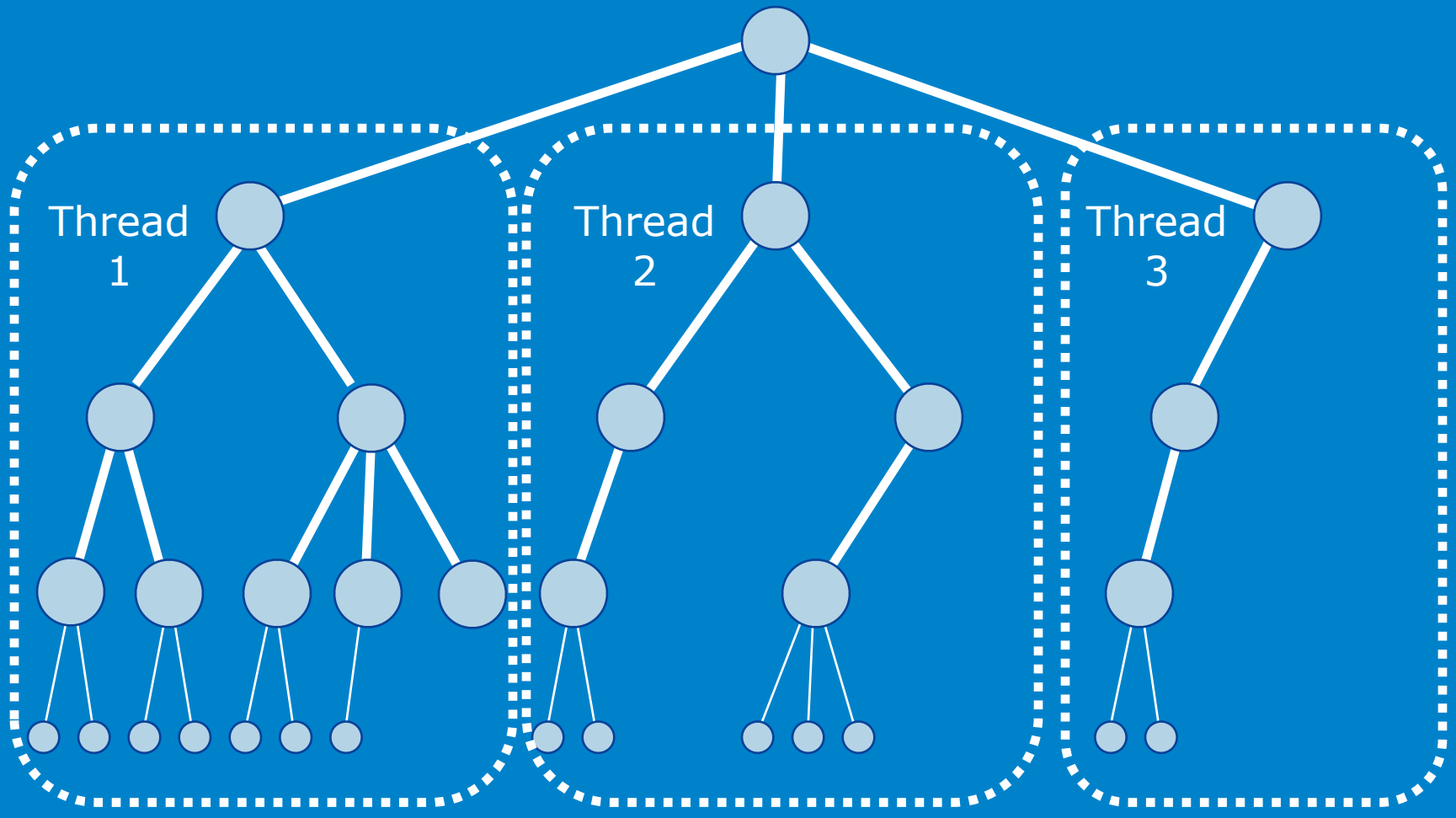
- Too many threads created
- Lifetime of threads too short
- Overhead costs too high

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

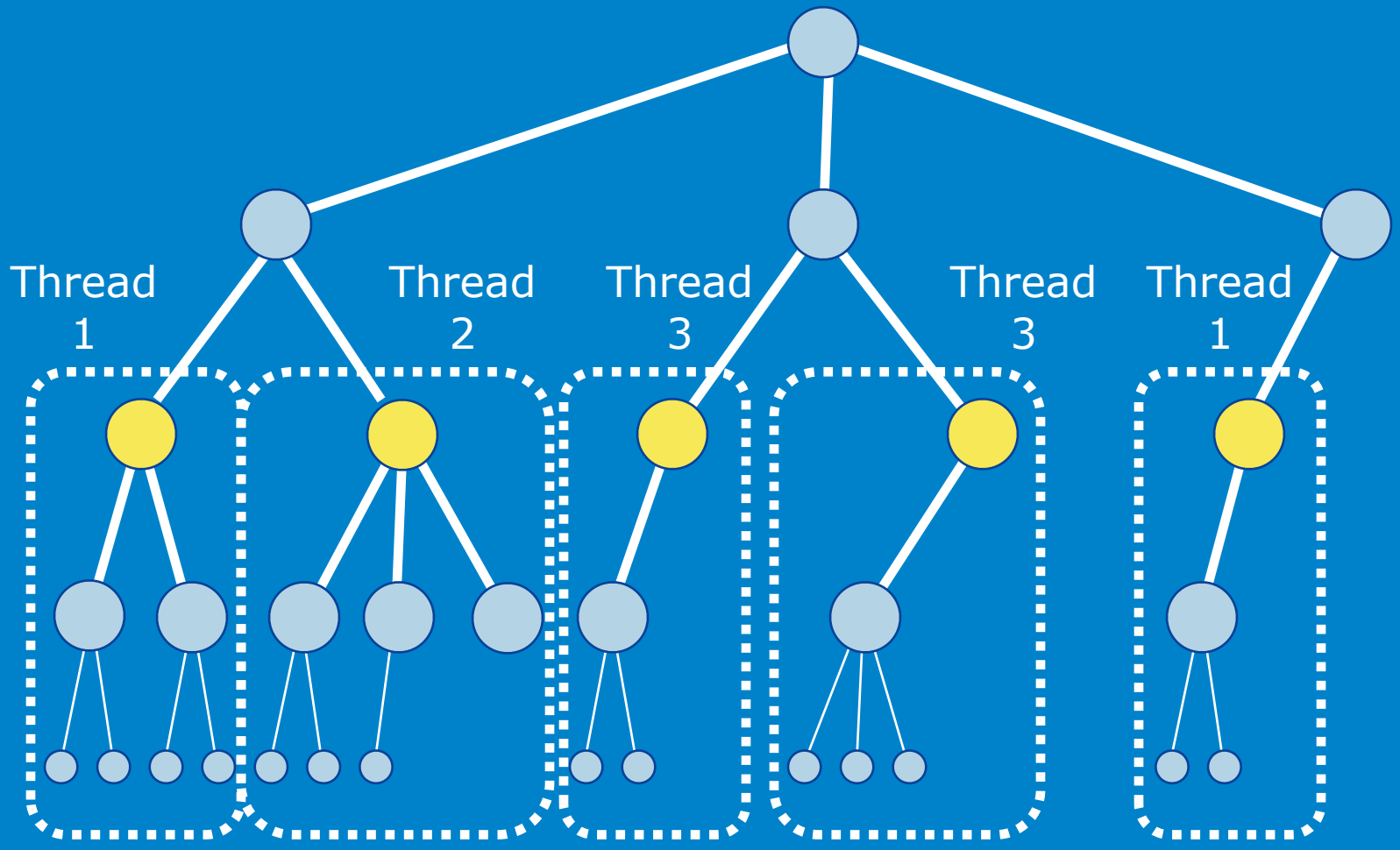
Main thread creates work pool—list of subtrees to explore

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

Design #3 in Action



Pros and Cons of Strategy #3

Pros

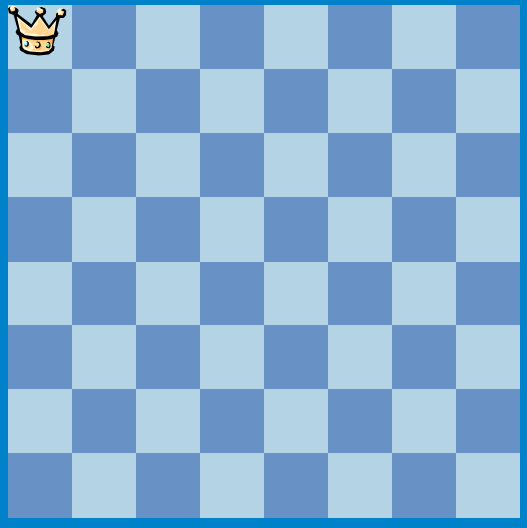
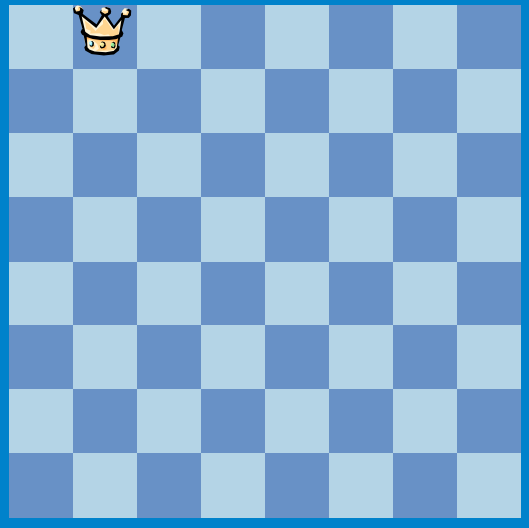
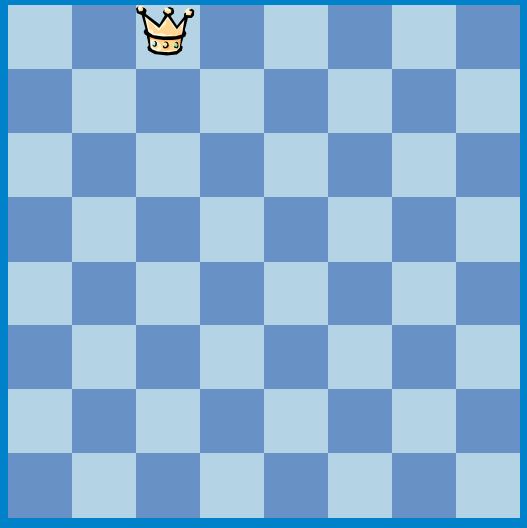
- Thread creation/termination time minimized
- Good workload balance

Cons

- Threads need exclusive access to data structure containing work to be done

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

N Queens Recap

N Queens Recap

- Unlike dynamic loop scheduling in OpenMP
  - Amount of work not known at the start
  - "context" of the task is more than just an iteration count

N Queens Recap

- Unlike dynamic loop scheduling in OpenMP
  - Amount of work not known at the start
  - "context" of the task is more than just an iteration count
- Encapsulate data and computation into a task
  - Number of queens placed thus far (how many rows processed)
  - State of the board
  - "execute" method

N Queens Recap

- Unlike dynamic loop scheduling in OpenMP
  - Amount of work not known at the start
  - "context" of the task is more than just an iteration count
- Encapsulate data and computation into a task
  - Number of queens placed thus far (how many rows processed)
  - State of the board
  - "execute" method
- Execute method of task
  - if (number of queens placed thus far > threshold):
    - Complete search sequentially
    - Atomic increment of global counter of found solutions
  - Else:
    - Enqueue a sub-task **for each** valid placement of queen on next row

N Queens Recap

- Unlike dynamic loop scheduling in OpenMP
  - Amount of work not known at the start
  - "context" of the task is more than just an iteration count
- Encapsulate data and computation into a task
  - Number of queens placed thus far (how many rows processed)
  - State of the board
  - "execute" method
- Execute method of task
  - if (number of queens placed thus far > threshold):
    - Complete search sequentially
    - Atomic increment of global counter of found solutions
  - Else:
    - Enqueue a sub-task **for each** valid placement of queen on next row
- Work-list scheduling of computation, no explicit dependencies

Task Decomposition & Dependencies

Task Decomposition and Dependencies

Task Decomposition and Dependencies

- In more general task-based models...
  - Tasks can have dependencies (implicit or explicit)

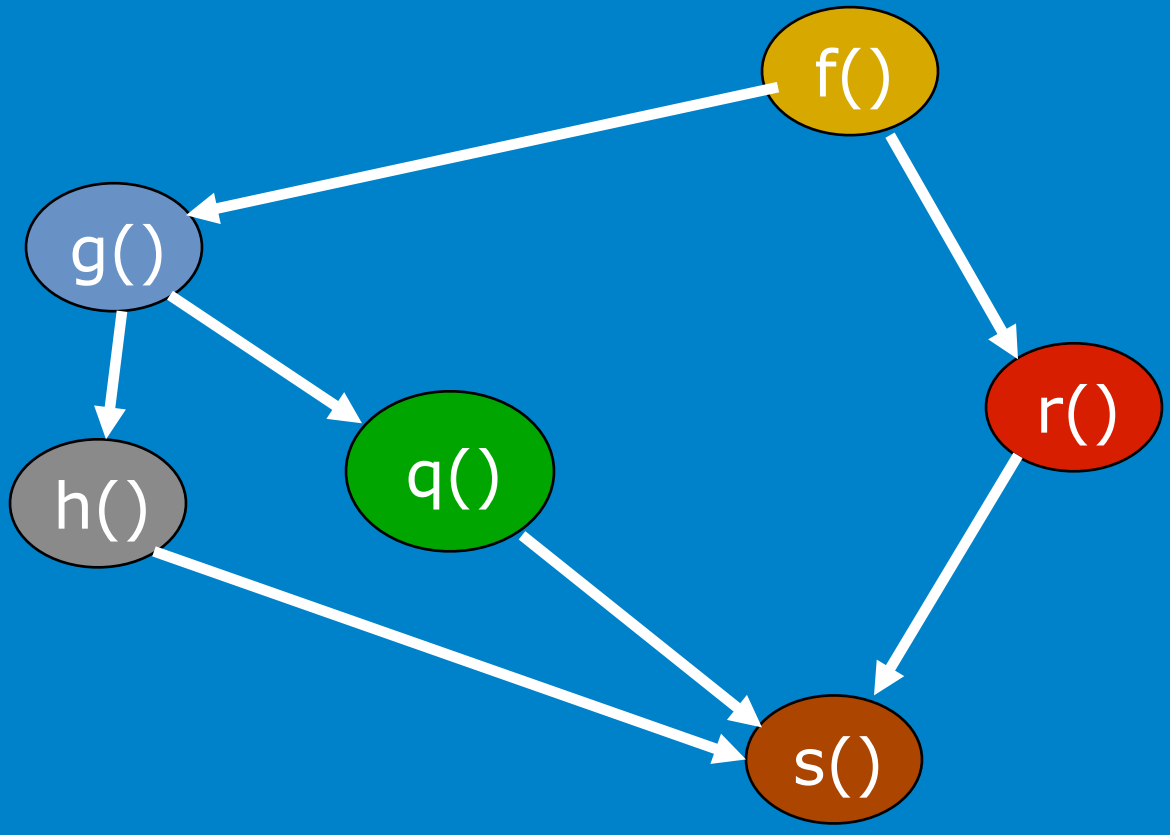
Task Decomposition and Dependencies

- In more general task-based models...
  - Tasks can have dependencies (implicit or explicit)
- Task Decomposition
  - Identify "tasks"
    - Bundles of mostly independent work
  - Identify dependencies between tasks
    - Creates a direct acyclic graph (DAG) of computation

Task Decomposition and Dependencies

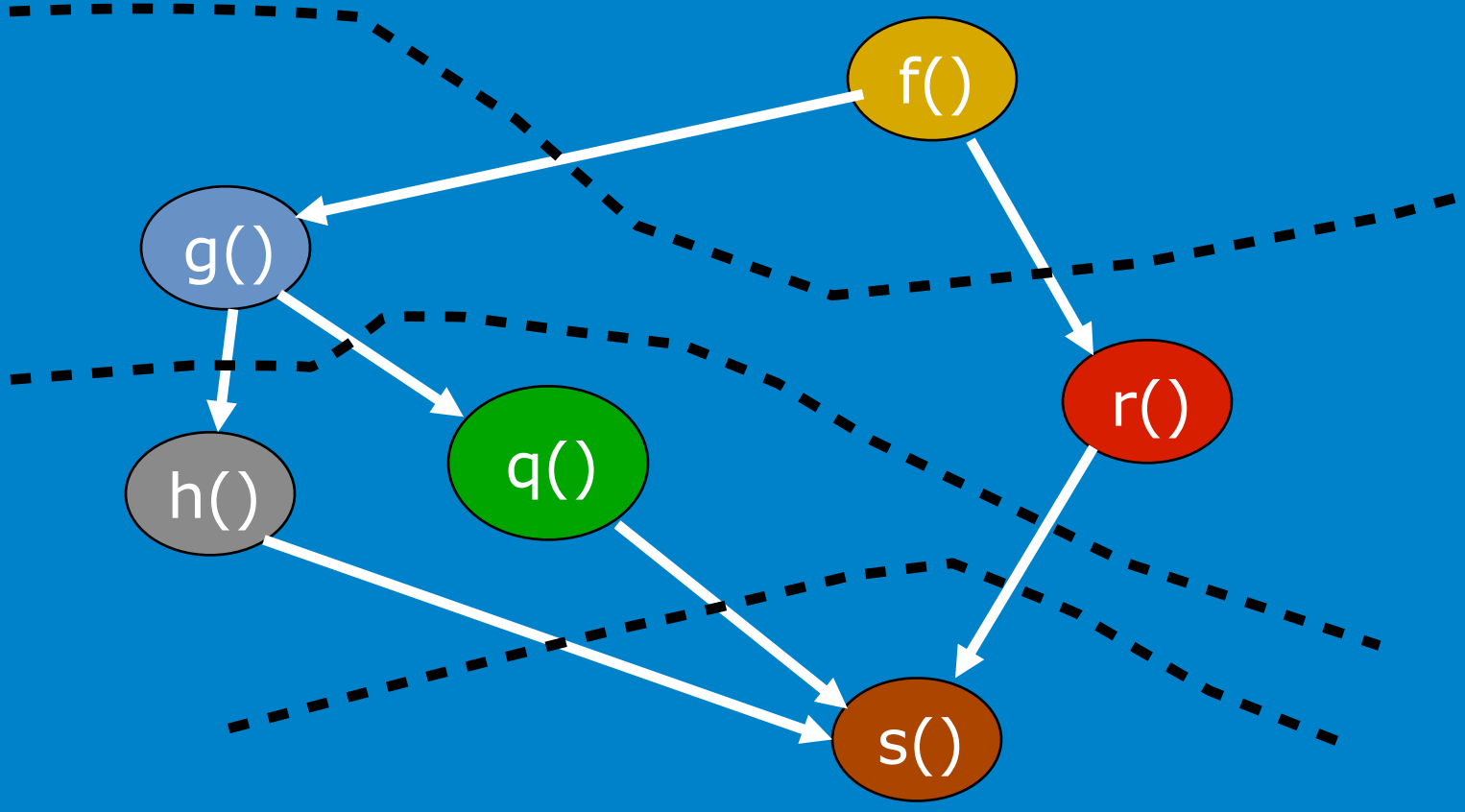
- In more general task-based models...
  - Tasks can have dependencies (implicit or explicit)
- Task Decomposition
  - Identify "tasks"
    - Bundles of mostly independent work
  - Identify dependencies between tasks
    - Creates a direct acyclic graph (DAG) of computation
- Task scheduling
  - Static: if all tasks know at start
  - Dynamic: worker threads executes tasks from a task pool
    - Allows tasks to create sub-tasks

Task Dependencies



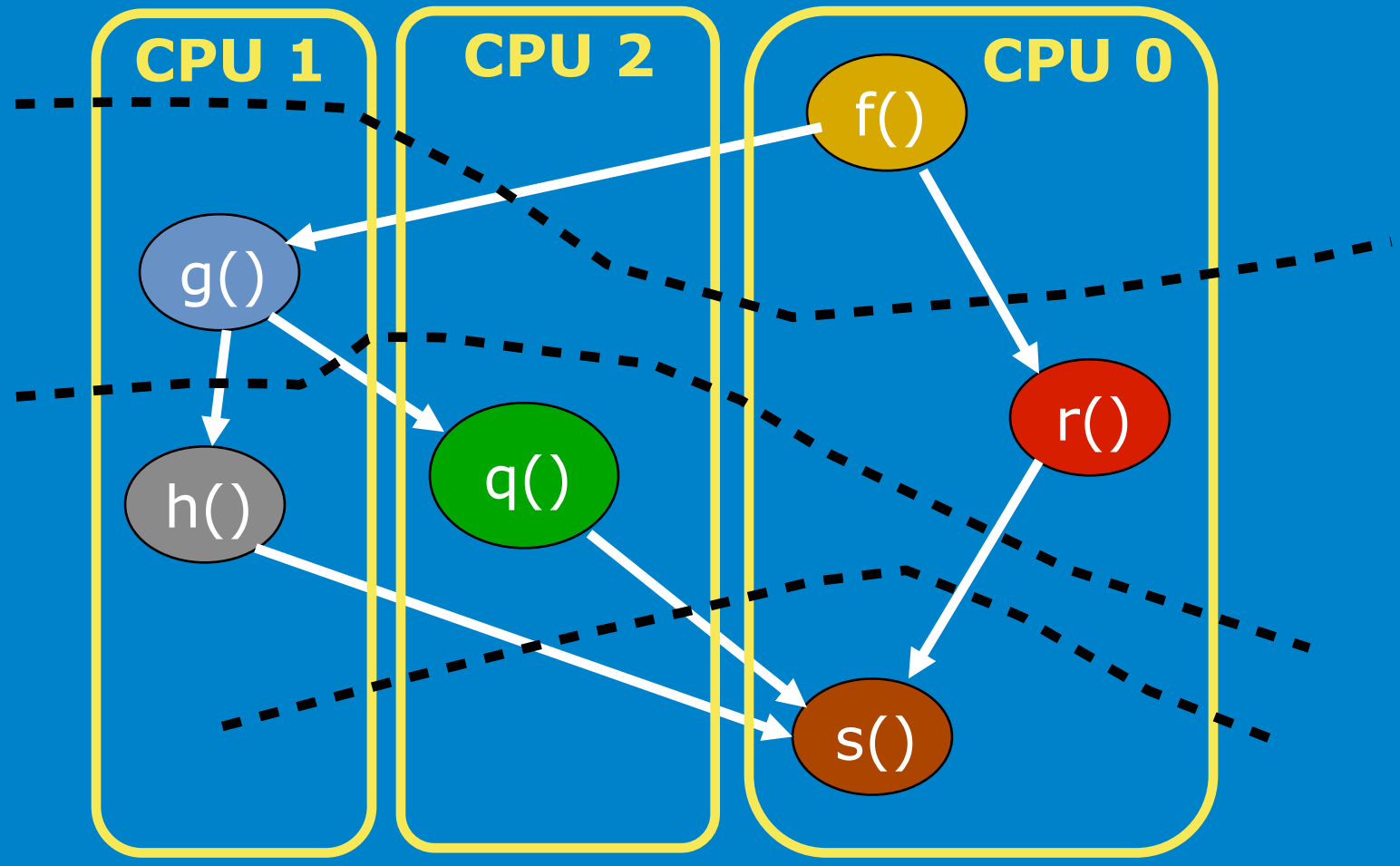
Total work: 6

Task Dependencies



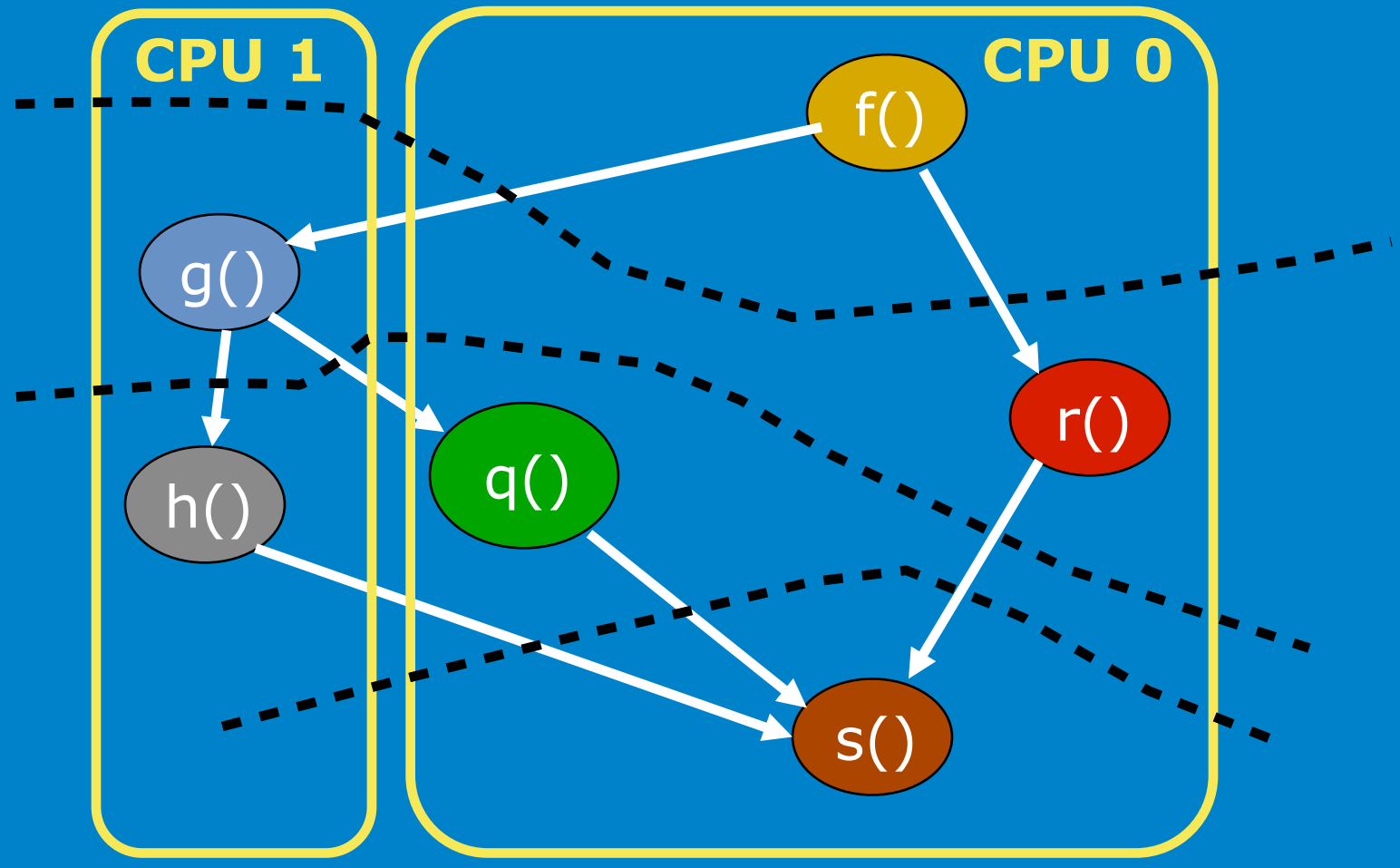
Total work: 6 Critical path: 4

Task Dependencies



Total work: 6 Critical path: 4 Max speedup: 1.5

Task Dependencies



Total work: 6 Critical path: 4 Max speedup: 1.5

Recursive Task-Based Parallelism

Recursive Task-Based Parallelism

Recursive Task-Based Parallelism

- Dynamic
  - Tasks can create more tasks

Recursive Task-Based Parallelism

- Dynamic
  - Tasks can create more tasks
- Recursive fork/join
  - Parent task "spawns" sub-tasks
  - "Join" operation waits until all sub-tasks complete
    - But not all tasks
  - Impact: need some way to track which sub-tasks have completed
    - Reference counting of live (non-completed) sub-tasks

Recursive Task-Based Parallelism

- Dynamic
  - Tasks can create more tasks
- Recursive fork/join
  - Parent task "spawns" sub-tasks
  - "Join" operation waits until all sub-tasks complete
    - But not all tasks
  - Impact: need some way to track which sub-tasks have completed
    - Reference counting of live (non-completed) sub-tasks
- In essence, creates a implicit task dependency

Recursive Task-Based Parallelism

- Dynamic
  - Tasks can create more tasks
- Recursive fork/join
  - Parent task "spawns" sub-tasks
  - "Join" operation waits until all sub-tasks complete
    - But not all tasks
  - Impact: need some way to track which sub-tasks have completed
    - Reference counting of live (non-completed) sub-tasks
- In essence, creates a implicit task dependency
- On "Join":
  - Option #1: continue with code at join point ("spawn and wait")
  - Option #2: call an explicit continuation (no "wait")

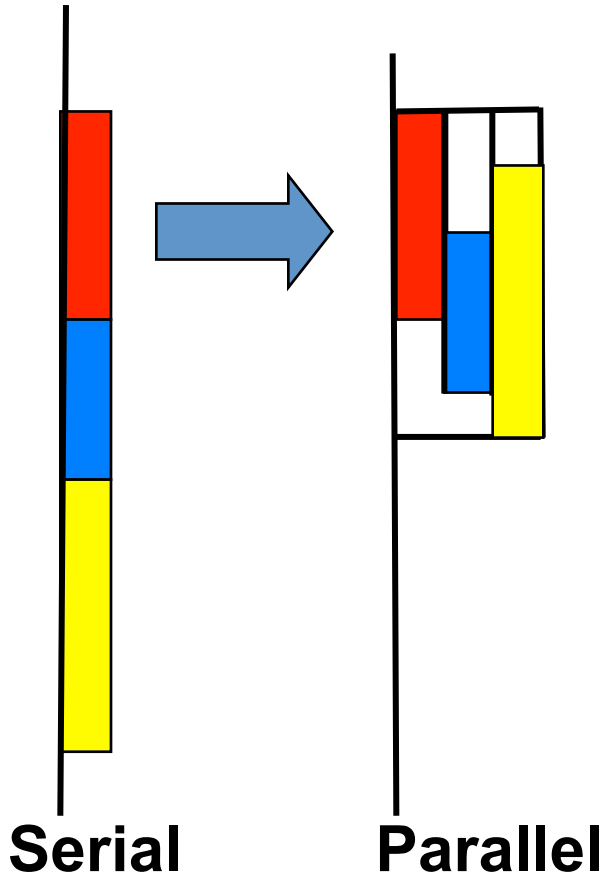
Tasks in OpenMP 3.0

New Addition to OpenMP

- Tasks – Main change for OpenMP 3.0
- 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
  - Tasks are composed of:
    - **code** to execute
    - **data** environment
    - **internal control variables** (ICV)



Task Construct – Explicit Task View

- A team of threads is created at the **`omp parallel`** construct
- A single thread, T0, is chosen to execute the **`while`** loop
- T0 operates the **`while`** loop, creates tasks, and fetches next pointers
- Each time T0 crosses the **`omp task`** construct it generates a new task
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region's single construct

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

Why are tasks useful?

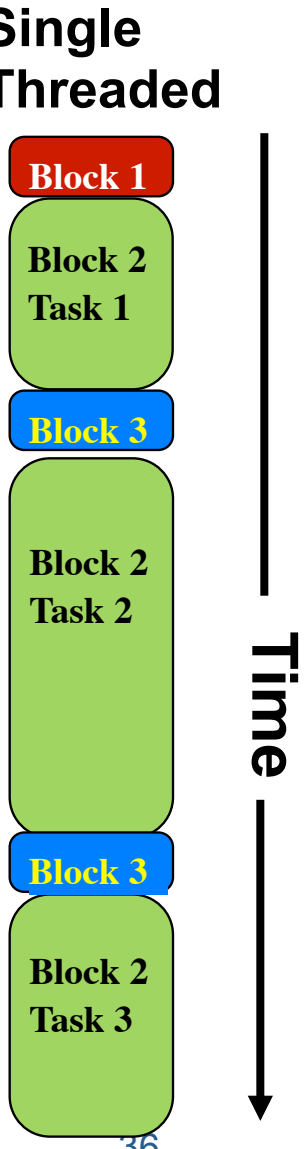
Have potential to parallelize irregular patterns and recursive function calls

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

Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

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

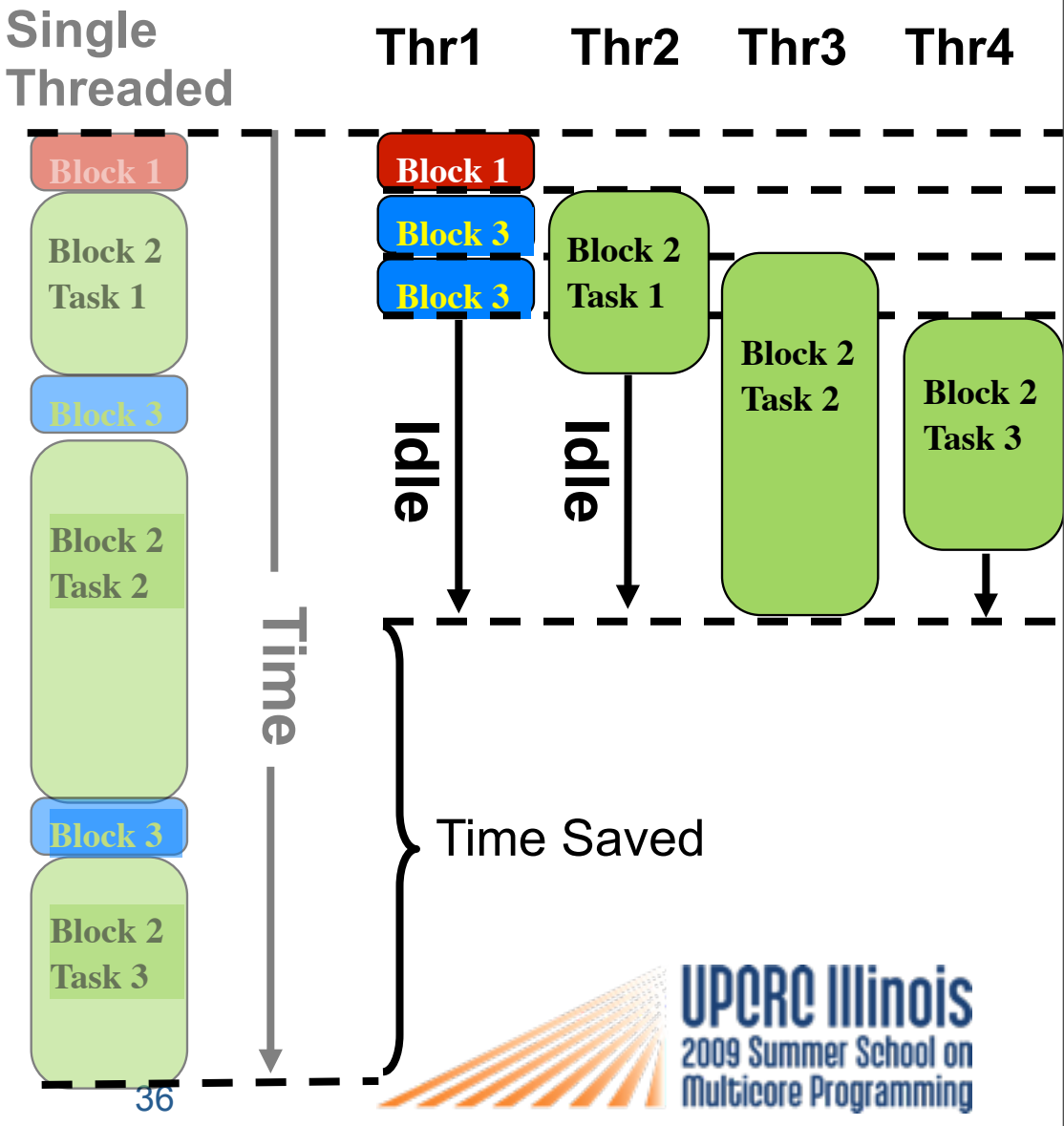


UPCRC Illinois
2009 Summer School on Multicore Programming

Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

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



When are tasks guaranteed to be complete?

Tasks are guaranteed to be complete:

- At thread or task barriers
- At the directive: **`#pragma omp barrier`**
- At the directive: **`#pragma omp taskwait`**

OpenMP 3.0

General task characteristics

- A task has
  - Code to execute
  - A data environment (it *owns* its data)
  - An assigned thread that executes the code and uses the data
- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task (code and data)
  - Some thread in the team executes the task at some later time

OpenMP 3.0

Definitions

- *Task construct* – `task` directive plus structured block
- *Task* – the package of code and instructions for allocating data created when a thread encounters a task construct
- *Task region* – the dynamic sequence of instructions produced by the execution of a task by a thread

Tasks and OpenMP

- Tasks have been fully integrated into OpenMP
- Key concept: OpenMP has always had tasks, we just never called them that.
  - Thread encountering `parallel` construct packages up a set of *implicit* tasks, one per thread.
  - Team of threads is created.
  - Each thread in team is assigned to one of the tasks (and *tied* to it).
  - Barrier holds original master thread until all implicit tasks are finished.
- We have simply added a way to create a task explicitly for the team to execute.
- Every part of an OpenMP program is part of one task or another!

OpenMP 3.0

`task` Construct

`#pragma omp task [`*`clause`*`[[,]`*`clause`*`] ...]`
structured-block

where *clause* can be one of:

`if (`*`expression`***`)`**
`untied`
`shared (`*`list`***`)`**
`private (`*`list`***`)`**
`firstprivate (`*`list`***`)`**
`default( shared | none )`

The `if` clause

- When the `if` clause argument is false
  - The task is executed immediately by the encountering thread.
  - The data environment is still local to the new task...
  - ...and it's still a different task with respect to synchronization.
- It's a user directed optimization
  - when the cost of deferring the task is too great compared to the cost of executing the task code
  - to control cache and memory affinity

When/where are tasks complete?

- At thread barriers, explicit or implicit
  - applies to all tasks generated in the current parallel region up to the barrier
  - matches user expectation
- At task barriers
  - i.e. Wait until all tasks defined in the current task have completed.

```
#pragma omp taskwait
```

  - Note: applies only to tasks generated in the current task, not to “descendants” .

Example – parallel pointer chasing using tasks

```
#pragma omp parallel
{
  #pragma omp single private(p)
   {
    p = listhead ;
    while (p) {
       #pragma omp task
                process (p)
       p=next (p) ;
     }
   }
}
```

`p` is firstprivate inside this task

Example – parallel pointer chasing on multiple lists using tasks

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

Example: postorder tree traversal

```
void postorder(node *p) {
    if (p->left)
      #pragma omp task
        postorder(p->left);
    if (p->right)
      #pragma omp task
        postorder(p->right);
#pragma omp taskwait // wait for descendants
  process(p->data);
}
```

Task scheduling point

- Parent task suspended until children tasks complete

OpenMP 3.0

Task switching

- Certain constructs have task scheduling points at defined locations within them
- When a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (called *task switching*)
- It can then return to the original task and resume

Task switching example

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Too many tasks generated in an eye-blink
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
  - execute an already generated task (draining the "*task pool*")
  - dive into the encountered task (could be very cache-friendly)

Thread switching

```
#pragma omp single
{
  #pragma omp task  untied
    for (i=0; i<ONEZILLION; i++)
      #pragma omp task
        process(item[i]);
}
```

- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving…

- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- Too strange to be the default: the programmer is responsible!

OpenMP 3.0

Dealing with taskprivate data

- The Taskprivate directive was removed from OpenMP 3.0
  - Too expensive to implement
- Restrictions on task scheduling allow threadprivate data to be used
  - User can avoid thread switching with tied tasks
  - Task scheduling points are well defined

Data Sharing: tasks (OpenMP 3.0)

- The default for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope).
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared, because the barrier guarantees task completion.

```
#pragma omp parallel shared(A) private(B)
{
   ...
#pragma omp task
   {
      int C;
      compute(A, B, C);
   }
}
```

A is shared
B is firstprivate
C is private

Parallel Sort Example (with work stealing)

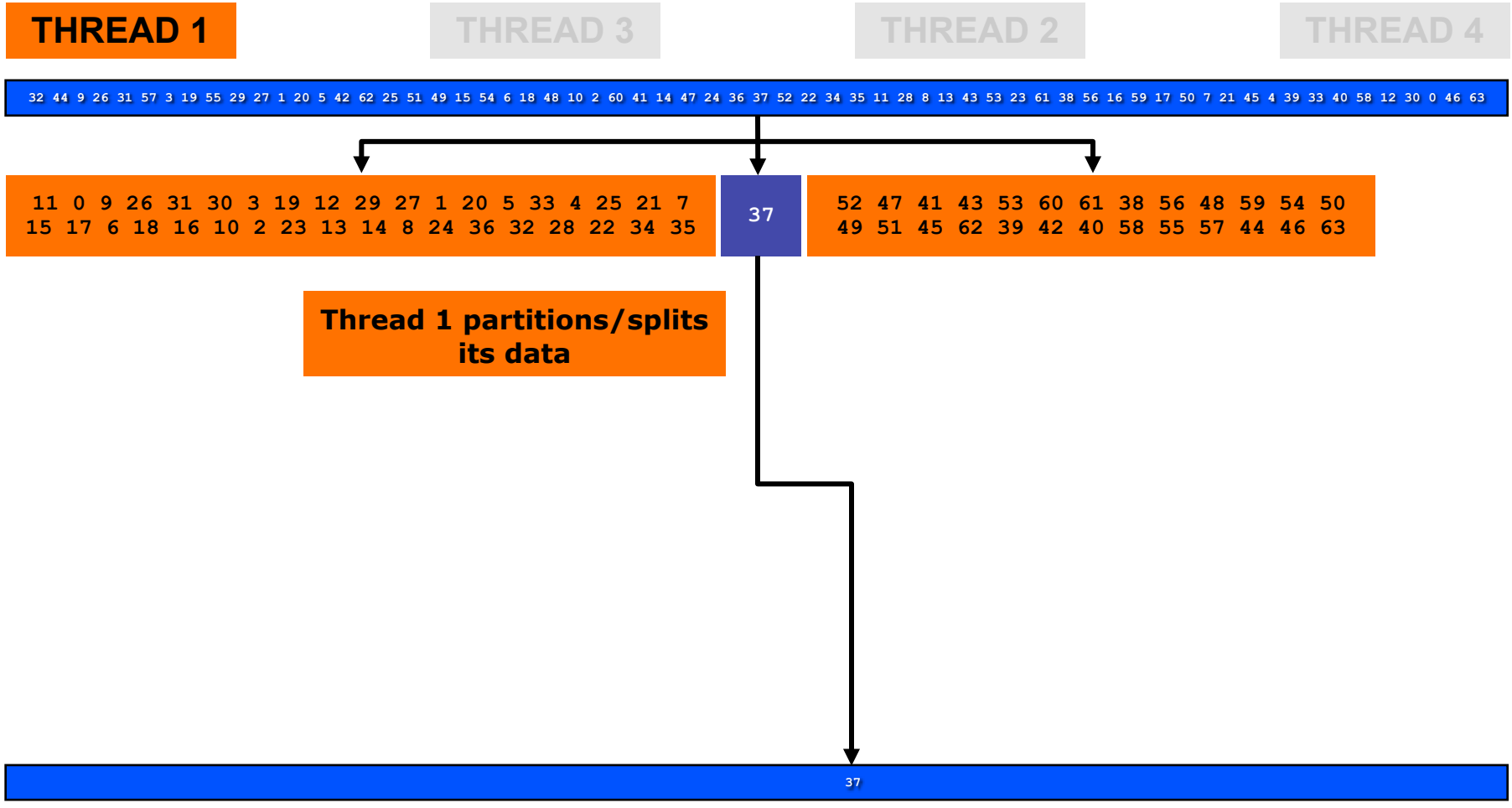
Quicksort – Step 1

```
tbb::parallel_sort (color, color+64);
```

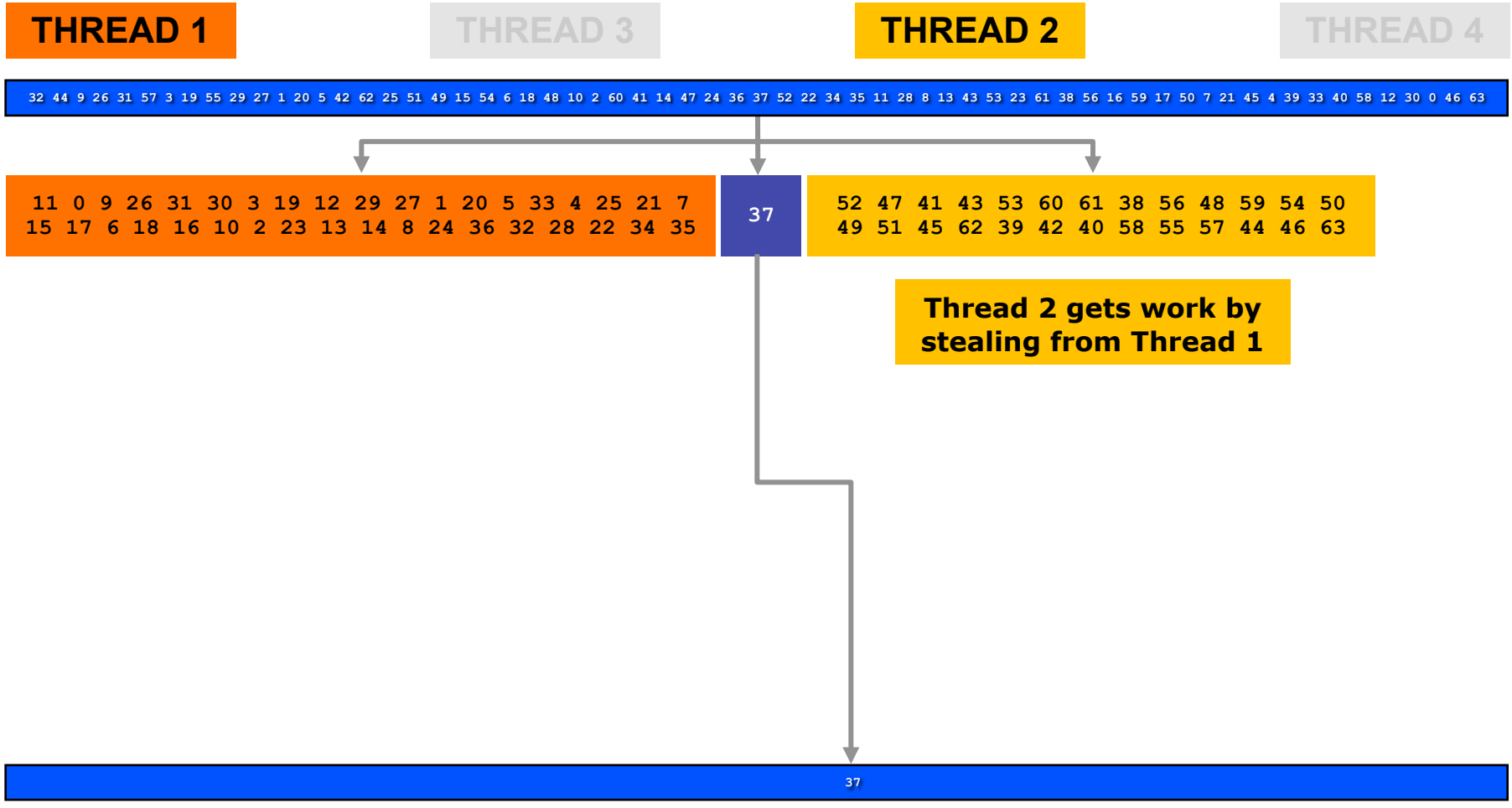
THREAD 1

Thread 1 starts with the initial data

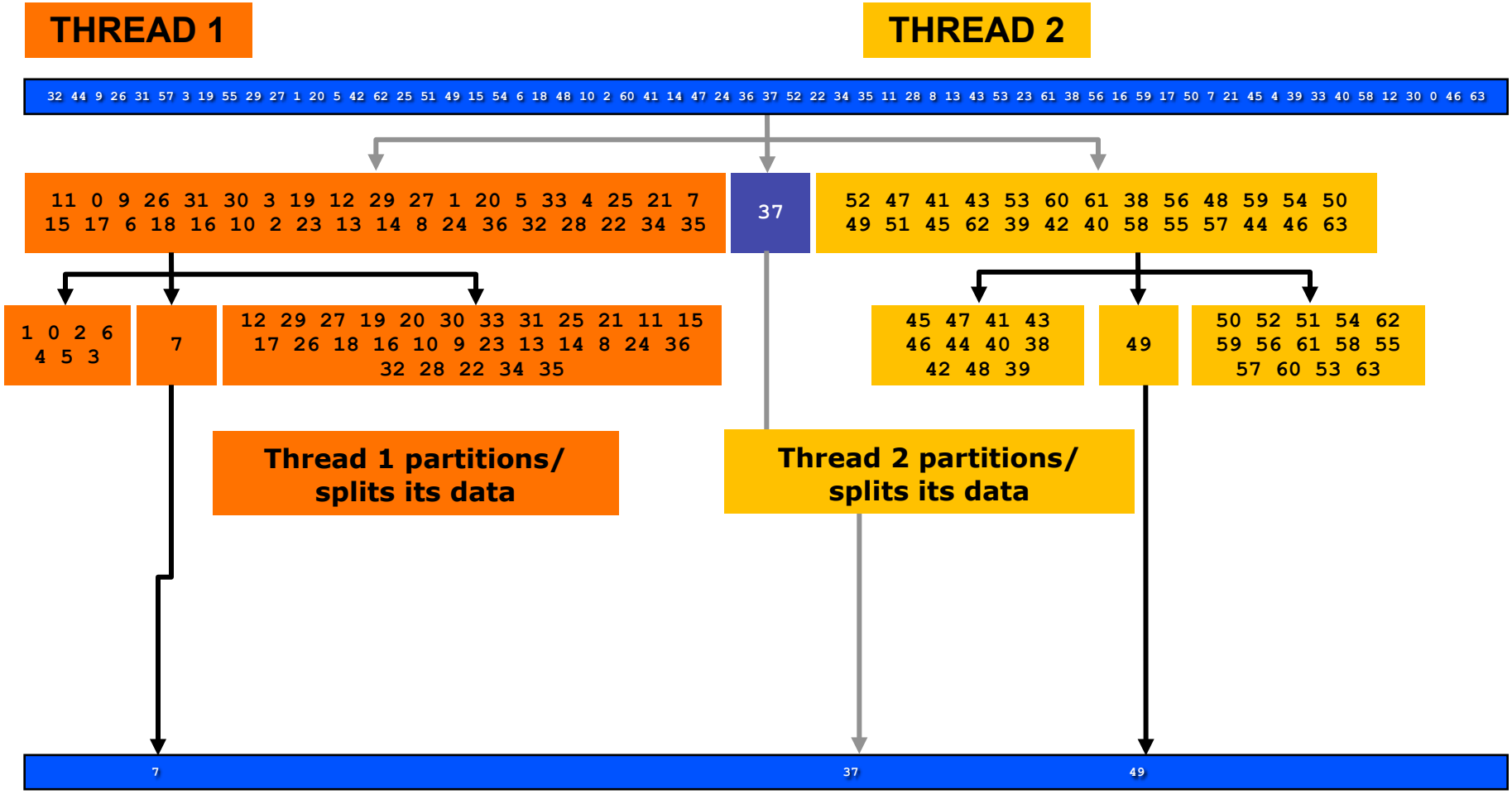
Quicksort – Step 2



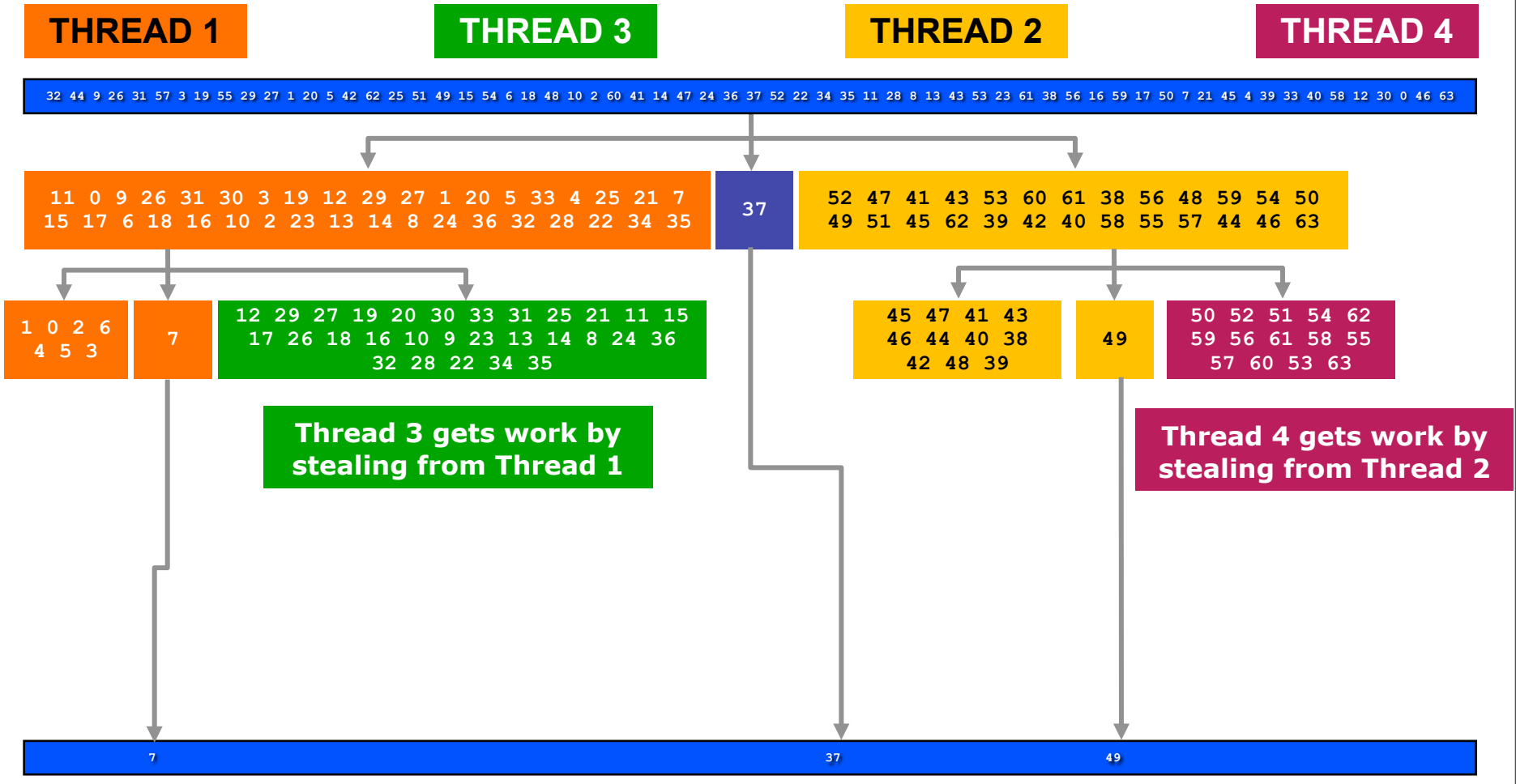
Quicksort – Step 2



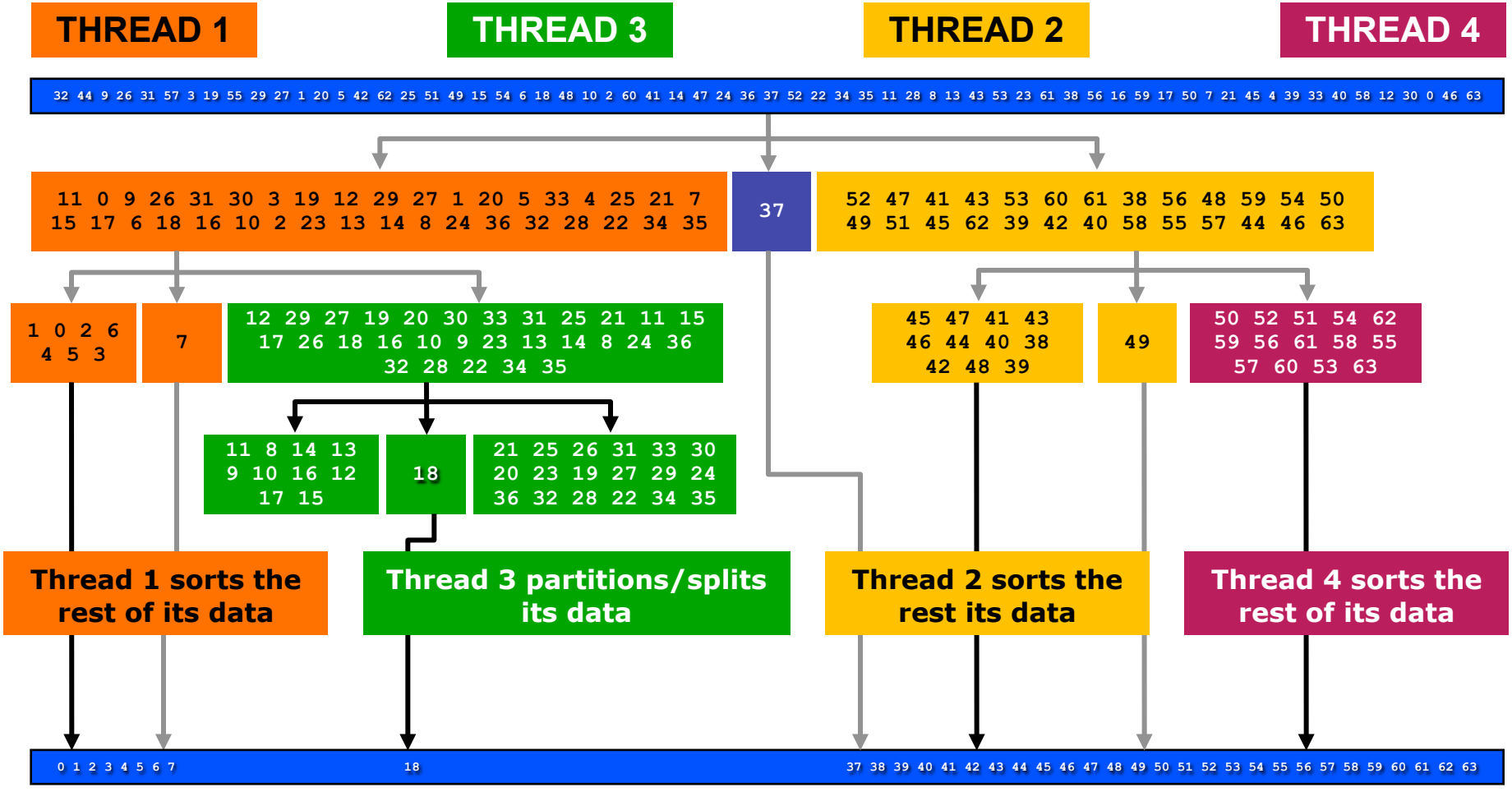
Quicksort – Step 3



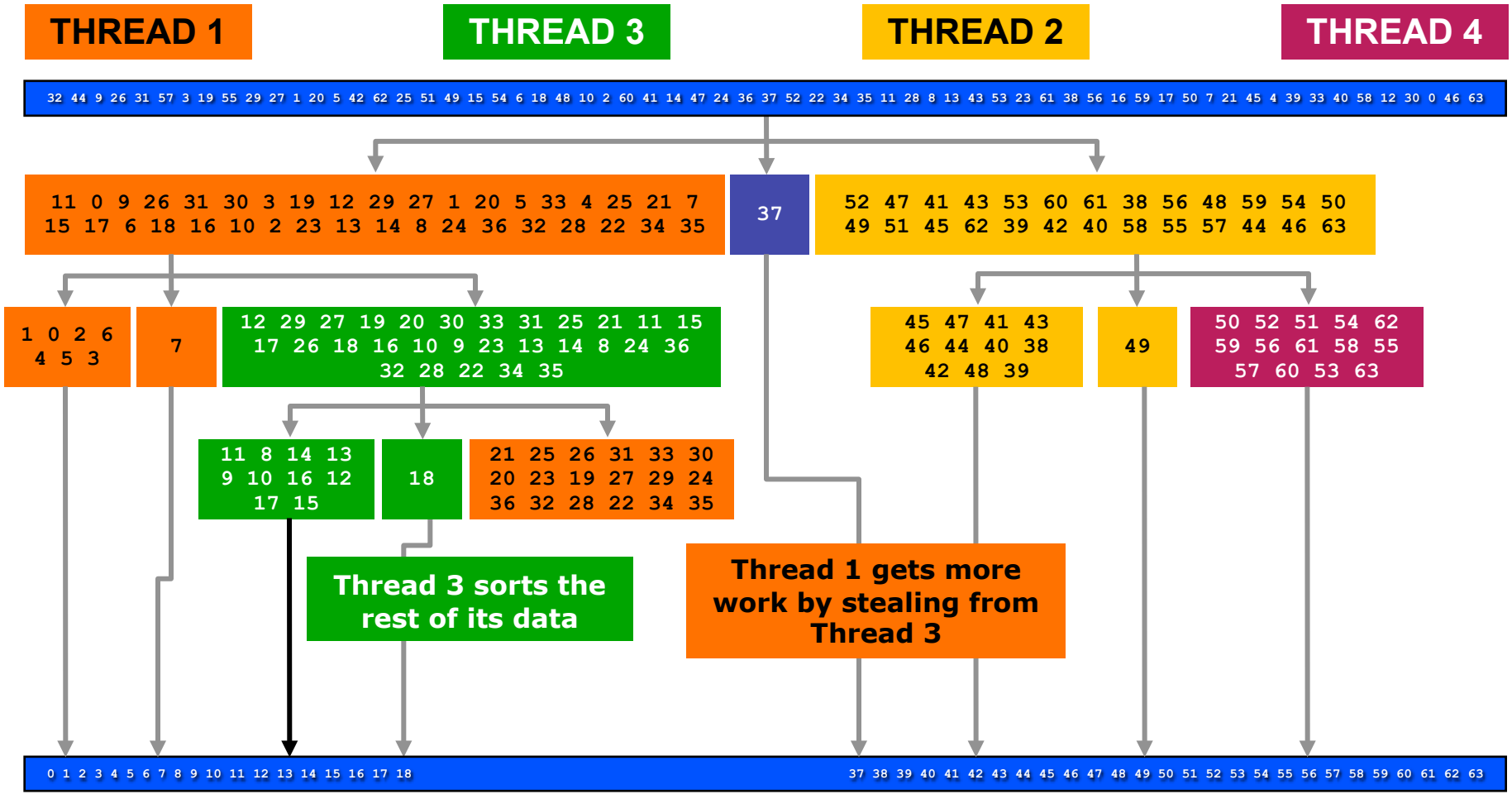
Quicksort – Step 3



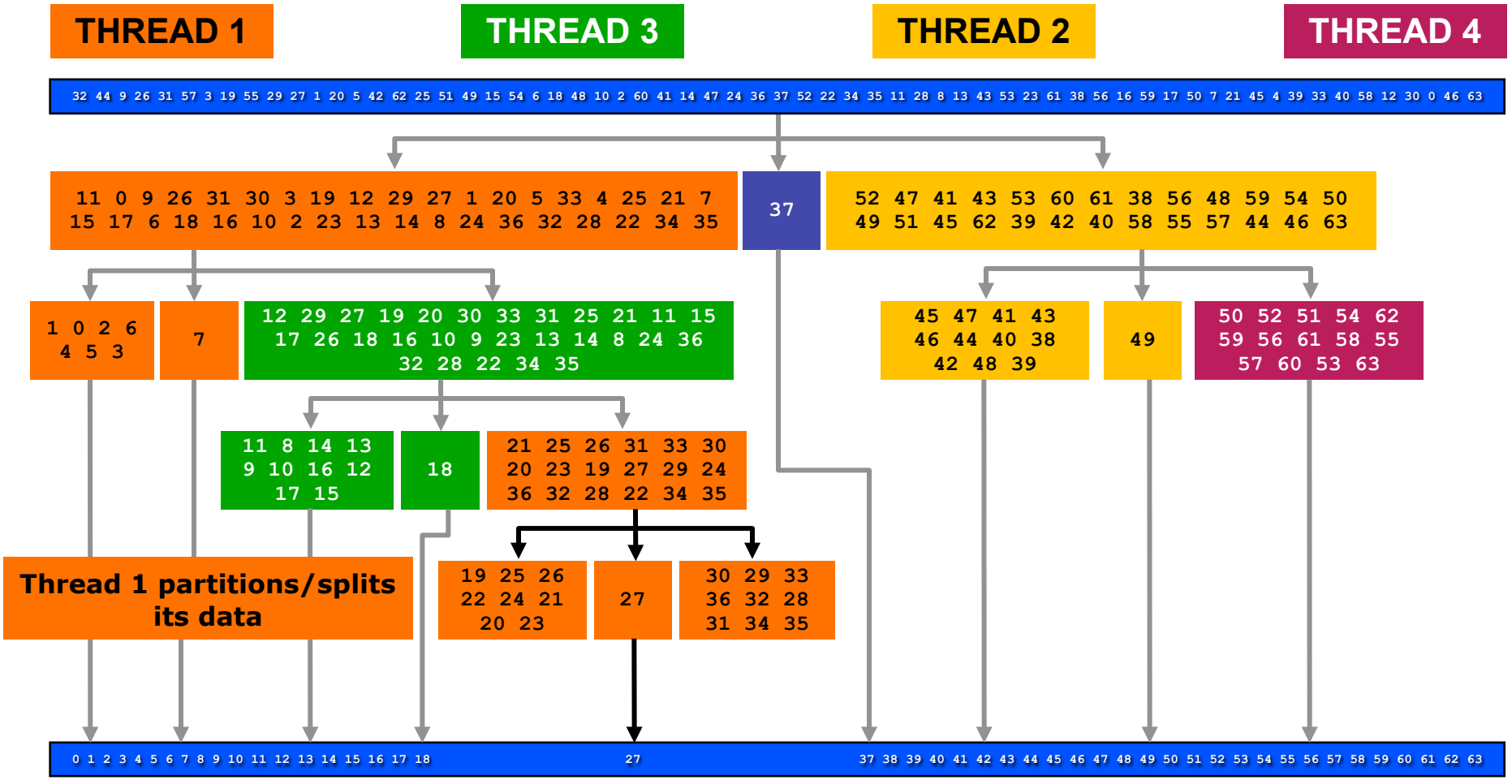
Quicksort – Step 4



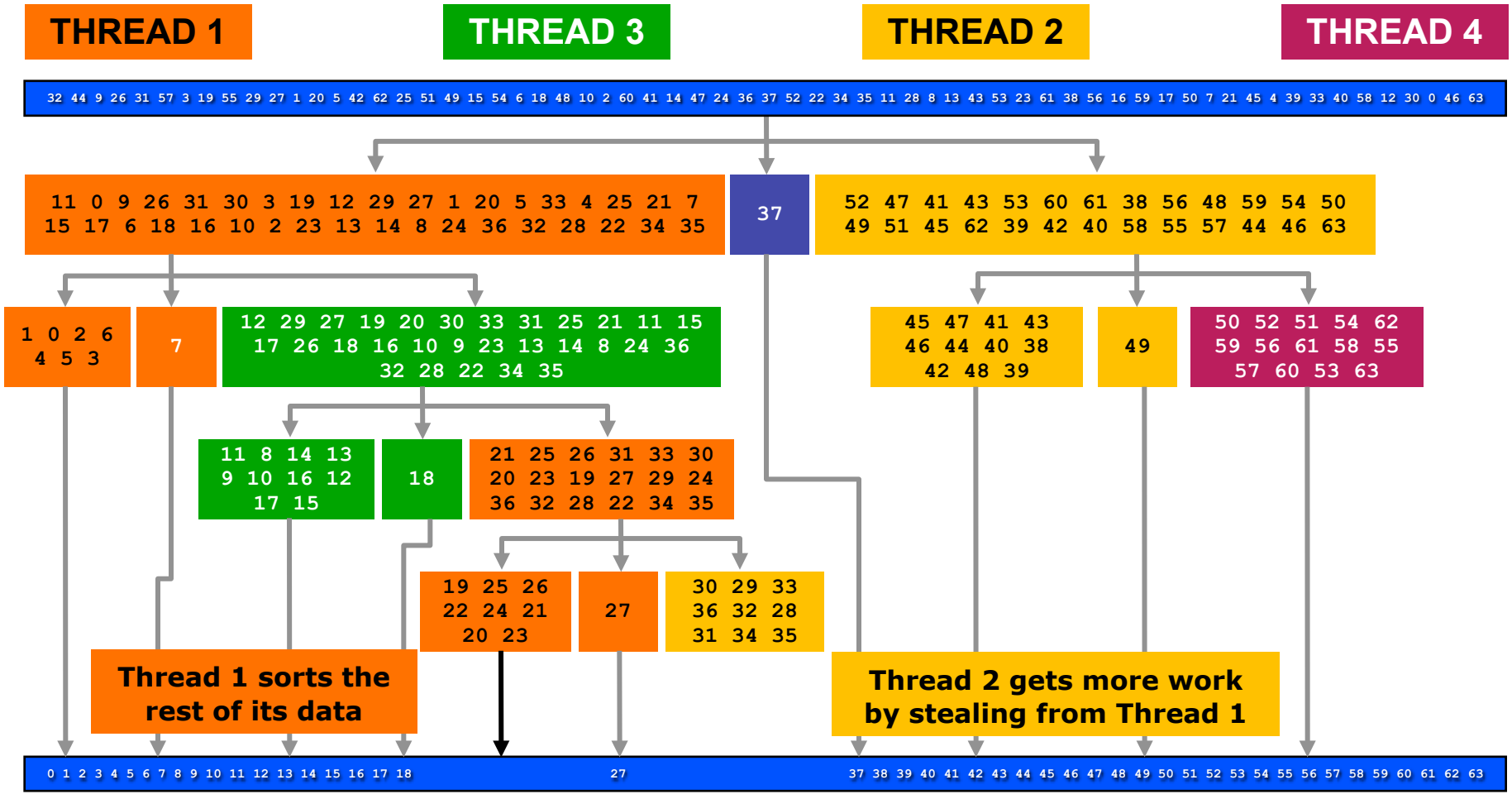
Quicksort – Step 5



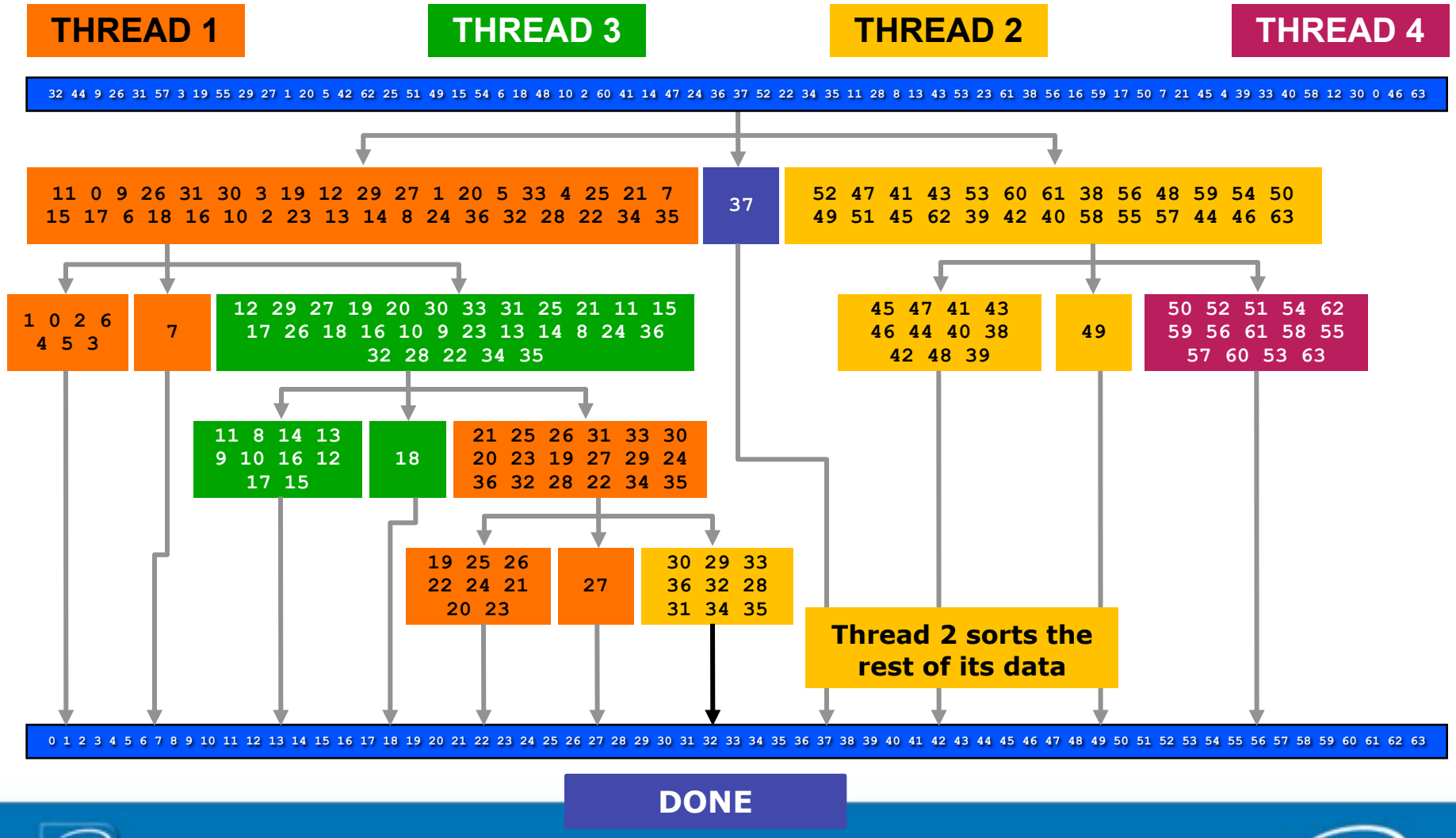
Quicksort – Step 6



Quicksort – Step 6



Quicksort – Step 7



Expressing Tasks in Libraries

Expressing Tasks in Libraries

- Functors

Expressing Tasks in Libraries

- Functors
- C++ "lambda expressions"
  - Coming soon in C++0x standard

Expressing Tasks in Libraries

- Functors
- C++ "lambda expressions"
  - Coming soon in C++0x standard
- "Blocks"
  - Apple's extension to C/C++/Obj-C