From Concurrency to Para*ll*elism

an illustrated guide to multi-core parallelism in Clojure

David Edgar Liebke

clojureconj

Durham, NC

23 October 2010

summary

Concurrency is commonly mistaken for parallelism, but the two are distinct concepts. Concurrency is concerned with managing access to shared state from different threads, whereas parallelism is concerned with utilizing multiple processors/cores to improve the performance of a computation.

Clojure has successfully improved the state of concurrent programming with its many *concurrency primitives*, and now the goal is to do the same for multi-core parallel programming, by introducing new *parallel processing* features that work with Clojure's existing data structures.

Clojure's original parallel processing function, *pmap*, will soon be joined by *pvmap* and *pvreduce,* based on JSR 166 and Doug Lea's *Fork/Join* Framework. From these building blocks, and the *fjvtree* function that underlies *pvmap* and *pvreduce*, higher-level parallel functions can be developed.

This talk will provide an illustrated walkthrough of the algorithms underlying *pmap*, *pvmap*, and *pvreduce*, comparing their strengths, weaknesses, and performance characteristics; and will conclude with an example of using these *primitives* to write a parallel version of Clojure's *filter* function.

outline

pmap
- *algorithm*
- *weaknesses*
- *performance characteristics*
- *chunking*
- *chunked performance characteristics*

fork-join
- *dequeues*
- *basic algorithm*

persistent-vector
- *overview*

fjvtree, pvmap, and pvreduce
- *algorithm*
- *performance characteristics*

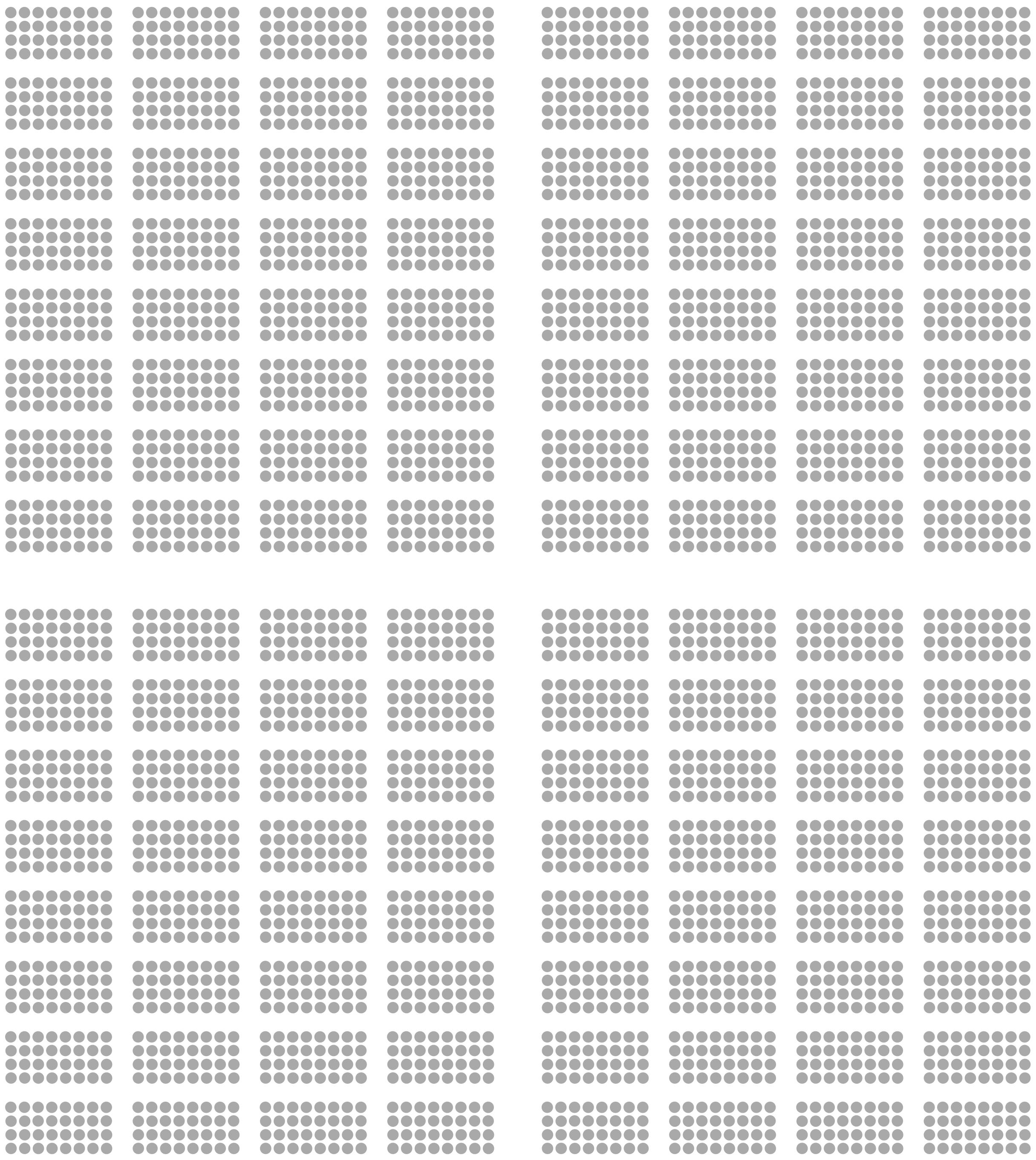
pvfilter
- *implementation*
- *performance characteristics*

pmap

lazy meets parallel

pmap

processors: 2   threads: 4

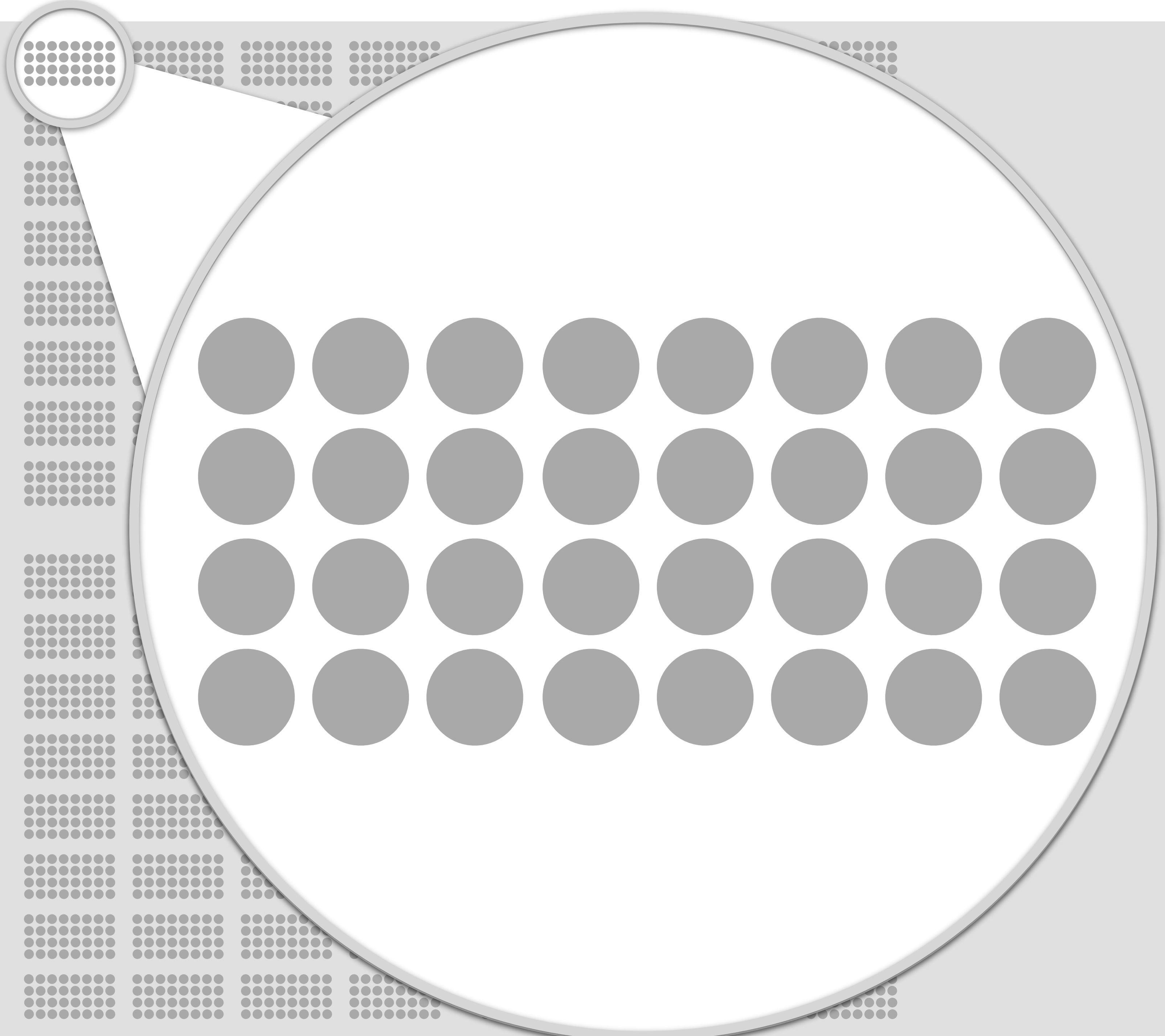


current threads

completed work

pmap

processors: 2 threads: 4

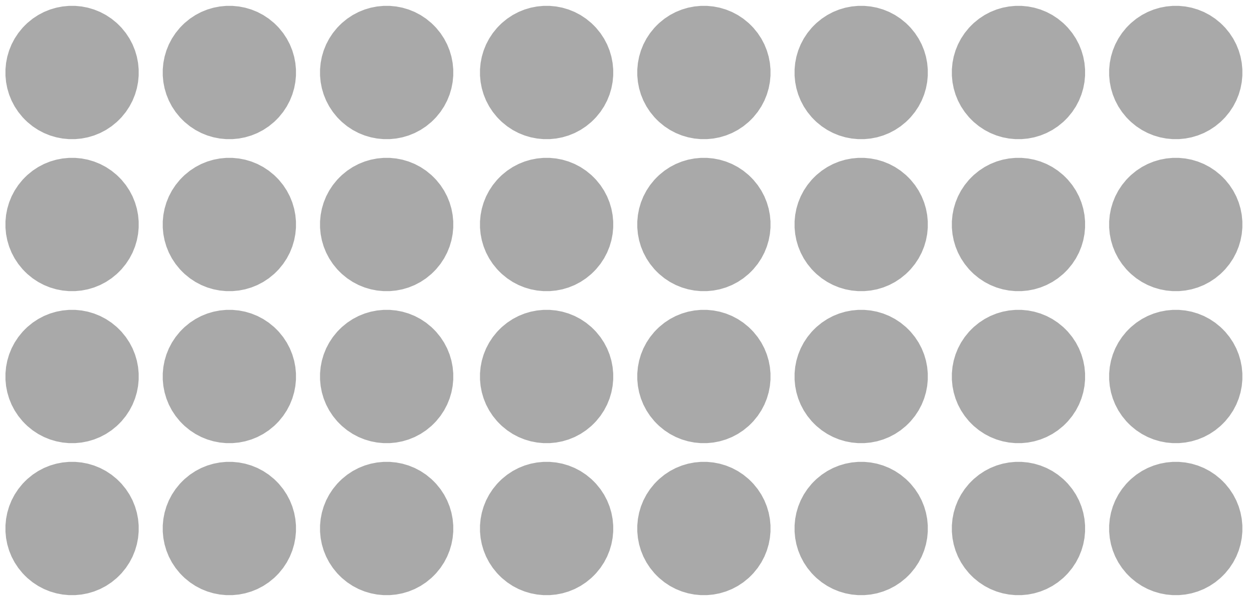


current threads

completed work

pmap

processors: 2 threads: 4



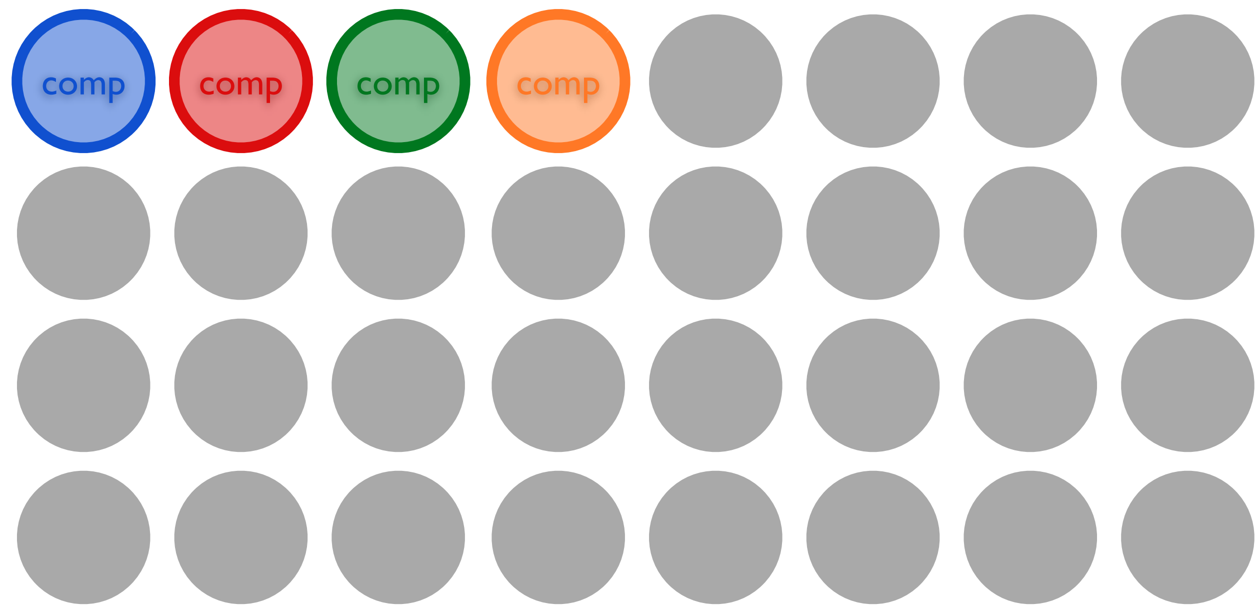
The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

completed work

pmap

processors: 2 threads: 4



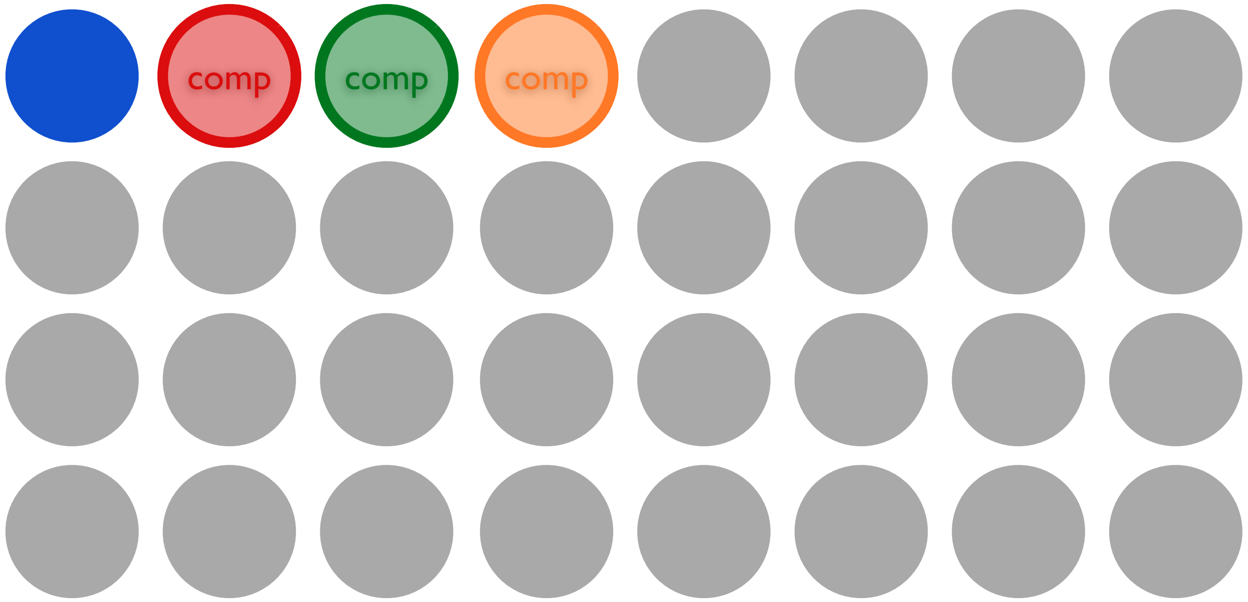
The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

completed work

pmap

processors: 2 threads: 4

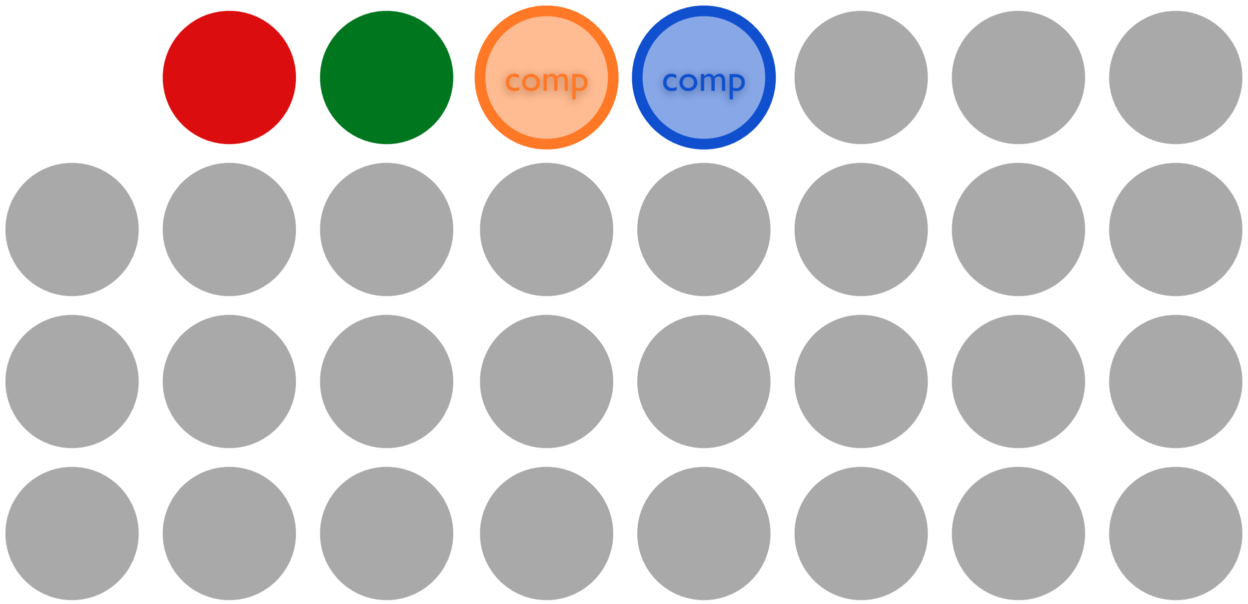


The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads completed work

pmap

processors: 2 threads: 4



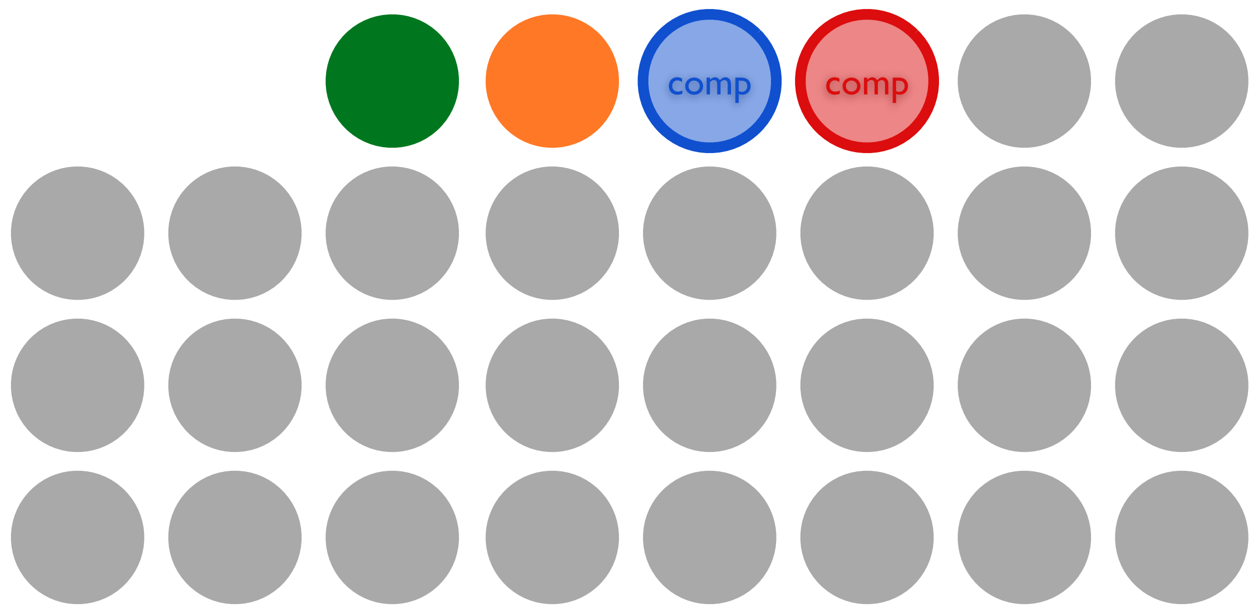
The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

completed work

pmap

processors: 2 threads: 4



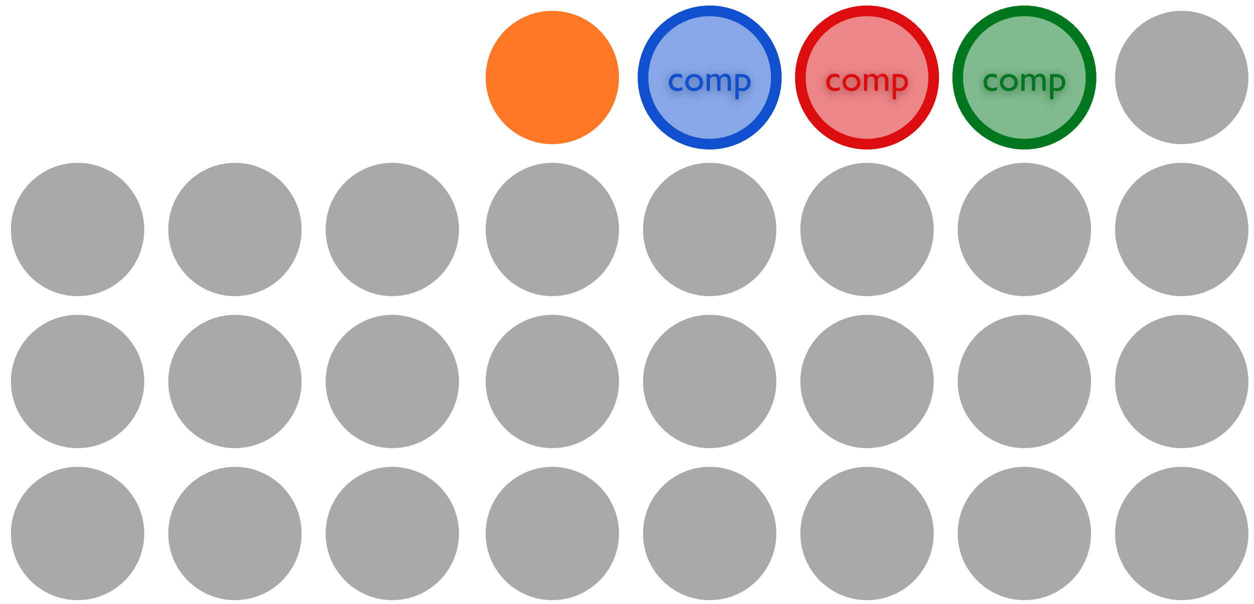
The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

completed work

pmap

processors: 2   threads: 4



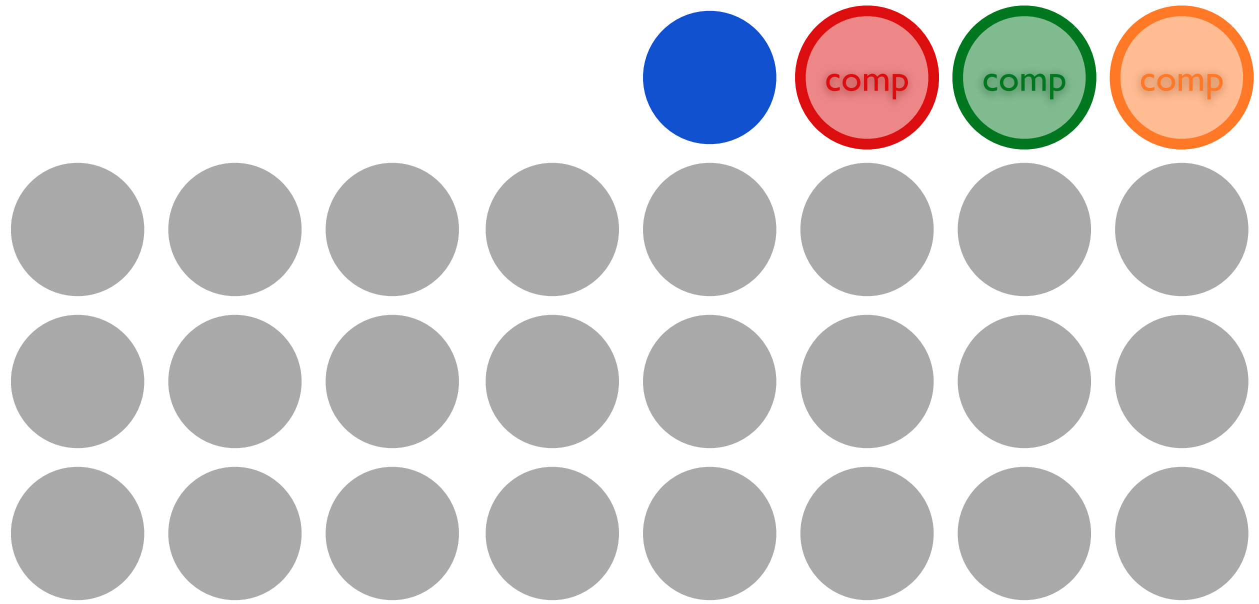
The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

completed work

pmap

processors: 2 threads: 4



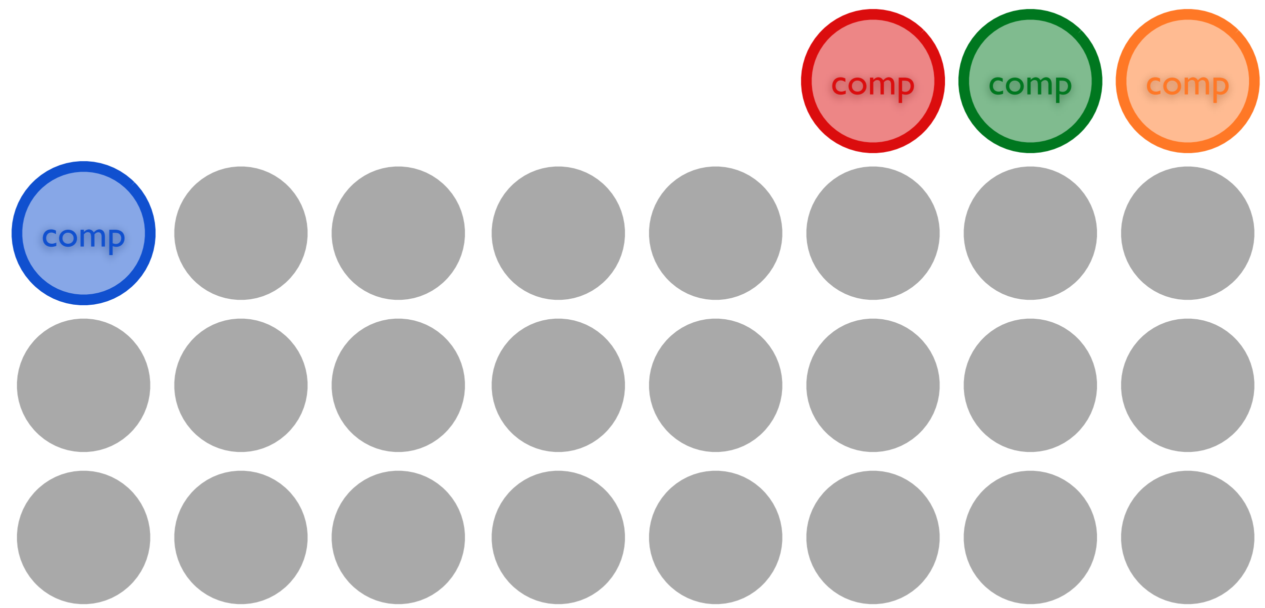
The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn’t process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

completed work

pmap

processors: 2 threads: 4



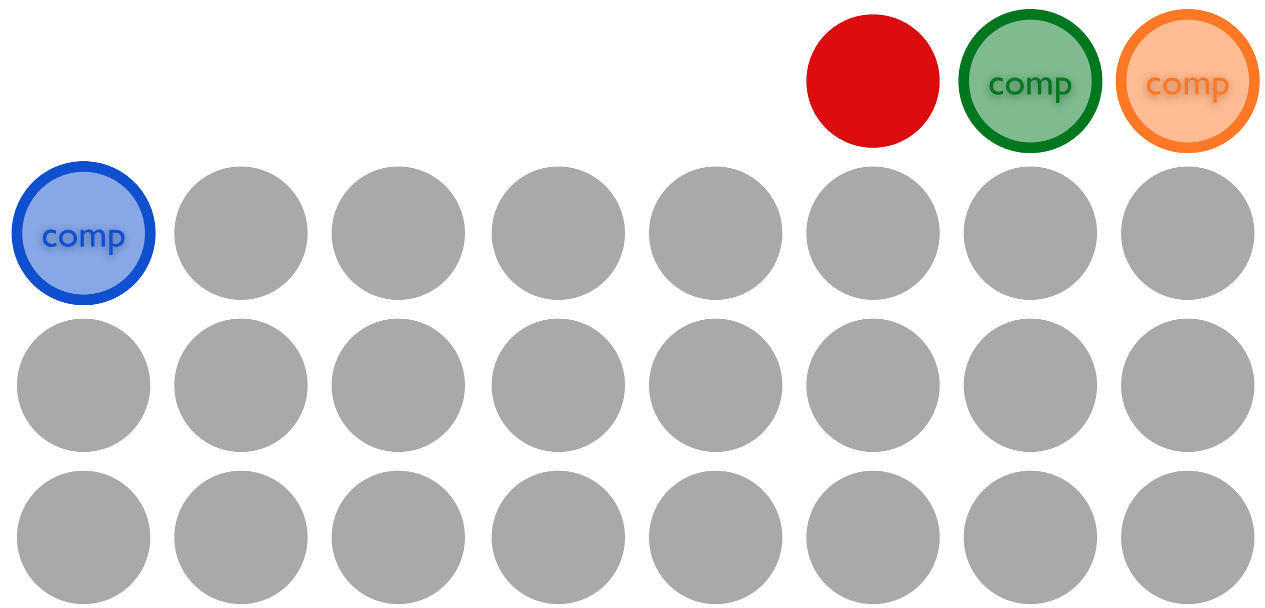
The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

completed work

pmap

processors: 2 threads: 4



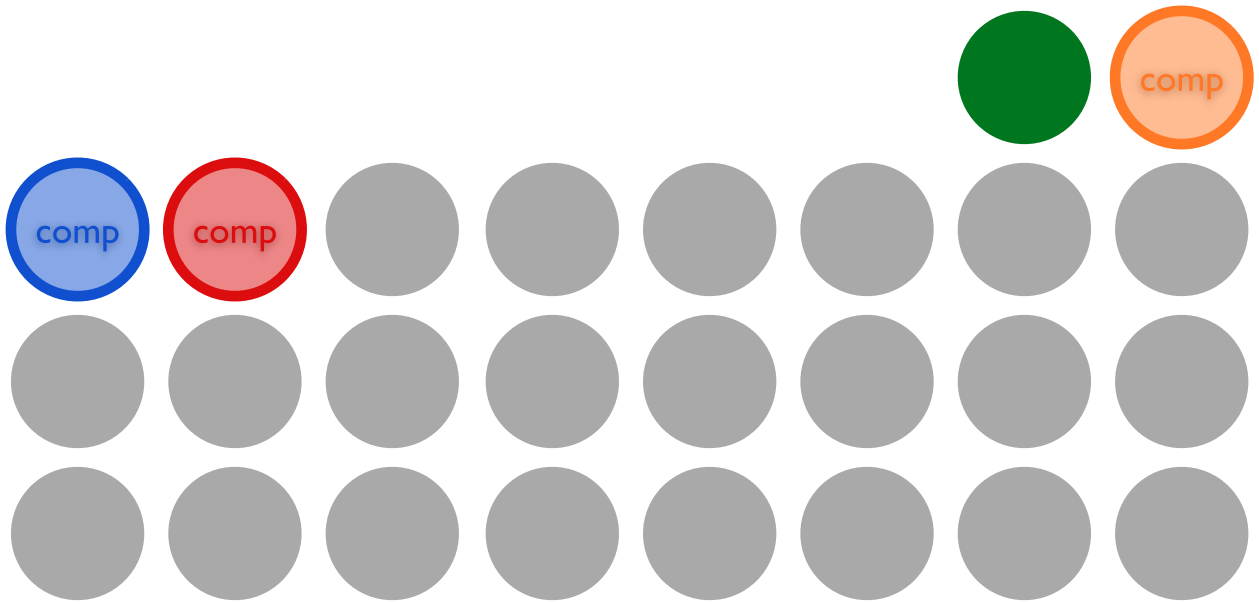
The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

completed work

pmap

processors: 2 threads: 4



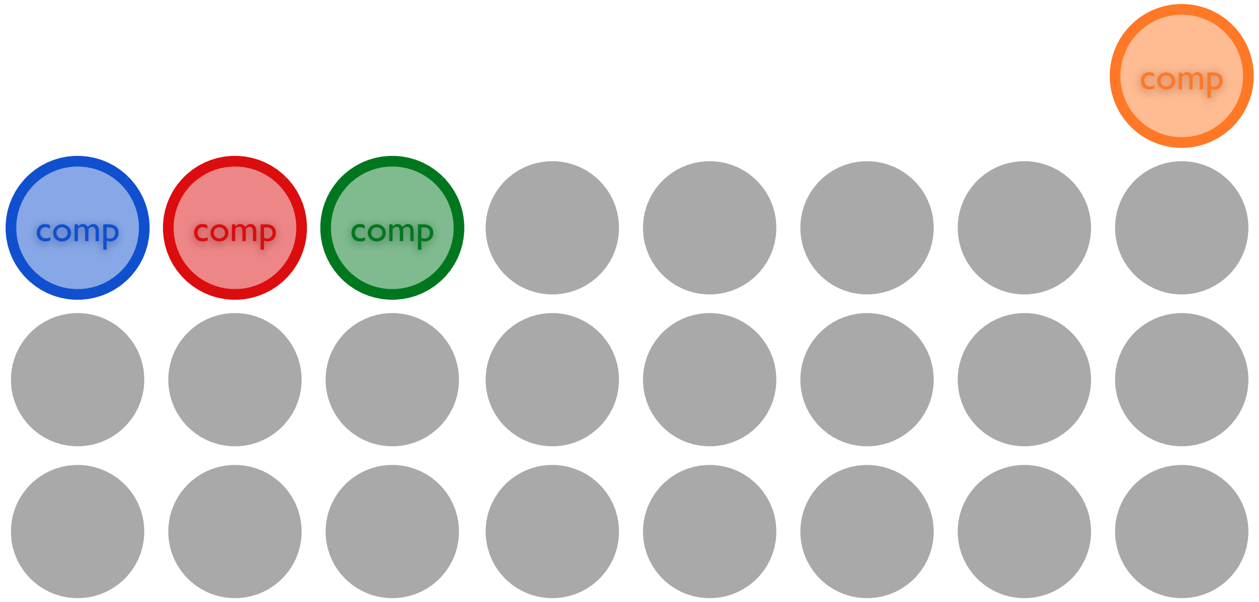
The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

completed work

pmap

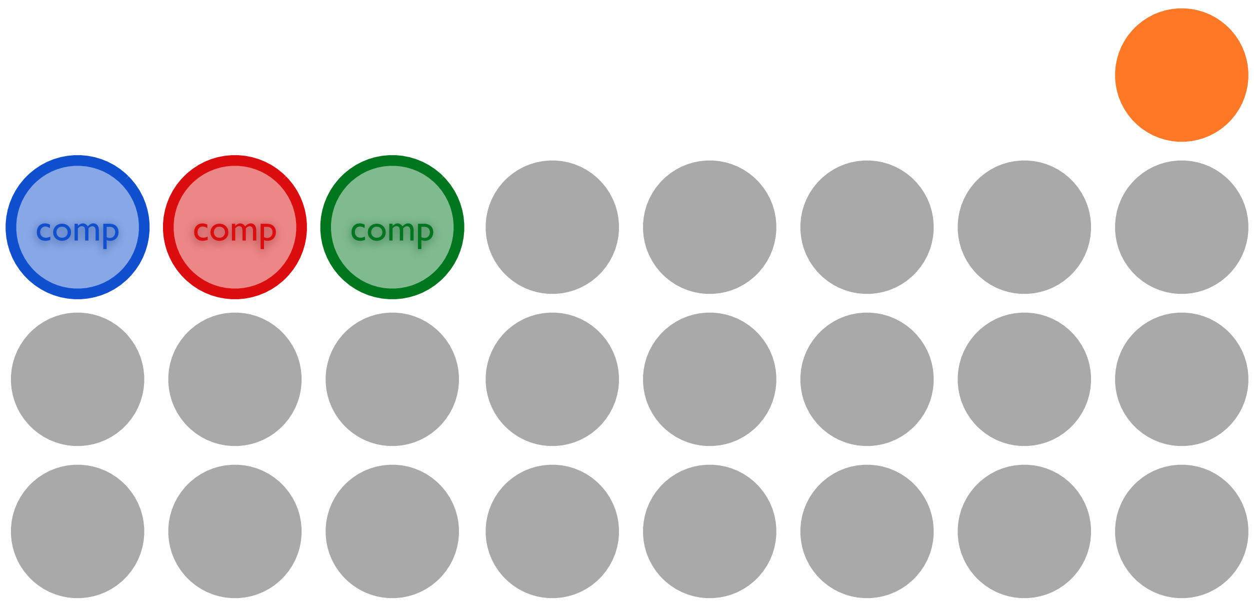
processors: 2 threads: 4



The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

completed work



The *pmap* function is semi-lazy, meaning it tries to keep just enough threads running so that all the processors are utilized, but doesn't process its entire input. It does this by using *futures* to invoke the function being mapped on just enough input values, but no more.

current threads

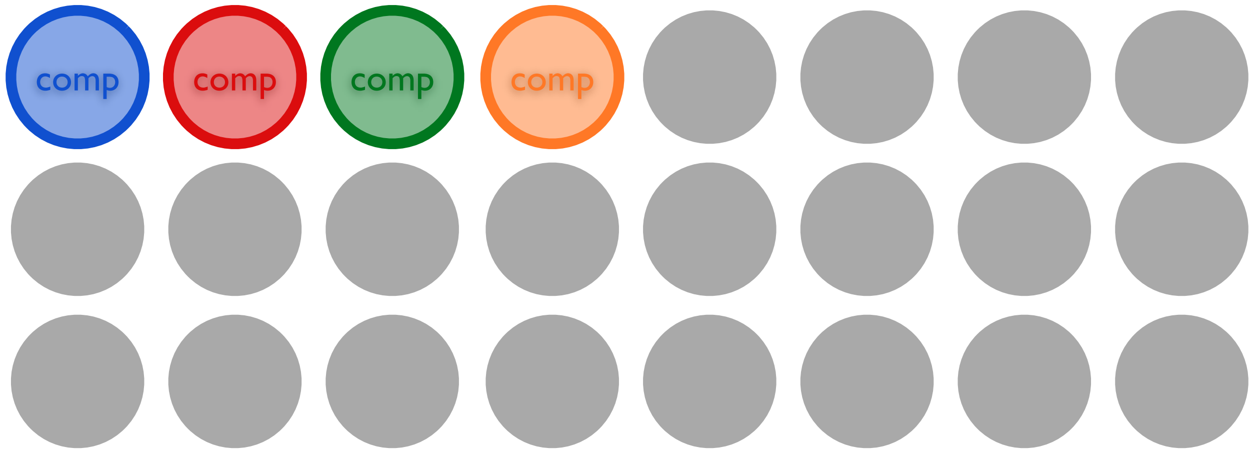
completed work

pmap

when lazy isn't parallel

pmap

processors: 2 threads: 4



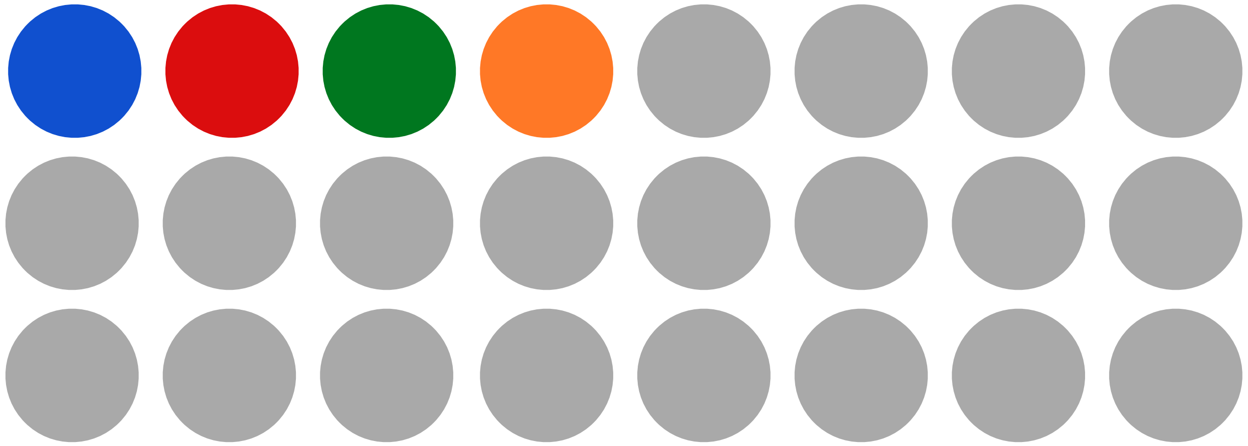
Because pmap is semi-lazy, it is necessary that the process consuming its output can do so at a faster rate than the process producing its output, otherwise all the processors won't be utilized.

current threads

completed work

pmap

processors: 2 threads: 4



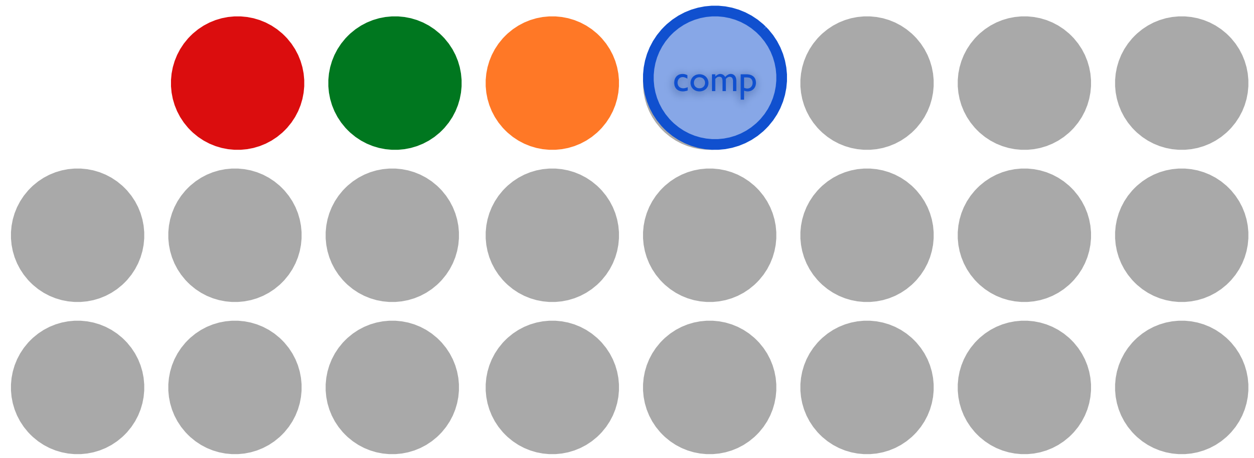
Because pmap is semi-lazy, it is necessary that the process consuming its output can do so at a faster rate than the process producing its output, otherwise all the processors won’t be utilized.

current threads

completed work

pmap

processors: 2   threads: 4



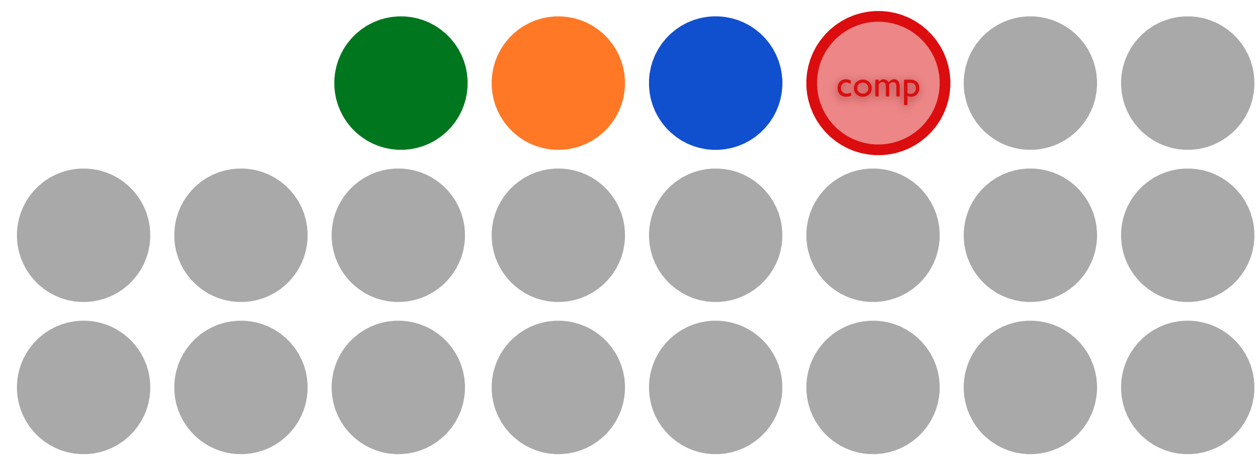
Because pmap is semi-lazy, it is necessary that the process consuming its output can do so at a faster rate than the process producing its output, otherwise all the processors won't be utilized.

current threads

completed work

pmap

processors: 2 threads: 4



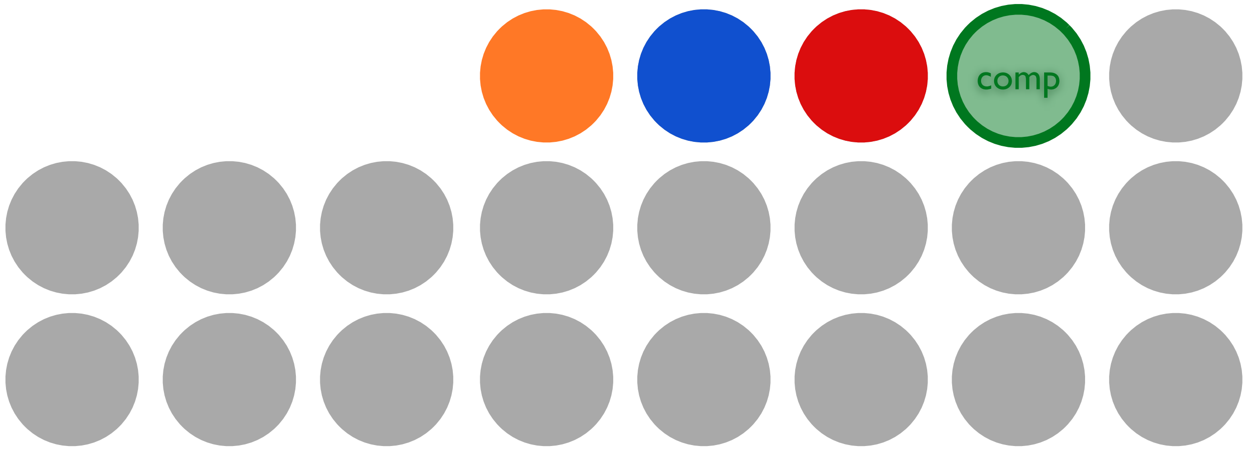
Because pmap is semi-lazy, it is necessary that the process consuming its output can do so at a faster rate than the process producing its output, otherwise all the processors won't be utilized.

current threads

completed work

pmap

processors: 2 threads: 4



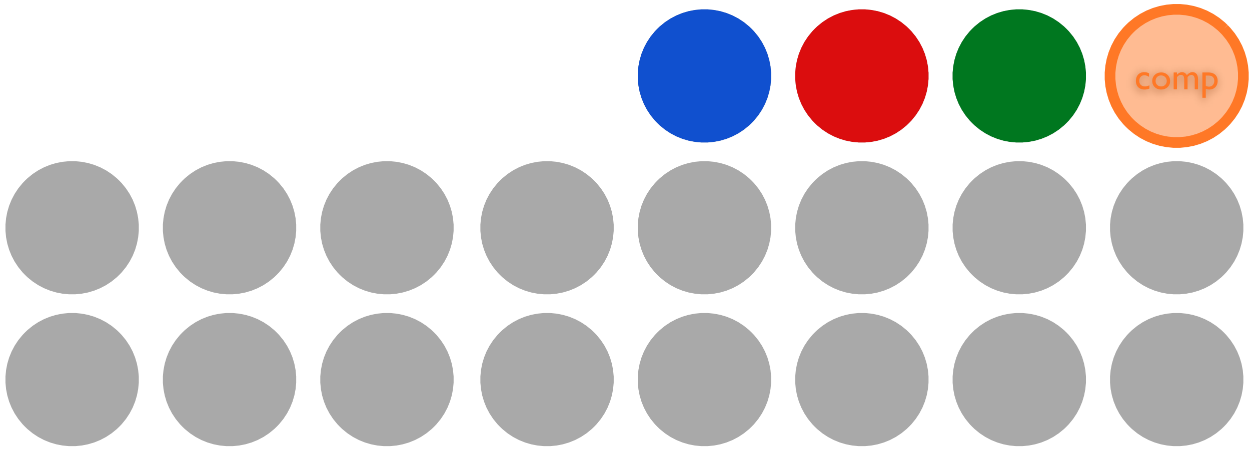
Because pmap is semi-lazy, it is necessary that the process consuming its output can do so at a faster rate than the process producing its output, otherwise all the processors won't be utilized.

current threads

completed work

pmap

processors: 2 threads: 4



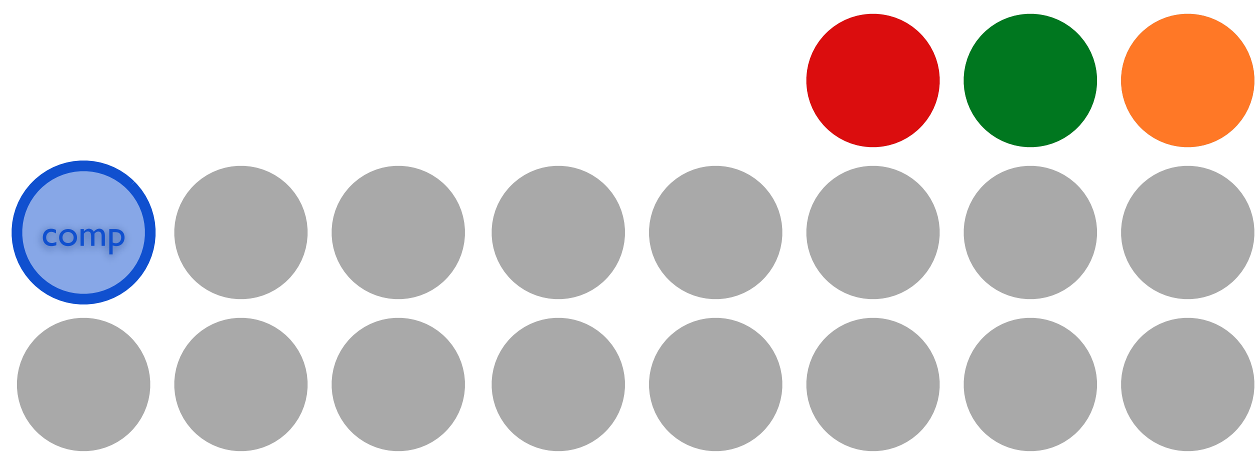
Because pmap is semi-lazy, it is necessary that the process consuming its output can do so at a faster rate than the process producing its output, otherwise all the processors won't be utilized.

current threads

completed work

pmap

processors: 2   threads: 4



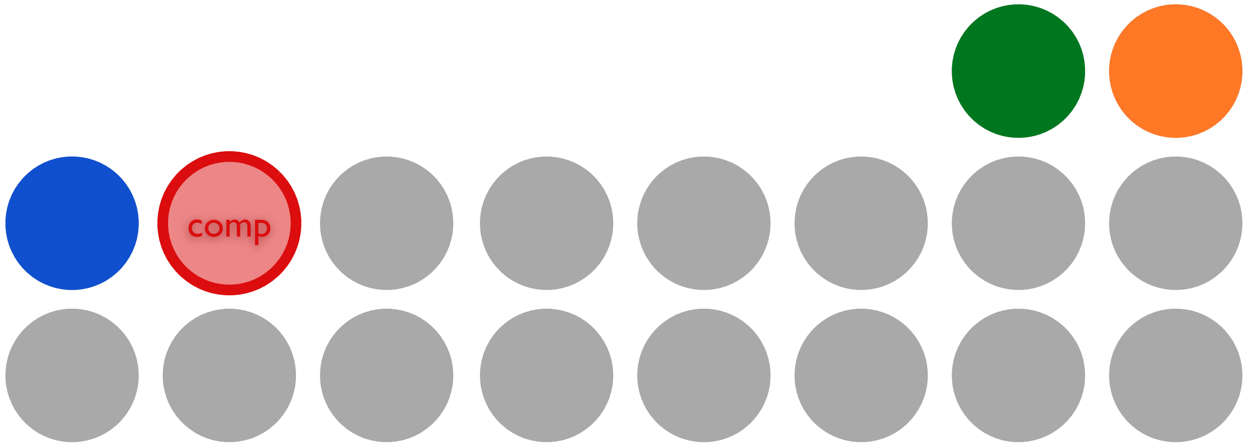
Because pmap is semi-lazy, it is necessary that the process consuming its output can do so at a faster rate than the process producing its output, otherwise all the processors won't be utilized.

current threads

completed work

pmap

processors: 2 threads: 4



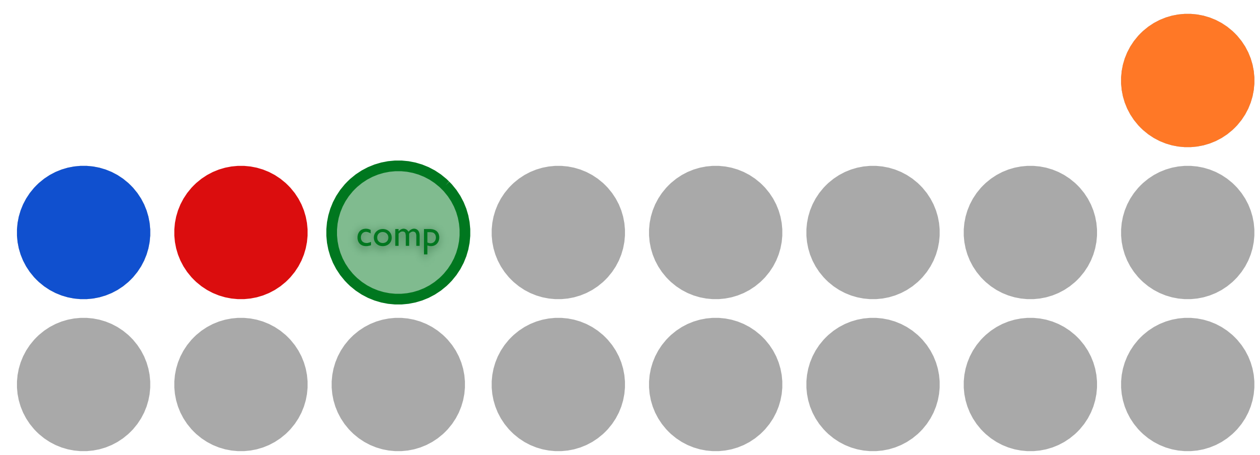
Because pmap is semi-lazy, it is necessary that the process consuming its output can do so at a faster rate than the process producing its output, otherwise all the processors won't be utilized.

current threads

completed work

pmap

processors: 2   threads: 4



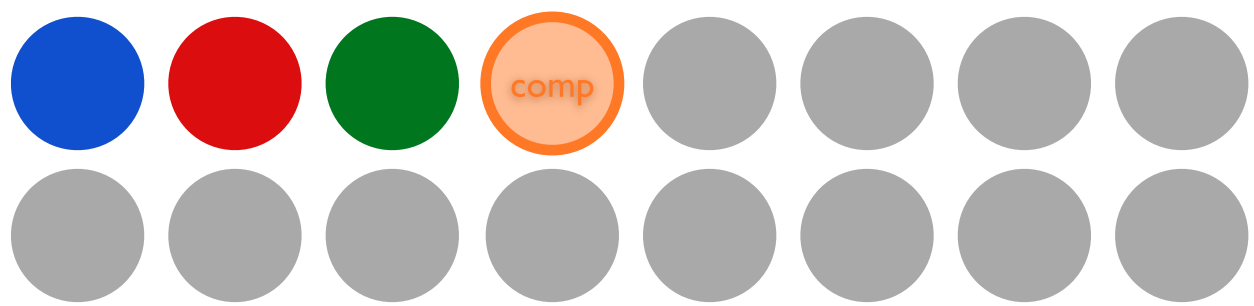
Because pmap is semi-lazy, it is necessary that the process consuming its output can do so at a faster rate than the process producing its output, otherwise all the processors won’t be utilized.

current threads

completed work

pmap

processors: 2 threads: 4



Because pmap is semi-lazy, it is necessary that the process consuming its output can do so at a faster rate than the process producing its output, otherwise all the processors won't be utilized.

current threads

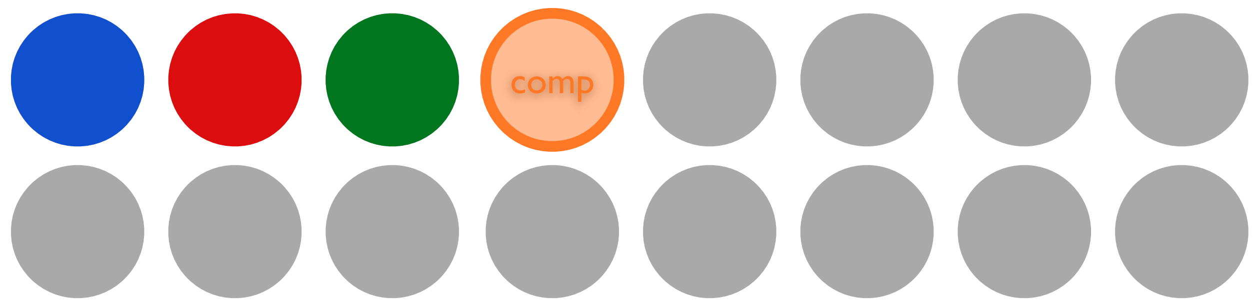
completed work

pmap

uneven loads

pmap

processors: 2 threads: 4



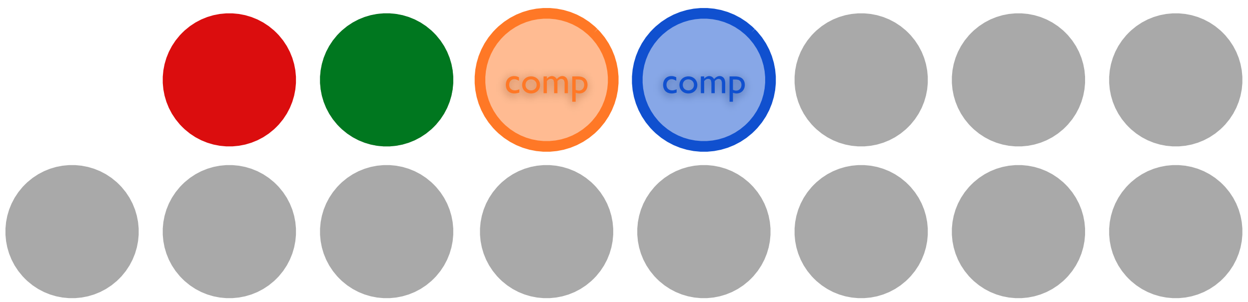
If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

pmap

processors: 2 threads: 4



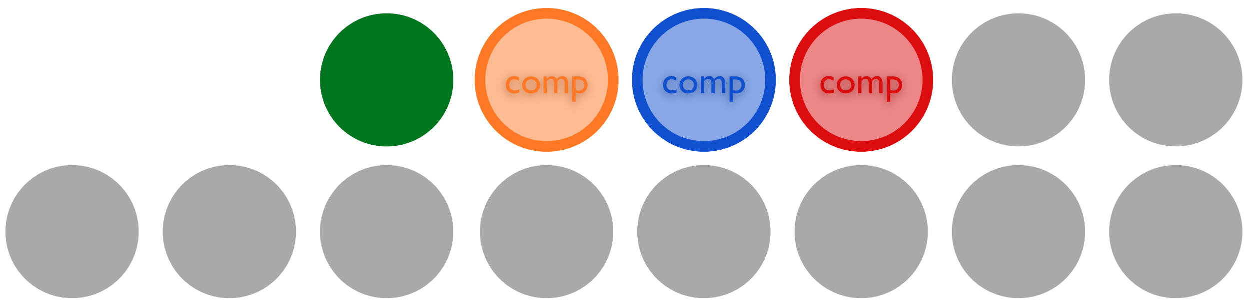
If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

pmap

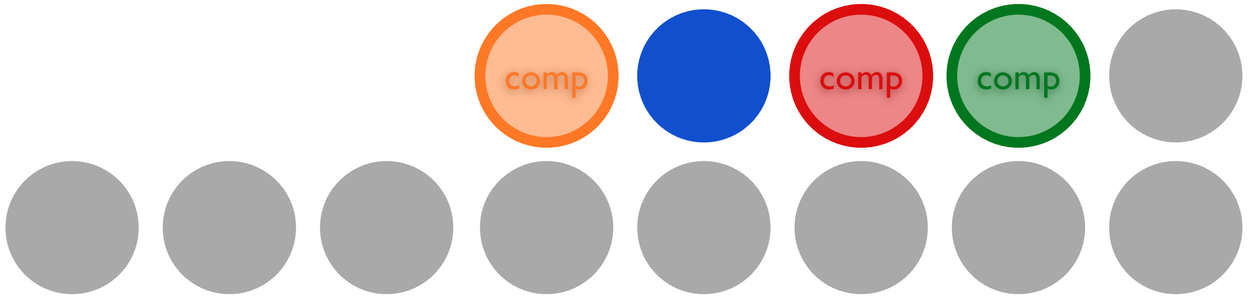
processors: 2 threads: 4



If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

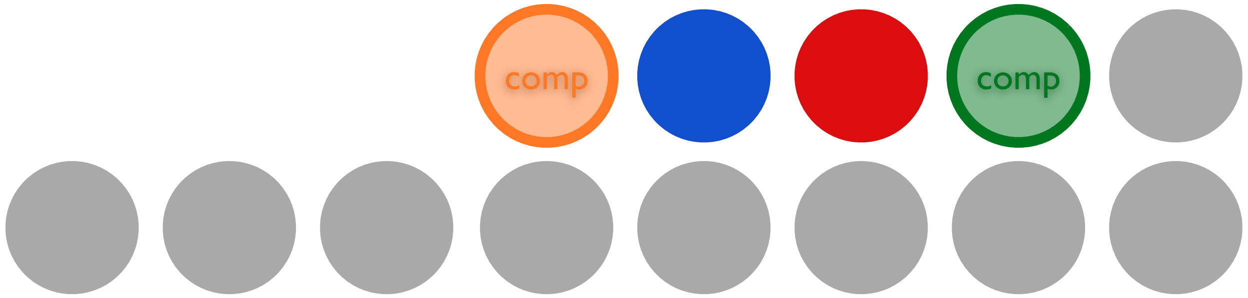


If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

processors: 2 threads: 4

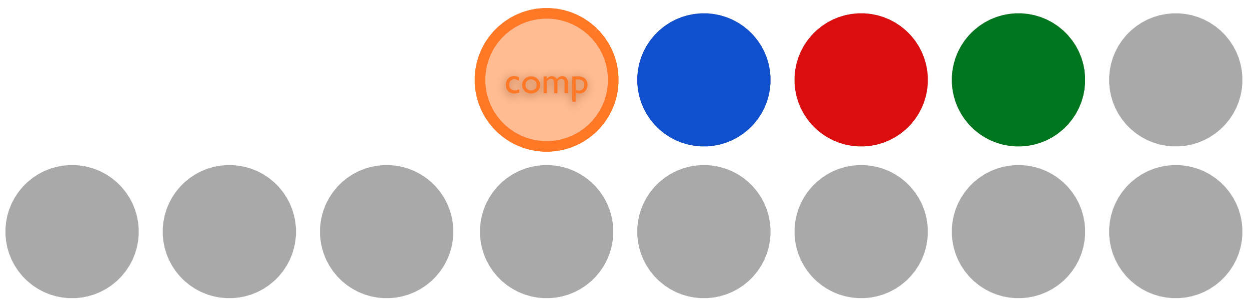


If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

processors: 2 threads: 4



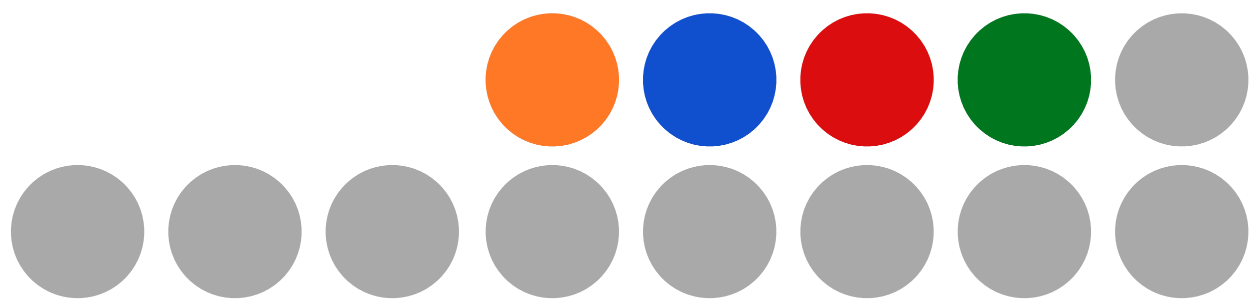
If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

pmap

processors: 2 threads: 4



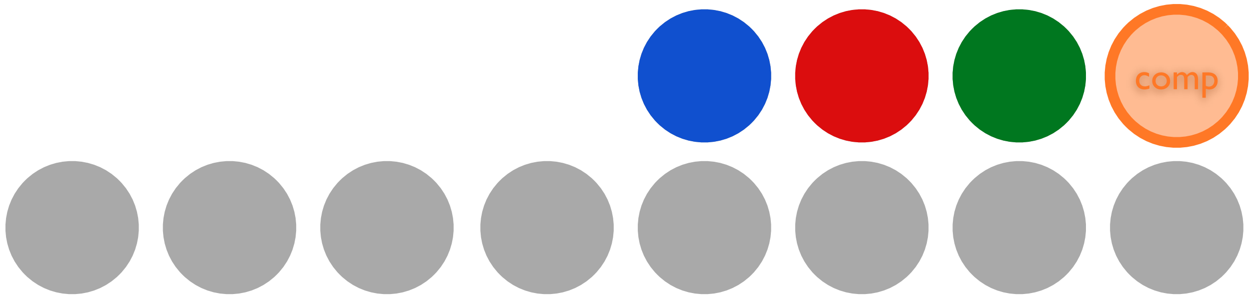
If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

pmap

processors: 2 threads: 4



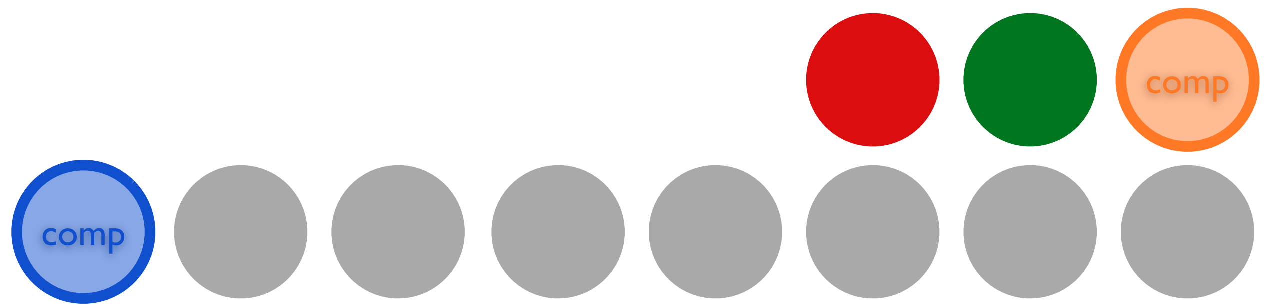
If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

pmap

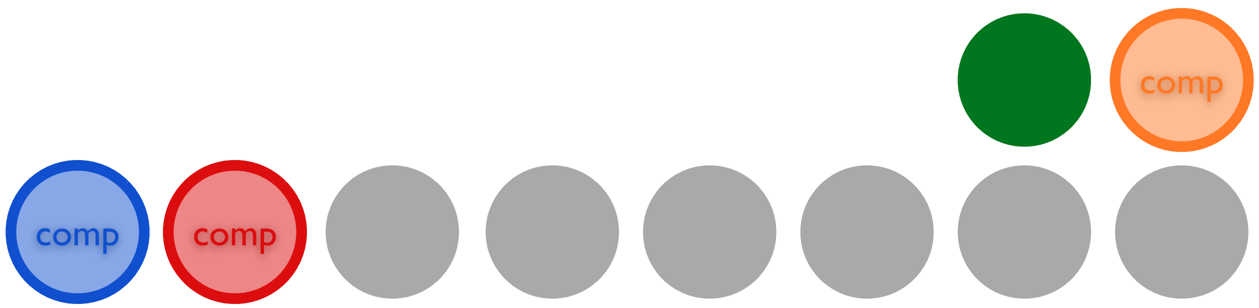
processors: 2 threads: 4



If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work



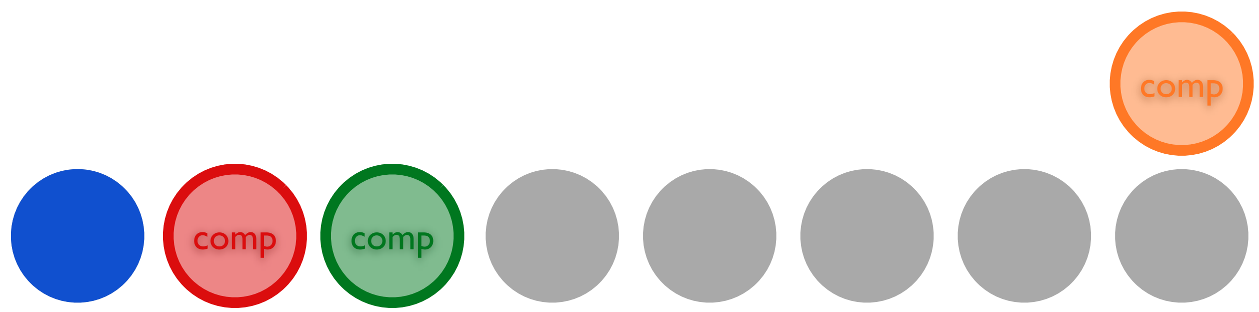
If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

pmap

processors: 2 threads: 4



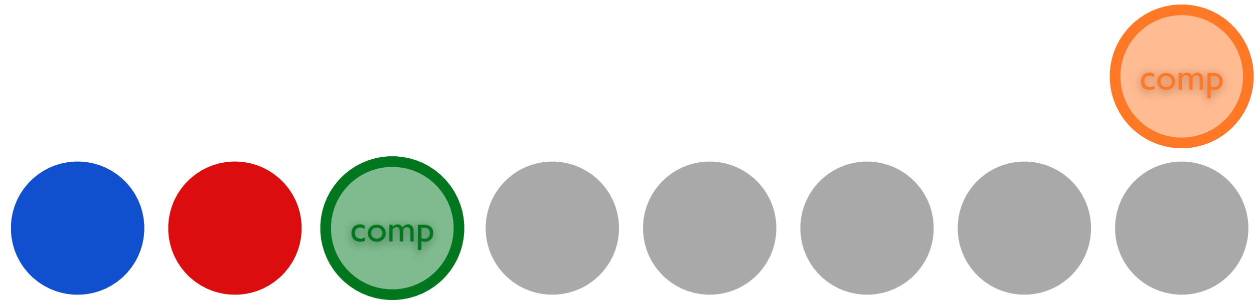
If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

pmap

processors: 2 threads: 4



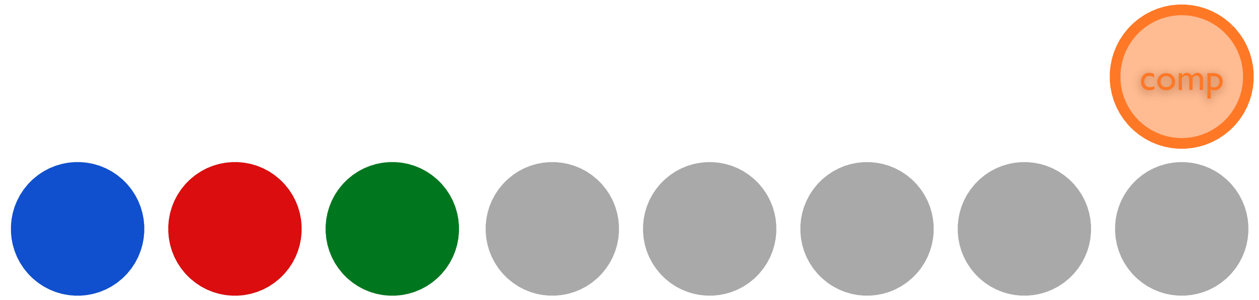
If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

pmap

processors: 2 threads: 4



If one or more processors are returning results at a slower rate than the others, there is no way to redistribute the load. The more responsive processors will complete their work and sit idle until the slowest completes and its result can be consumed.

current threads

completed work

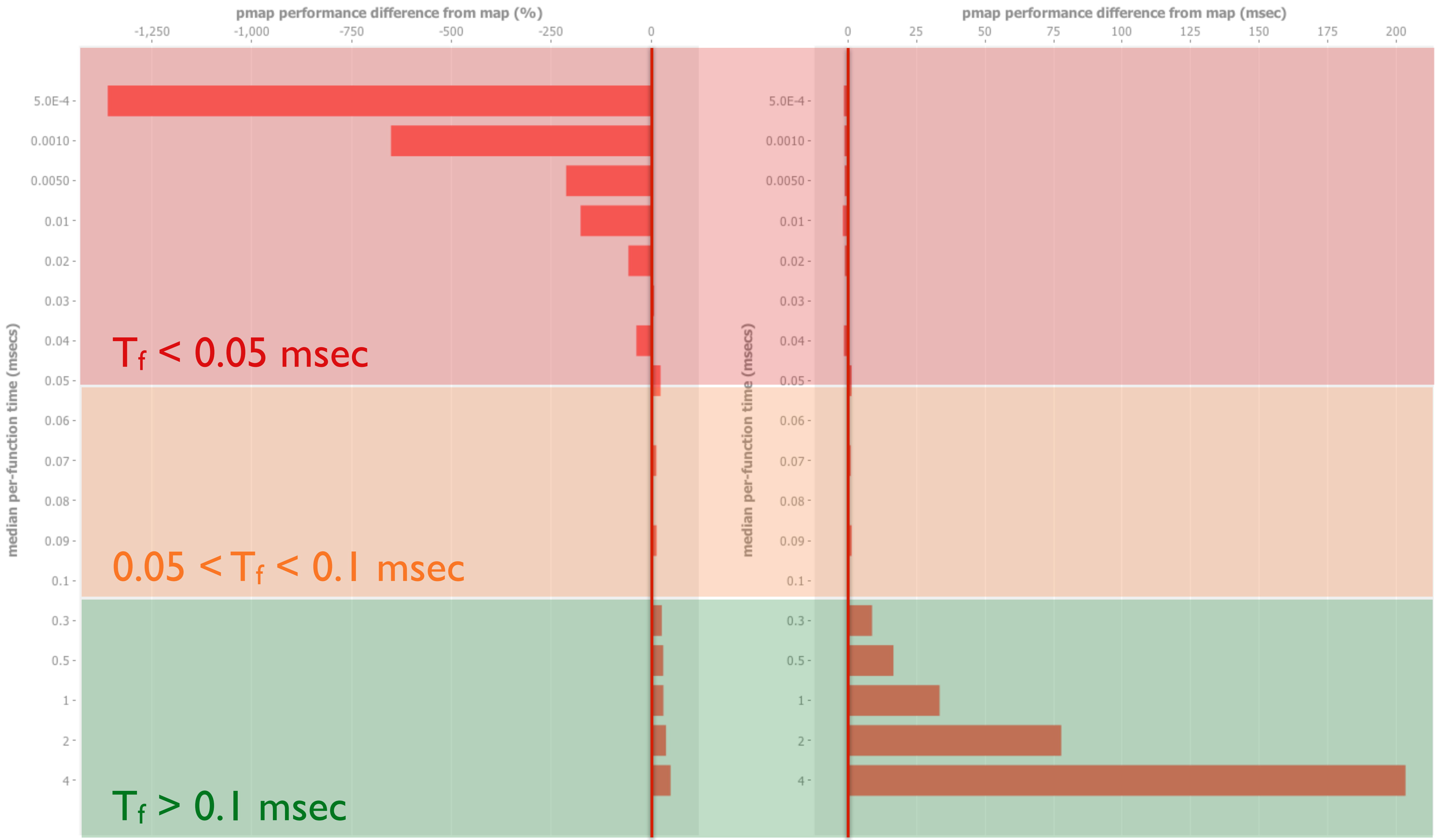
pmap

performance characteristics

pmap

compared to map (2 cores)

* results are the median of 50 samples, where a test function was pmapped over a vector of 100 values

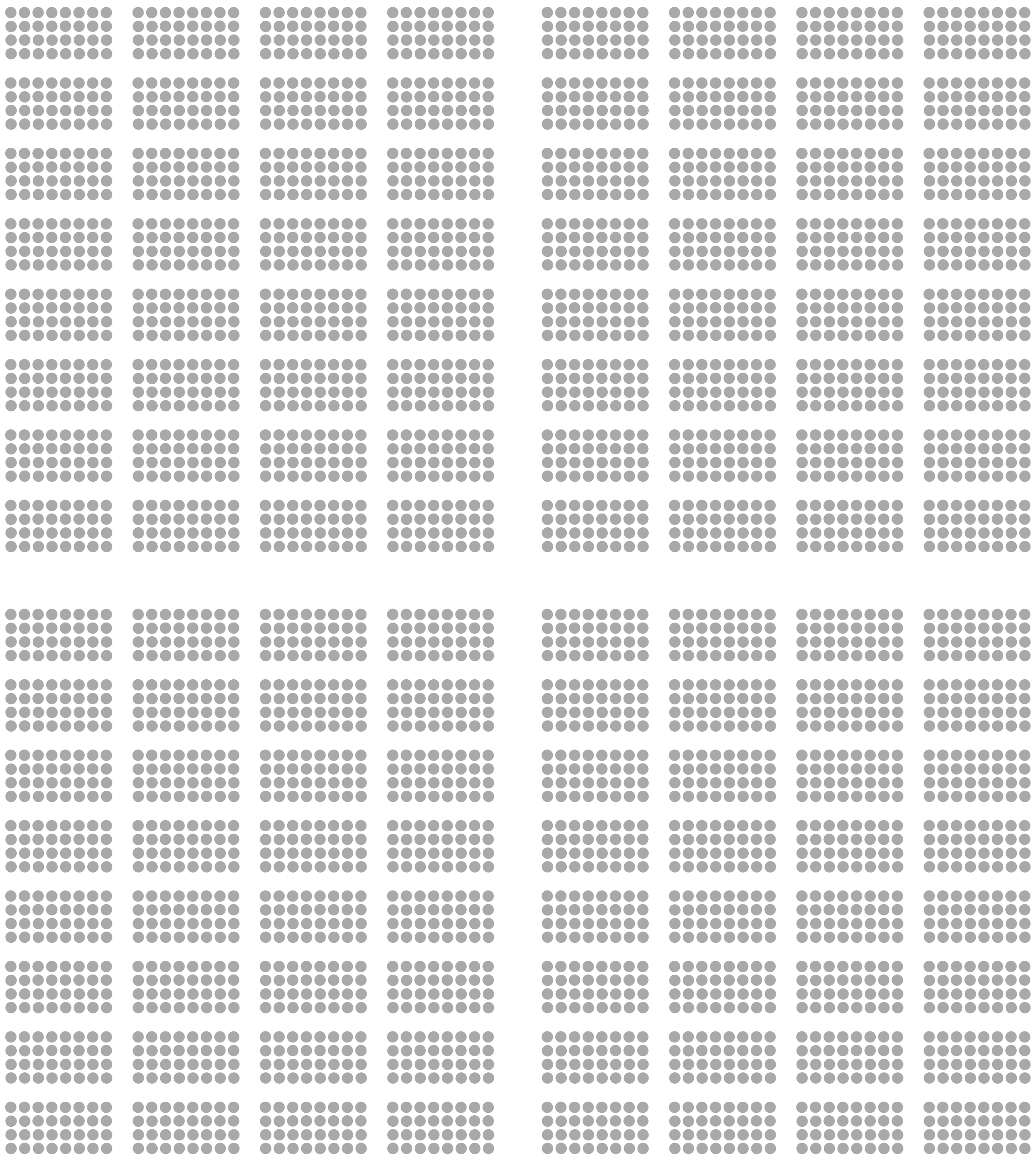


pmap

chunky style

pmap

processors: 2 threads: 4 chunk size: 32

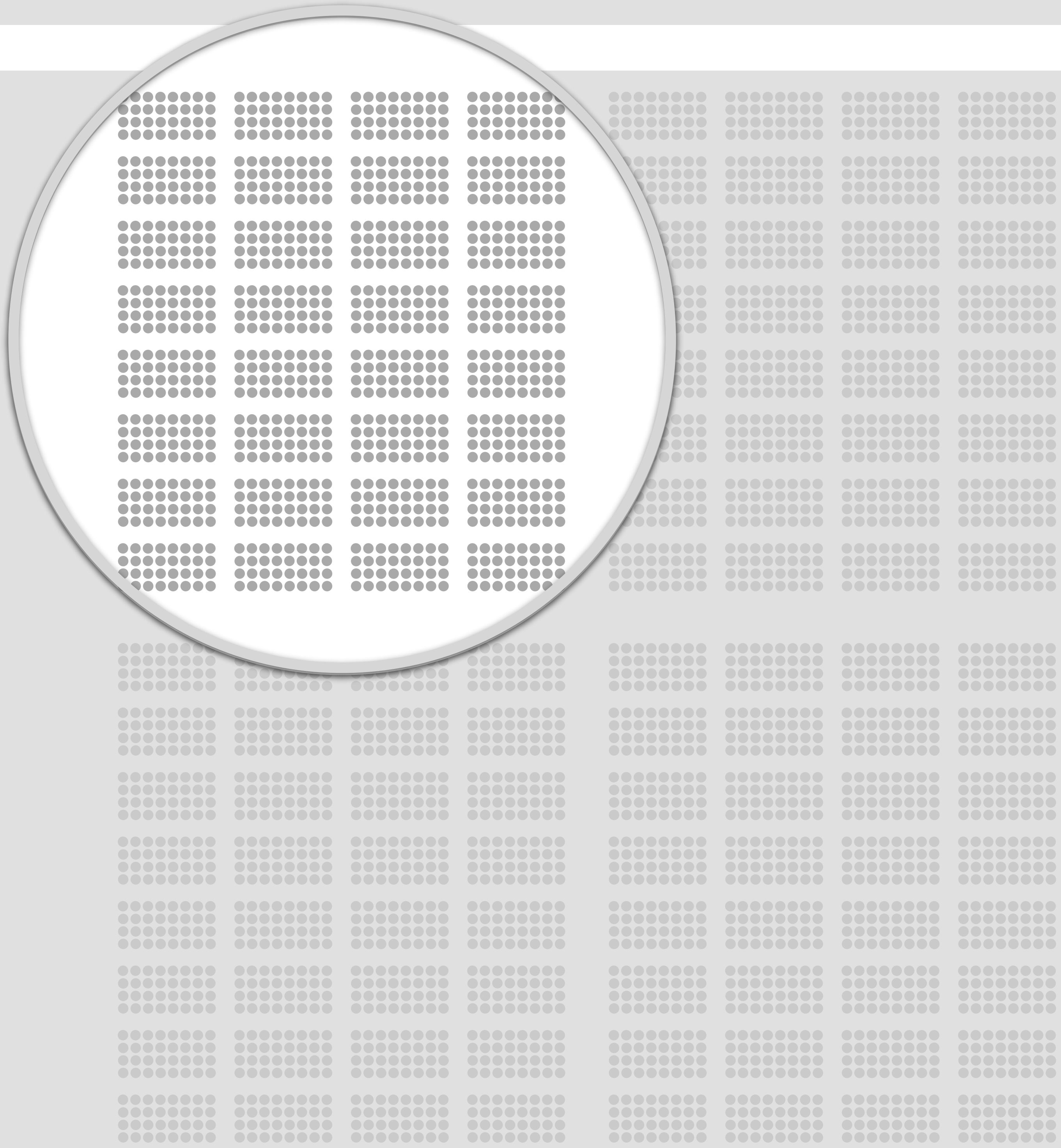


current threads

completed work

pmap

processors: 2 threads: 4 chunk size: 32



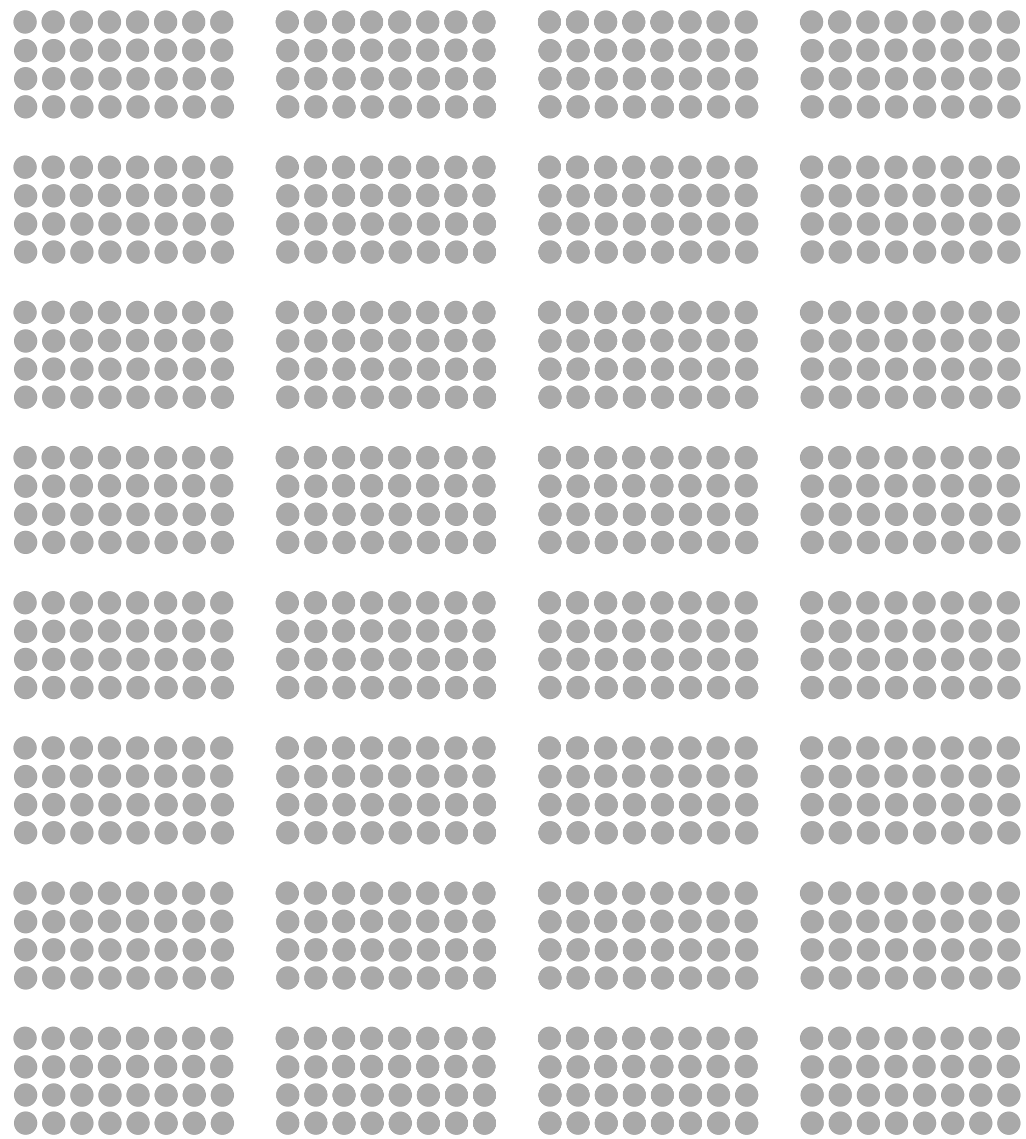
current threads

completed work

pmap

processors: 2 threads: 4 chunk size: 32

One strategy for handling situations where the consuming process cannot keep up with the producing process, hence underutilizing the processors, is to partition the input data into chunks and pass those to *pmap* instead of individual values.



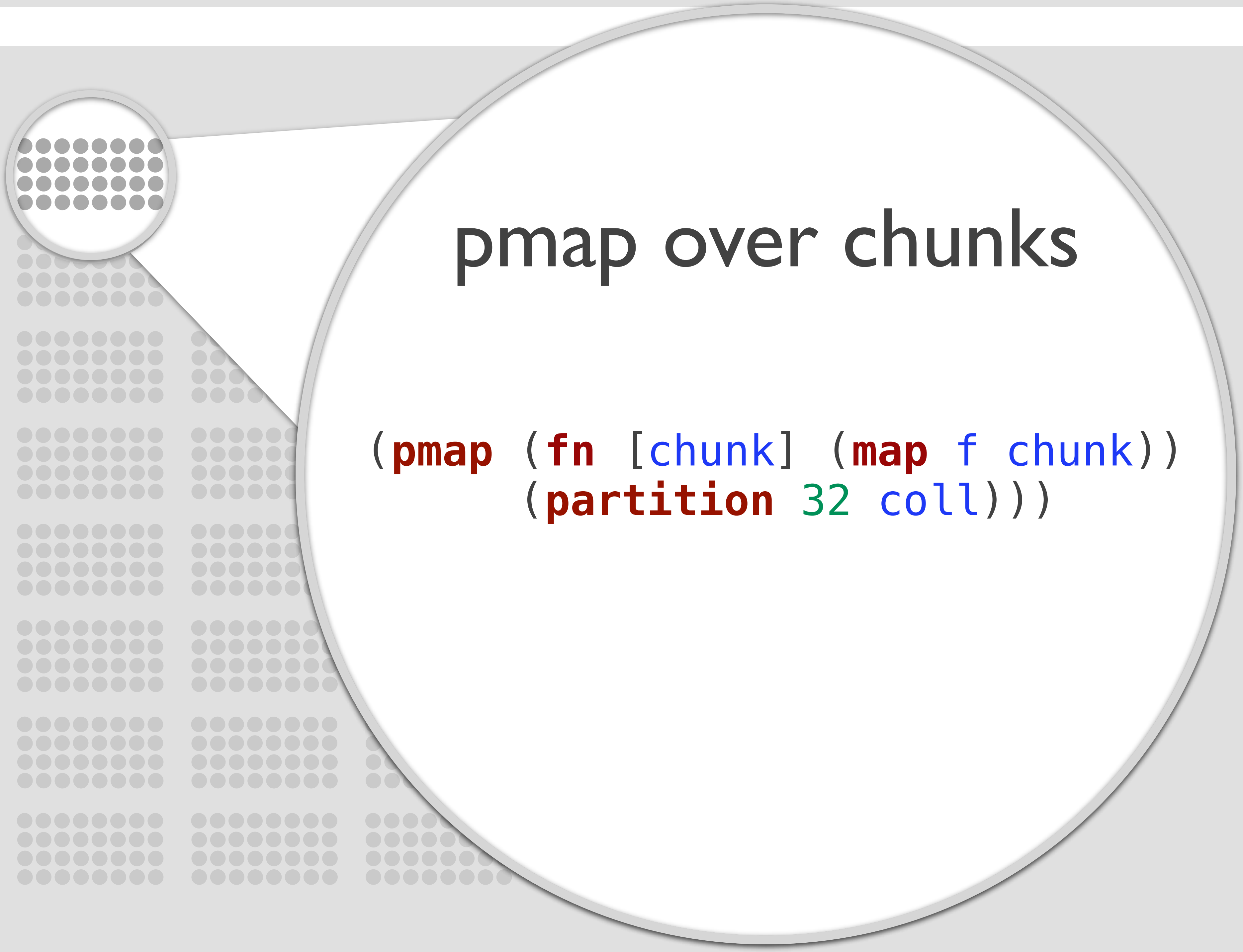
current threads

completed work

pmap

processors: 2 threads: 4 chunk size: 32

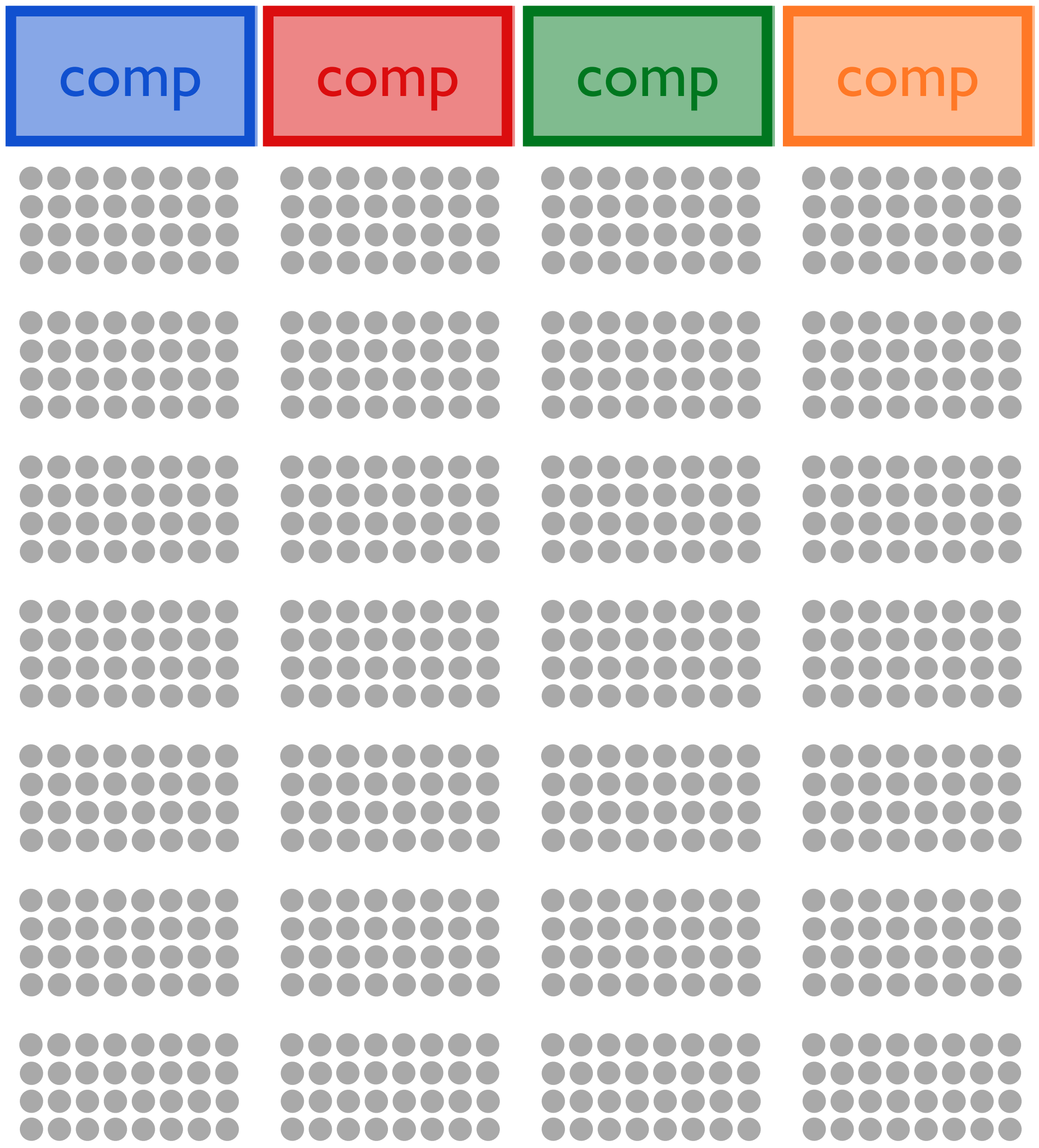
One strategy for handling situations where the consuming process cannot keep up with the producing process, hence underutilizing the processors, is to partition the input data into chunks and pass those to *pmap* instead of individual values.



current threads

completed work

One strategy for handling situations where the consuming process cannot keep up with the producing process, hence underutilizing the processors, is to partition the input data into chunks and pass those to *pmap* instead of individual values.



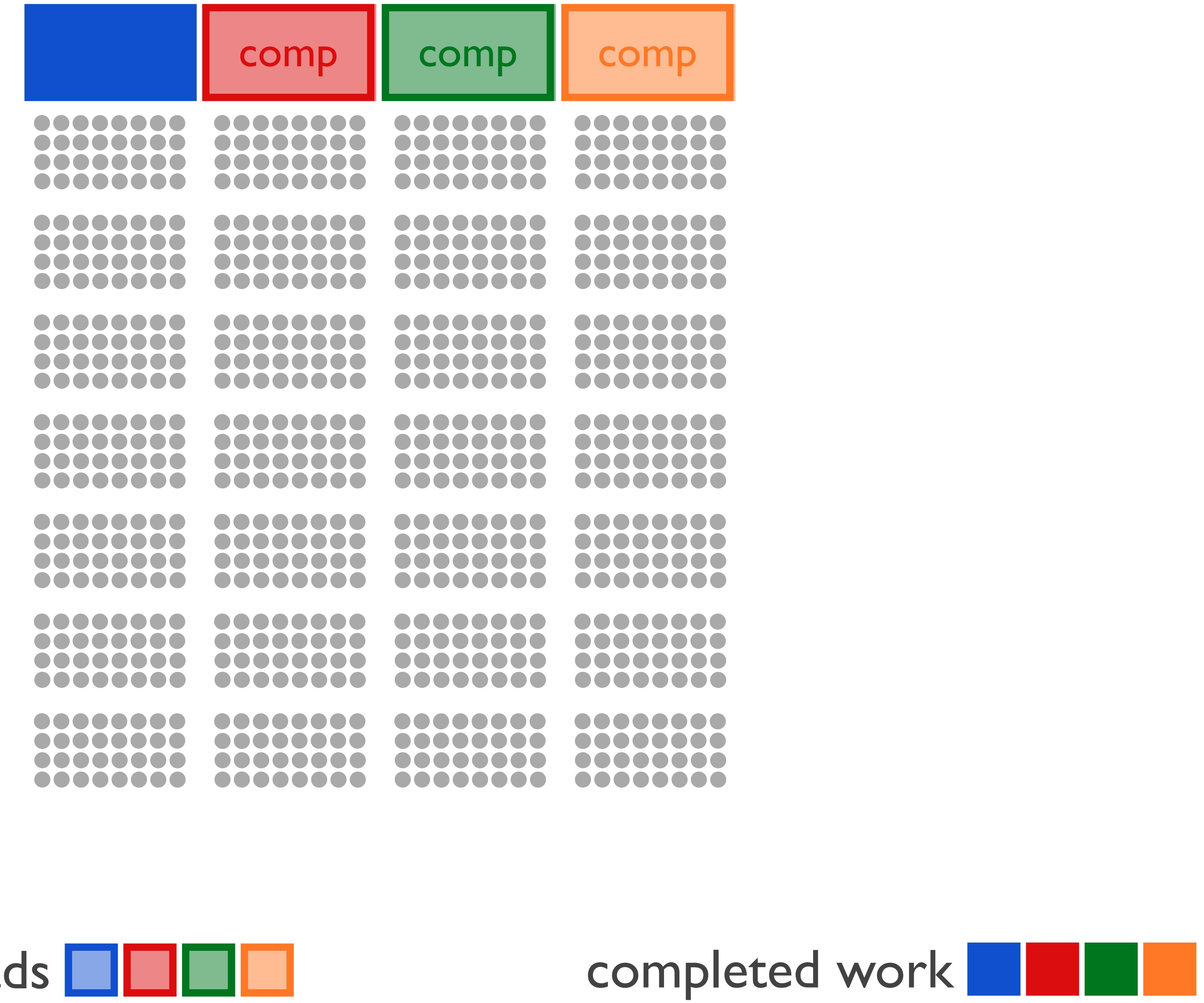
current threads

completed work

pmap

processors: 2 threads: 4 chunk size: 32

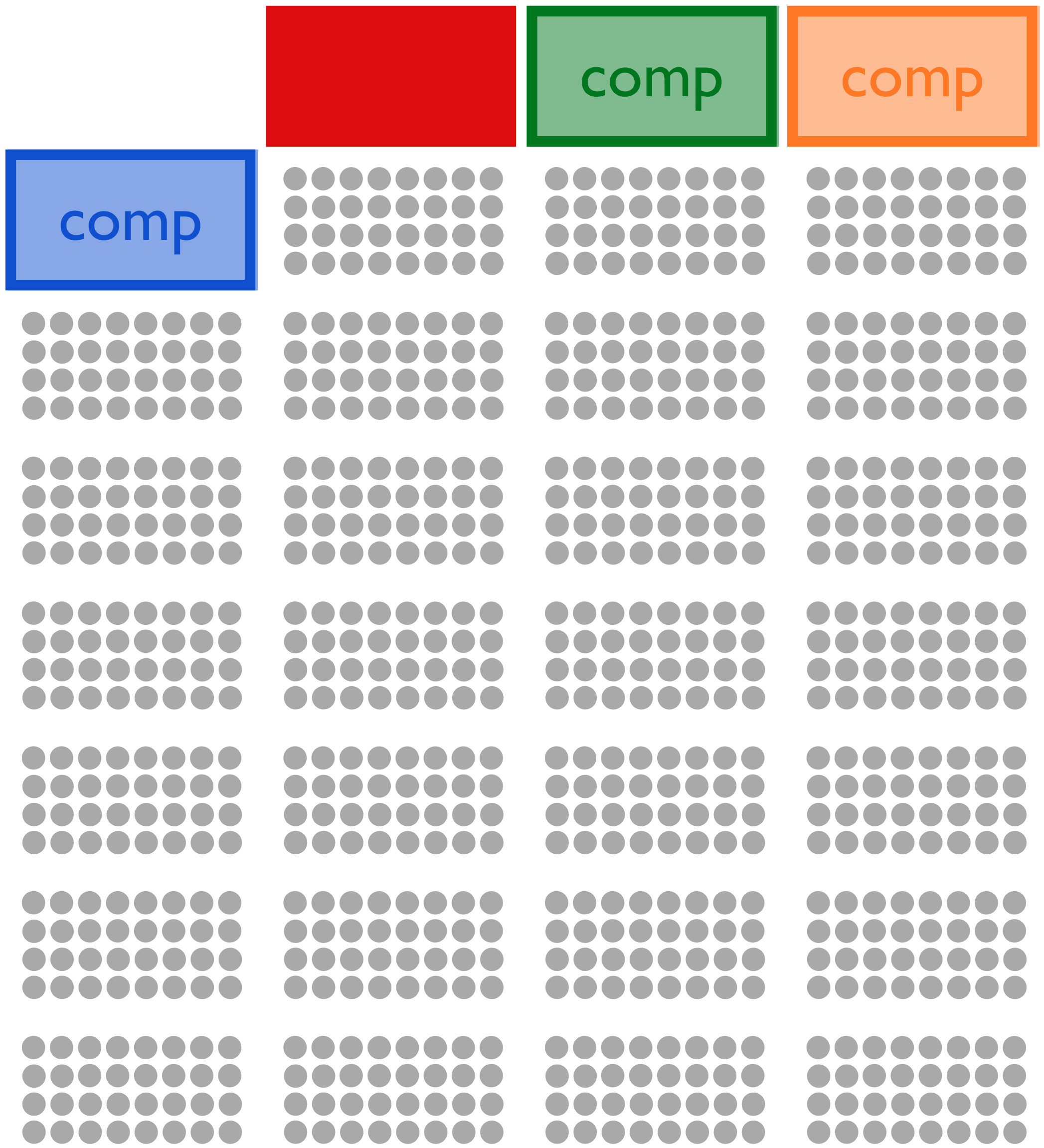
One strategy for handling situations where the consuming process cannot keep up with the producing process, hence underutilizing the processors, is to partition the input data into chunks and pass those to *pmap* instead of individual values.



pmap

processors: 2 threads: 4 chunk size: 32

One strategy for handling situations where the consuming process cannot keep up with the producing process, hence underutilizing the processors, is to partition the input data into chunks and pass those to *pmap* instead of individual values.



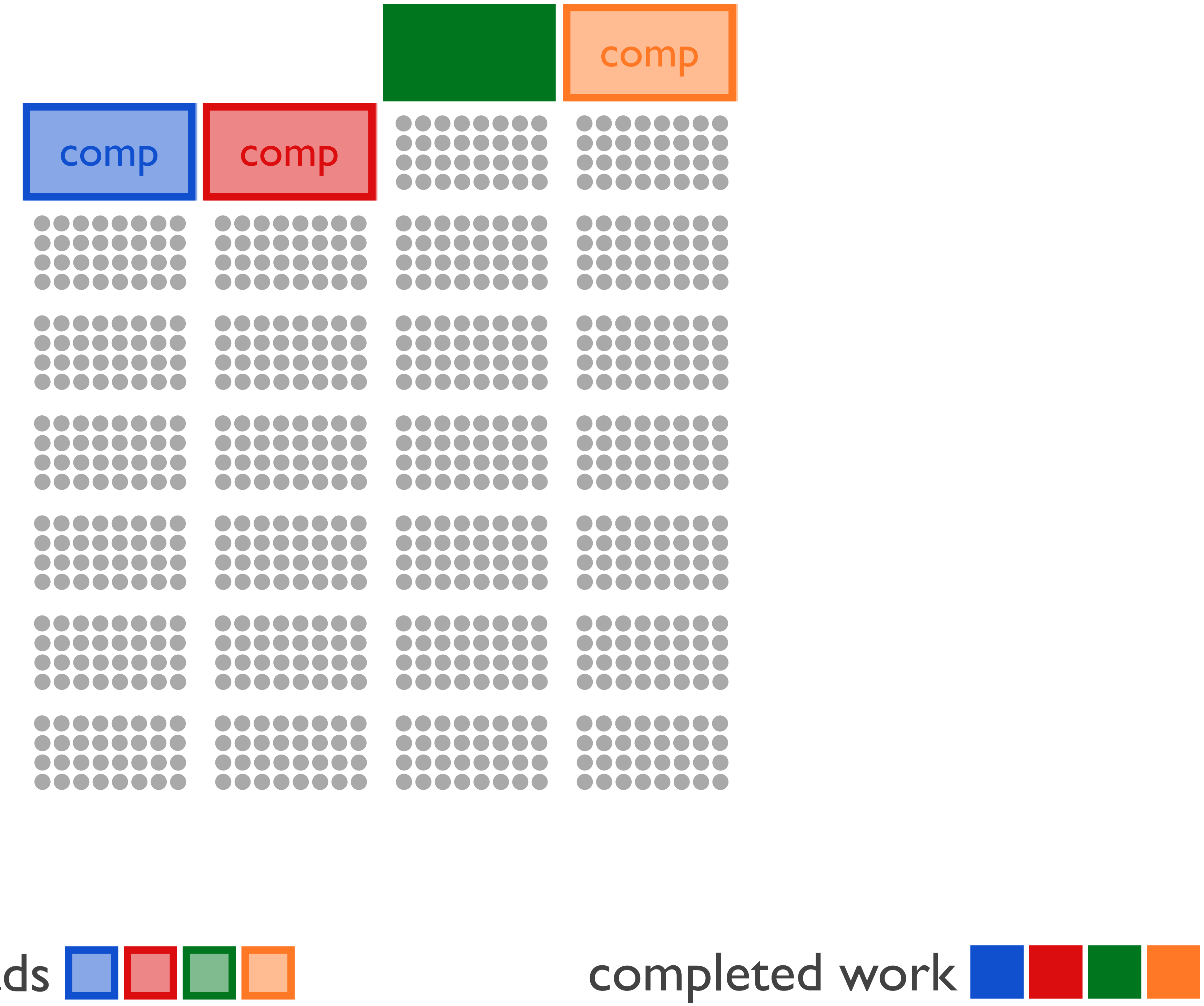
current threads

completed work

pmap

processors: 2 threads: 4 chunk size: 32

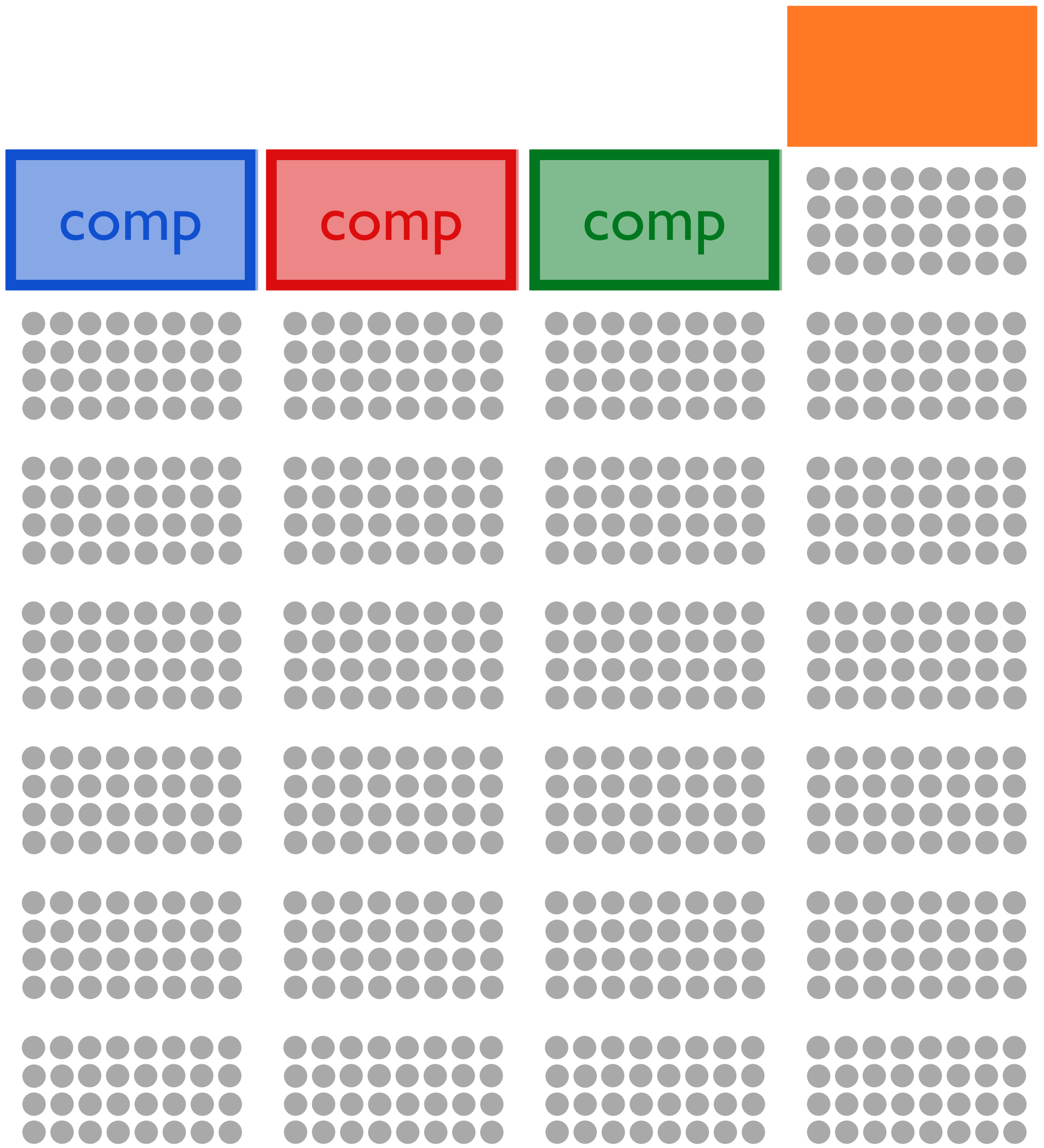
One strategy for handling situations where the consuming process cannot keep up with the producing process, hence underutilizing the processors, is to partition the input data into chunks and pass those to *pmap* instead of individual values.



pmap

processors: 2 threads: 4 chunk size: 32

One strategy for handling situations where the consuming process cannot keep up with the producing process, hence underutilizing the processors, is to partition the input data into chunks and pass those to *pmap* instead of individual values.

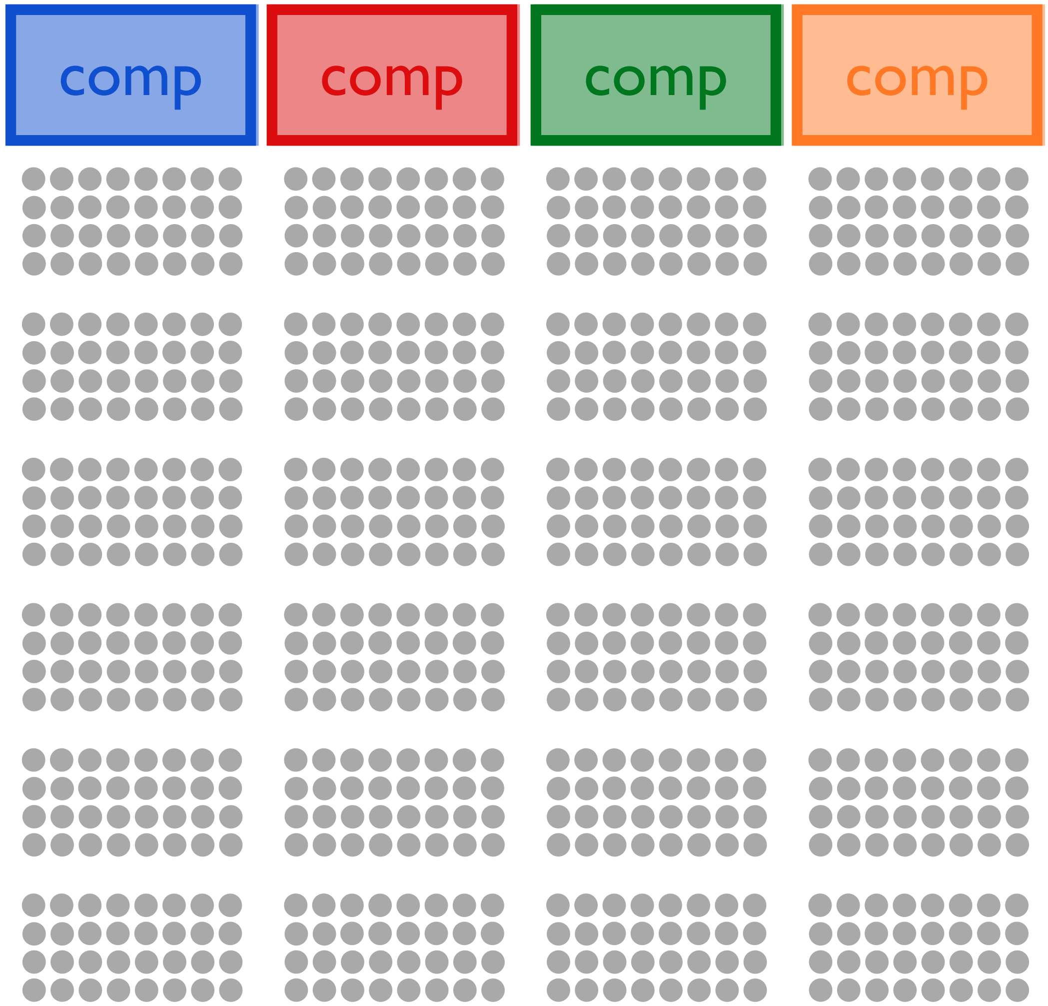


current threads

completed work

processors: 2 threads: 4 chunk size: 32

One strategy for handling situations where the consuming process cannot keep up with the producing process, hence underutilizing the processors, is to partition the input data into chunks and pass those to *pmap* instead of individual values.



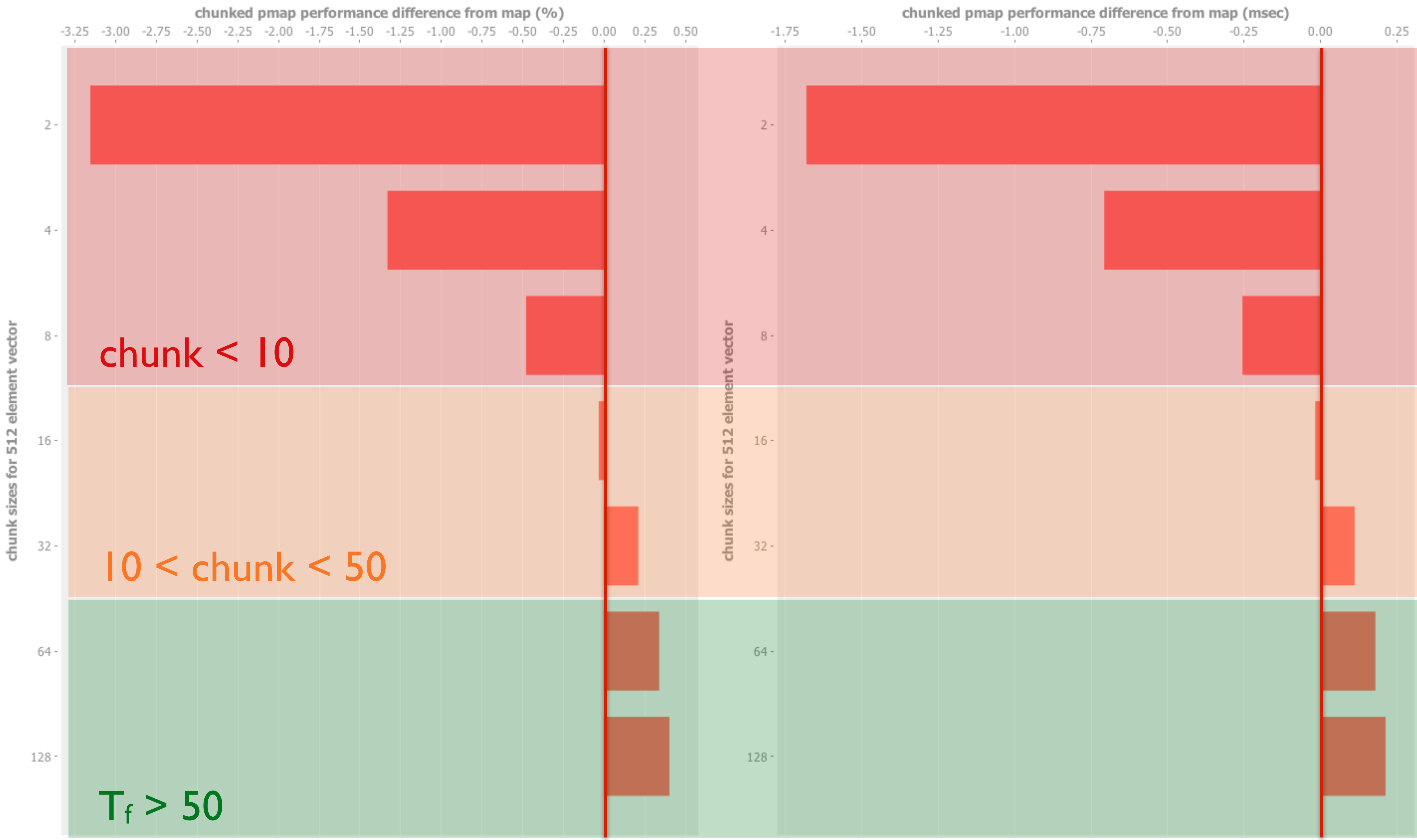
current threads

completed work

chunked pmap

compared to map (2 cores)

* results are the median of 500 samples, where a test function (duration < 0.005 msec) was pmapped over chunked vectors of 512 values



fork-join parallelism

divide and conquer

dequeue

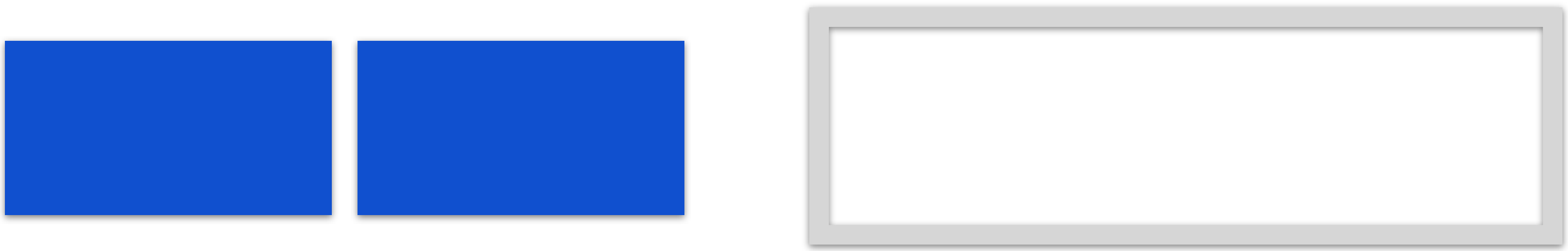
double-ended queue

dequeue

double-ended queue

A dequeue (pronounced "deck") is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue



A dequeue (pronounced “deck”) is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue

push

A dequeue (pronounced “deck”) is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

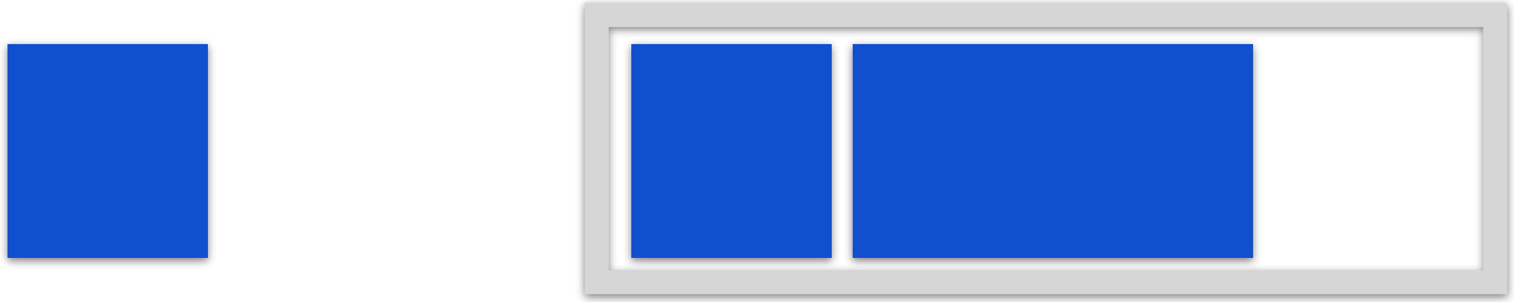
dequeue

A dequeue (pronounced “deck”) is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue

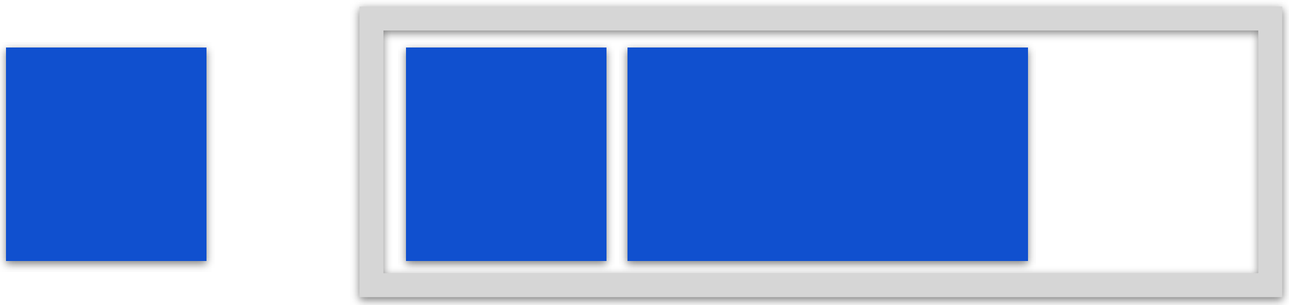
A dequeue (pronounced “deck”) is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue



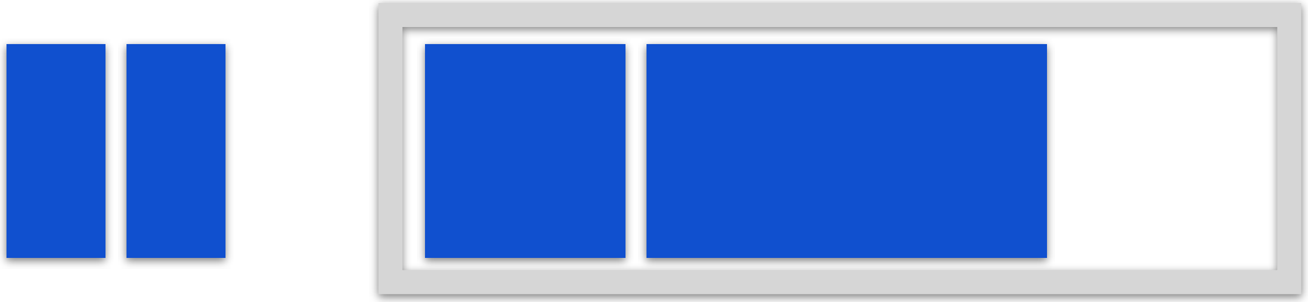
A dequeue (pronounced “deck”) is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue

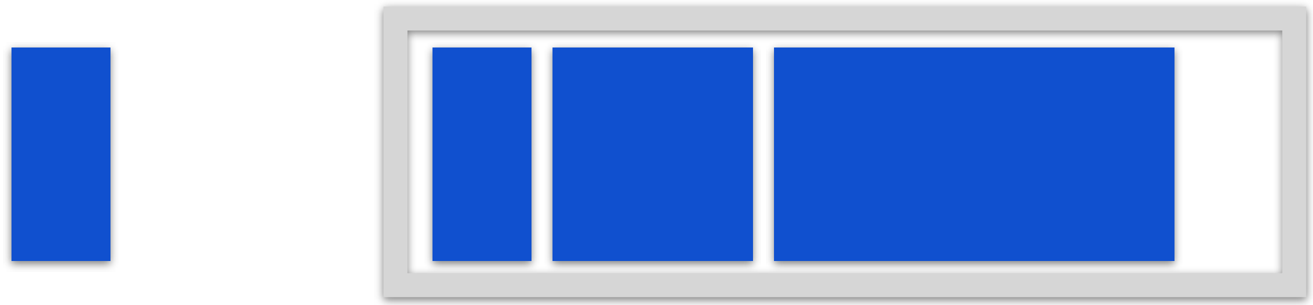


A dequeue (pronounced "deck") is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue



A dequeue (pronounced "deck") is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

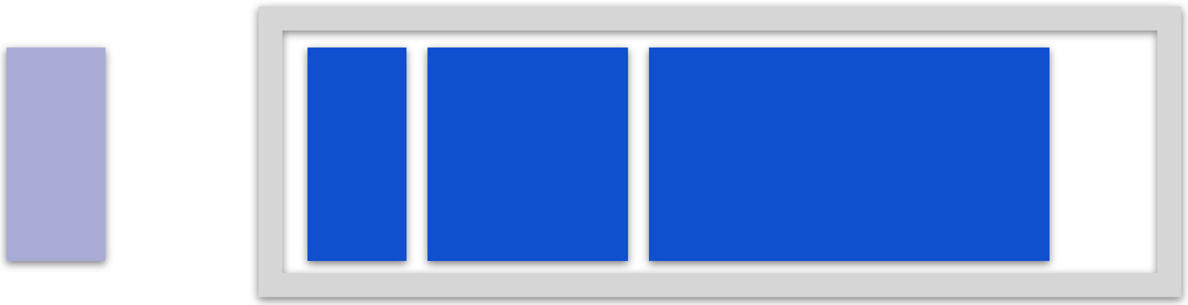


A dequeue (pronounced “deck”) is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue

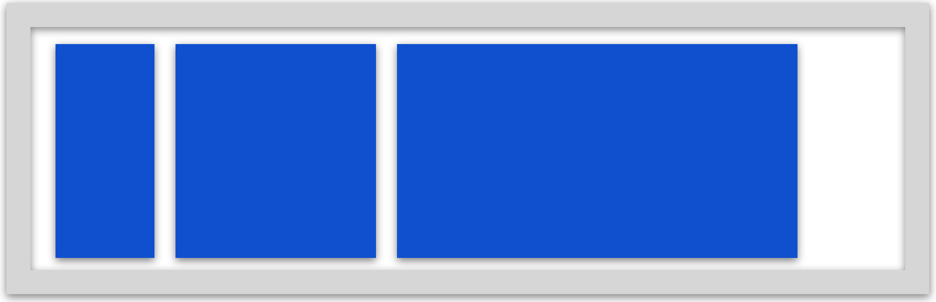
A dequeue (pronounced "deck") is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue



A dequeue (pronounced “deck”) is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue

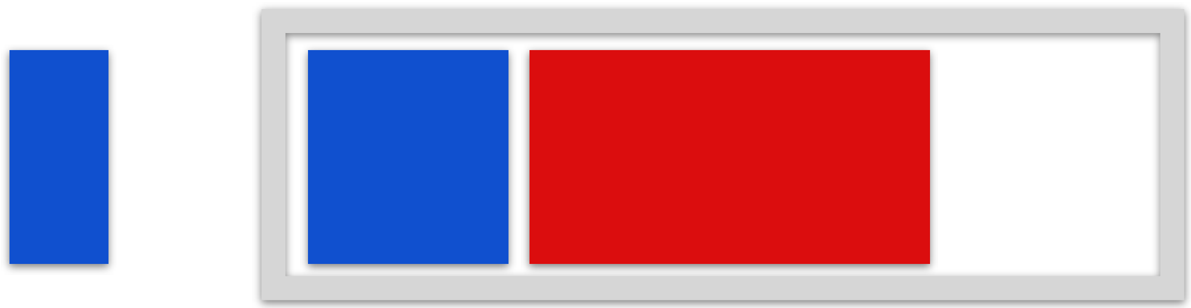


A dequeue (pronounced “deck”) is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue

A dequeue (pronounced "deck") is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue



A dequeue (pronounced “deck”) is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

dequeue

take

A dequeue (pronounced “deck”) is a double-ended queue, where values can be *pushed* and *popped* off the front or *taken* from the back. The Fork-Join algorithm systematically divides a job into tasks that are pushed onto the dequeue, resulting in the largest tasks being located at the back, which in turn improves the efficiency of its work-stealing algorithm.

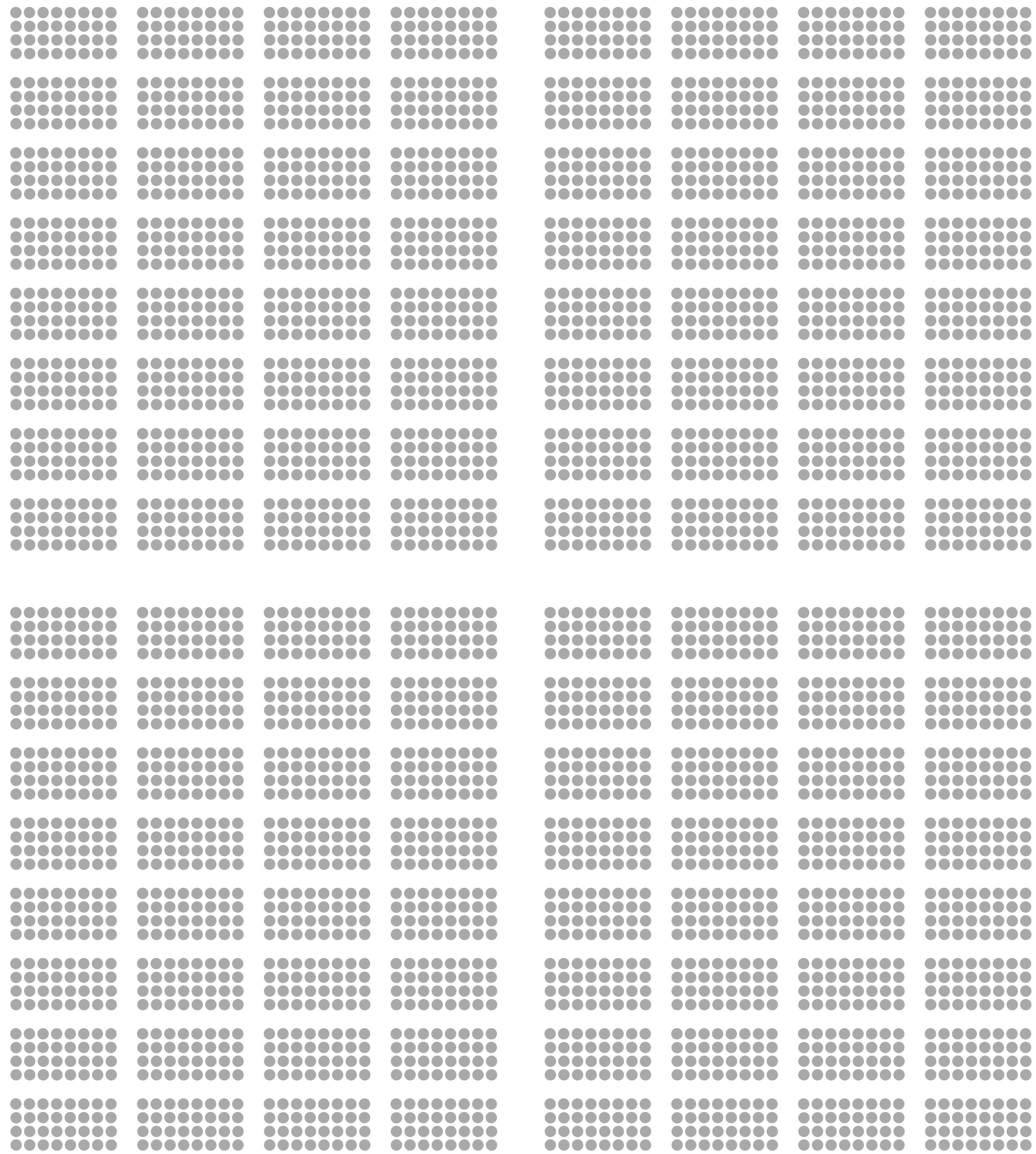
fork-join

basic algorithm

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

Fork-Join is a divide-and-conquer algorithm that iteratively divides a job into smaller and smaller *tasks*, placing them on a dequeue, until the size of the current *task* is below a configured threshold.

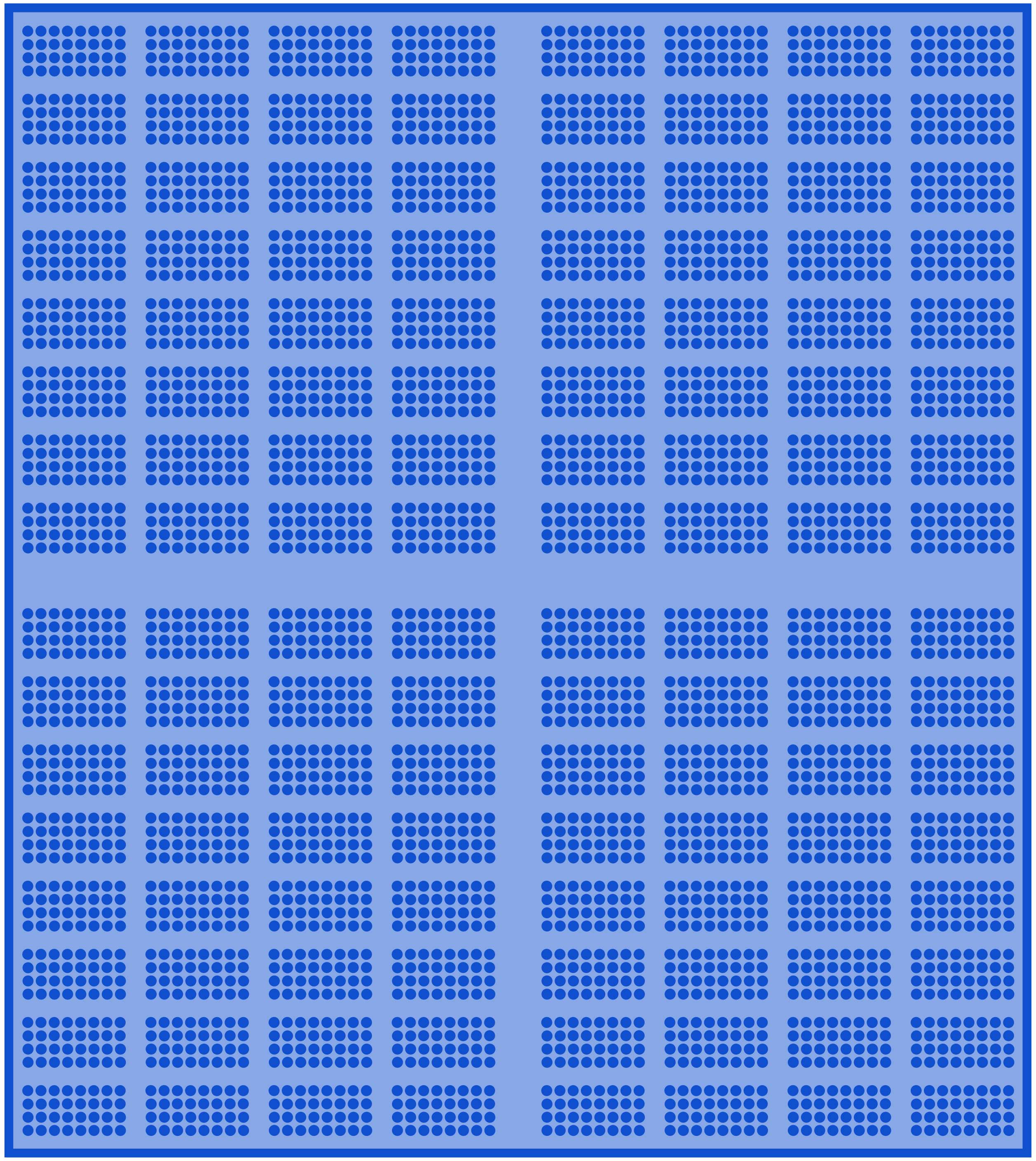


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

Fork-Join is a divide-and-conquer algorithm that iteratively divides a job into smaller and smaller *tasks*, placing them on a dequeue, until the size of the current *task* is below a configured threshold.

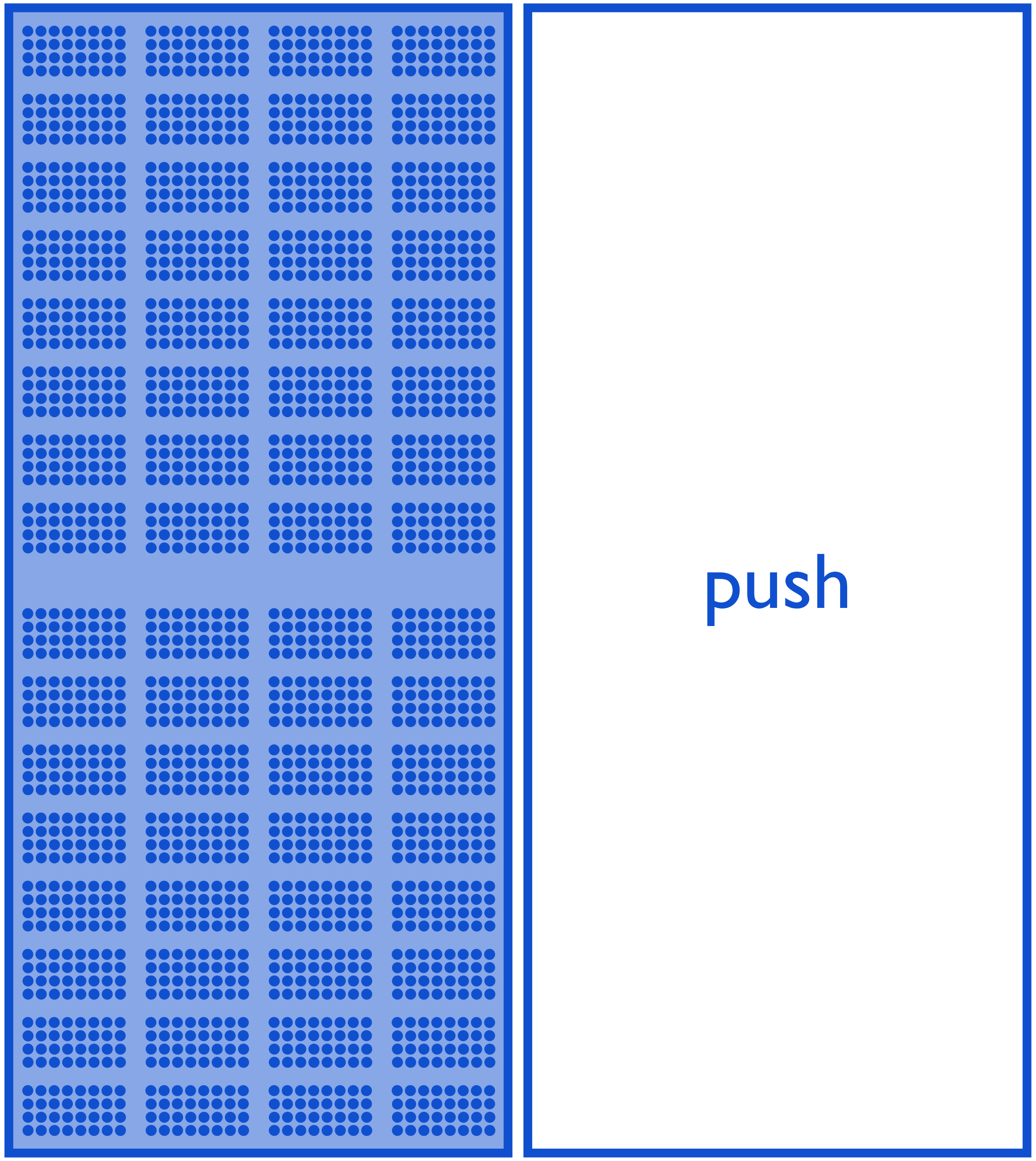


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

Fork-Join is a divide-and-conquer algorithm that iteratively divides a job into smaller and smaller *tasks*, placing them on a dequeue, until the size of the current *task* is below a configured threshold.

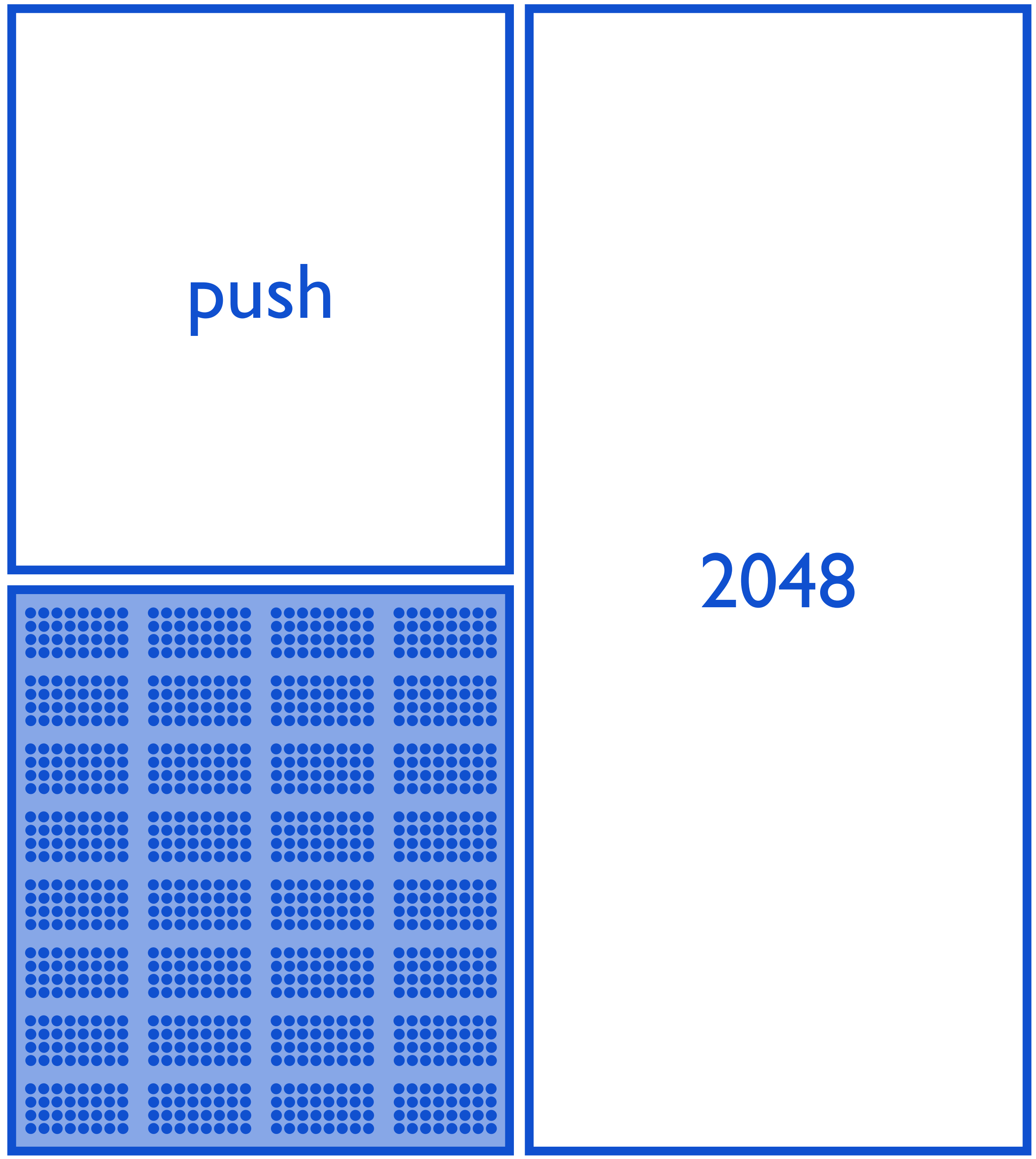


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

Fork-Join is a divide-and-conquer algorithm that iteratively divides a job into smaller and smaller *tasks*, placing them on a dequeue, until the size of the current *task* is below a configured threshold.

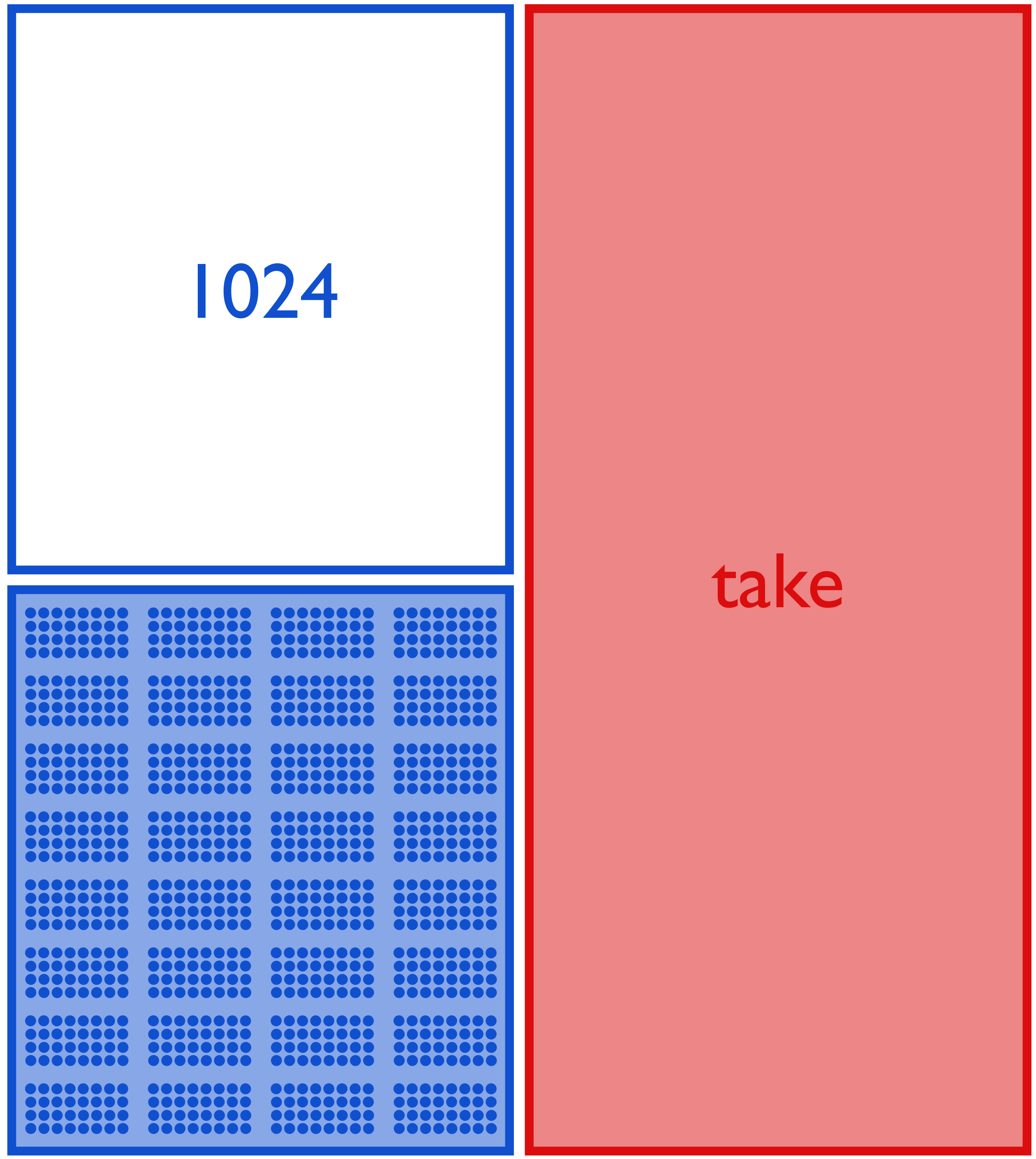


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

As one worker is processing its tasks, another worker can steal a task from the back of its dequeue.

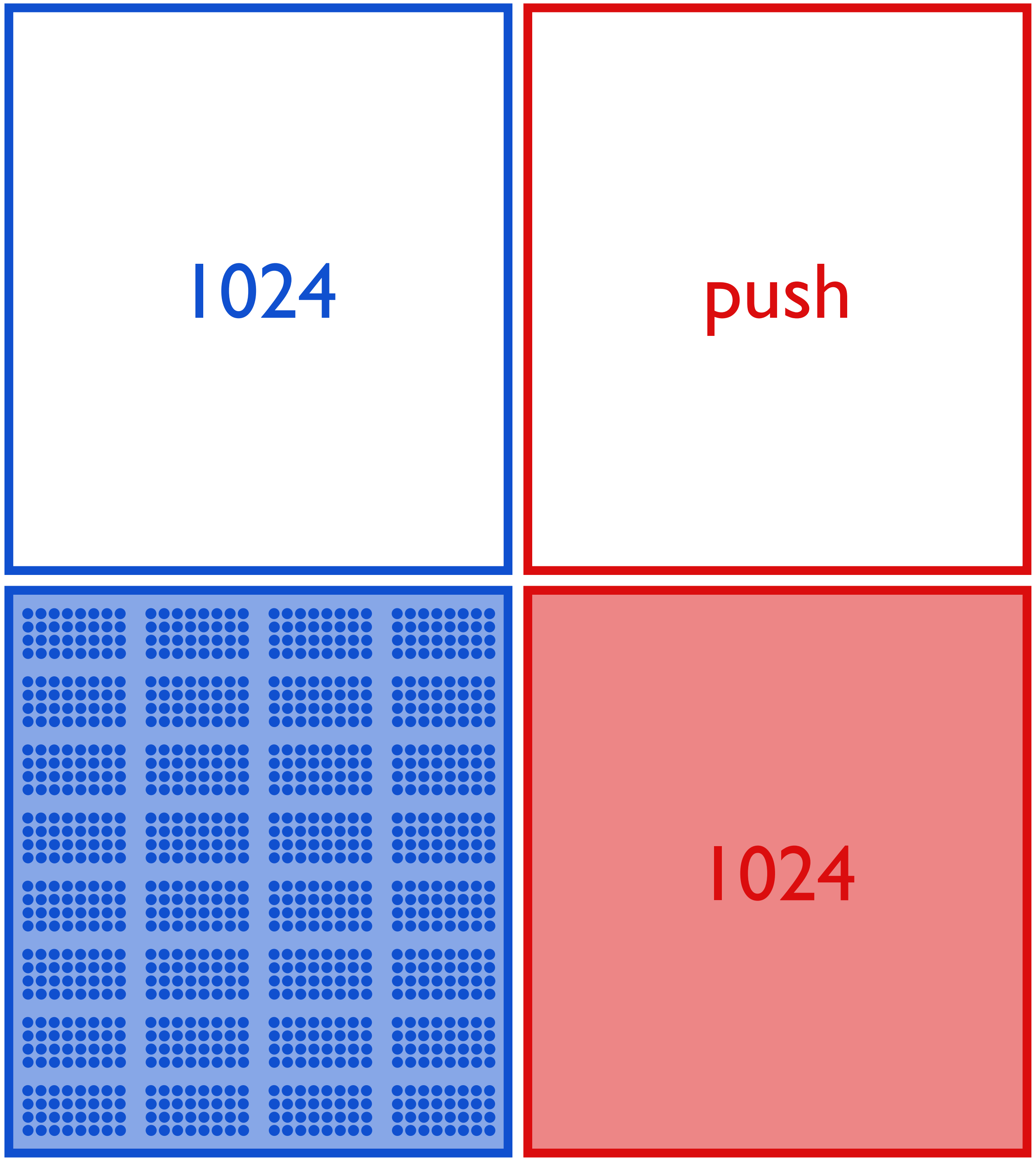


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

As one worker is processing its tasks, another worker can steal a task from the back of its dequeue.

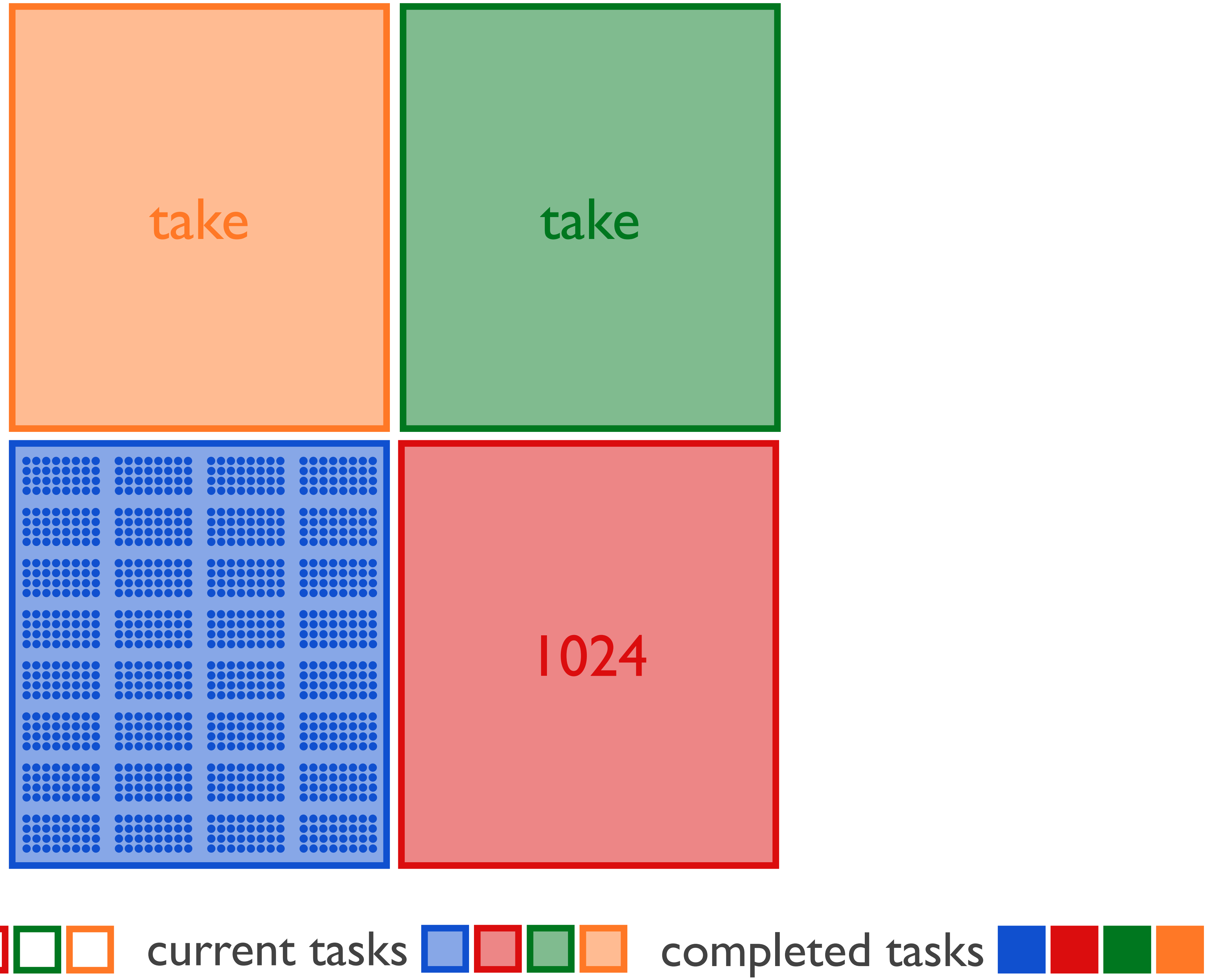


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

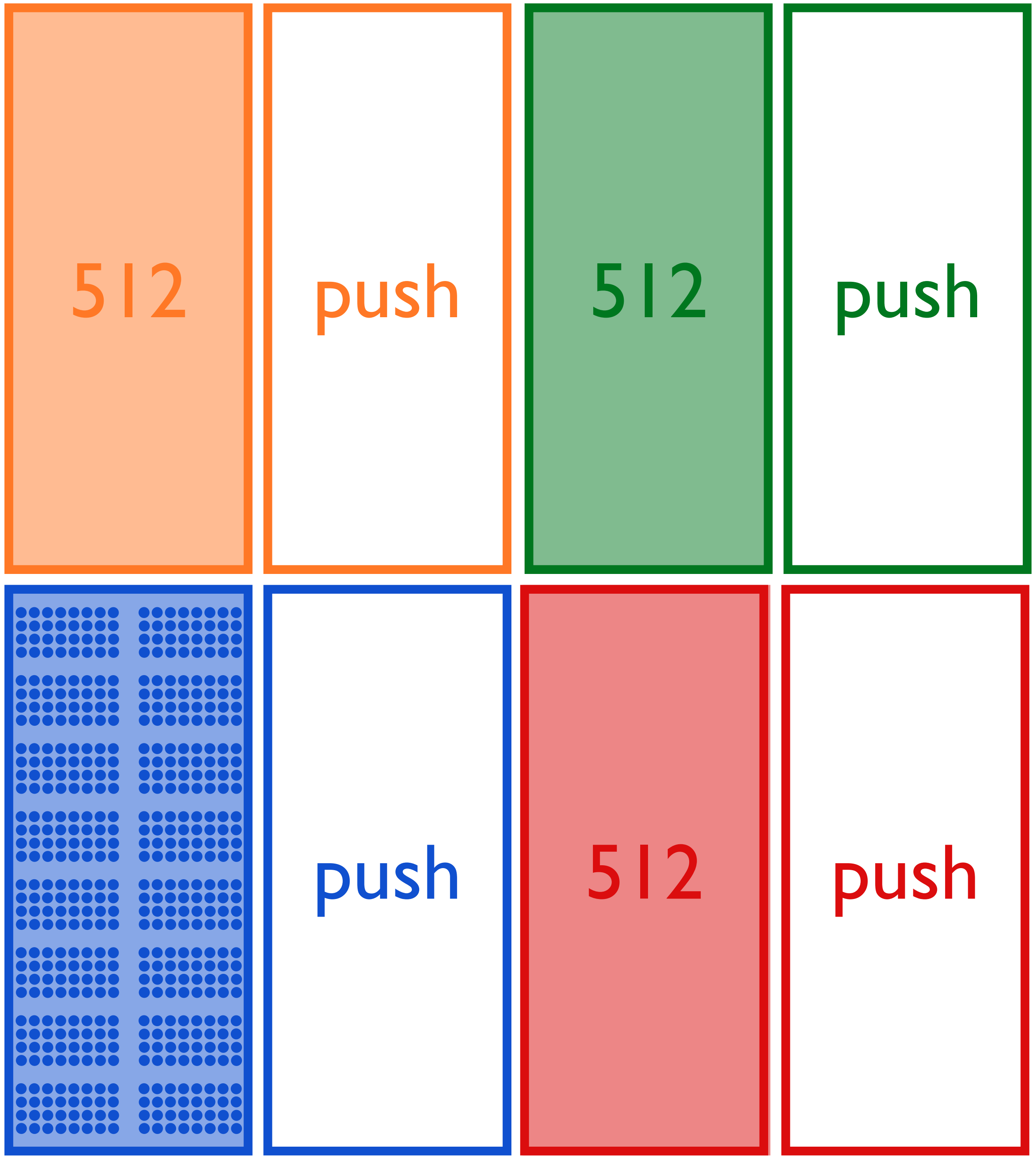
As one worker is processing its tasks, another worker can steal a task from the back of its dequeue.



fork-join

workers: 4 branching factor: 2 sequential threshold: 256

As one worker is processing its tasks, another worker can steal a task from the back of its dequeue.

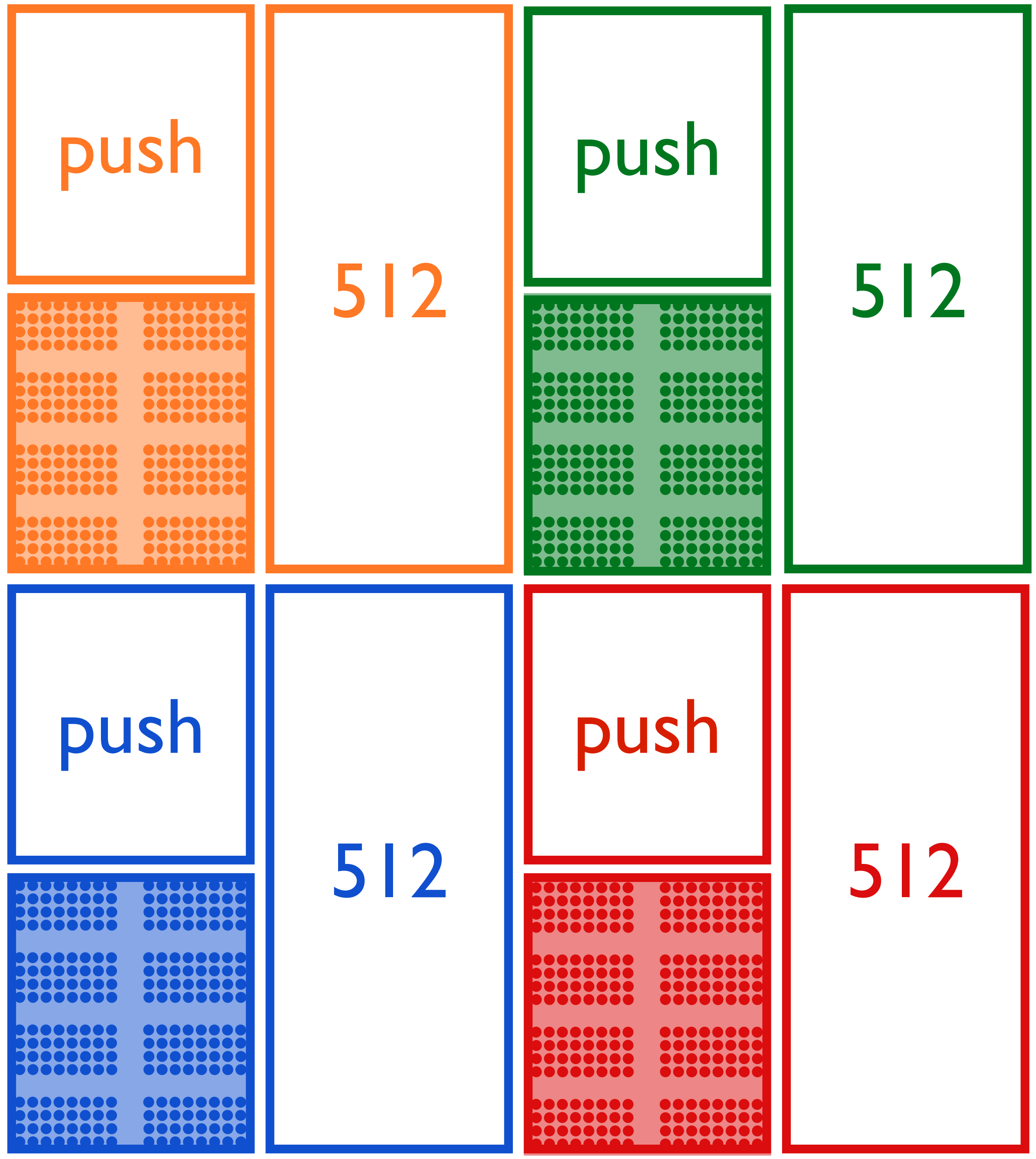


worker deques current tasks completed tasks

fork-join

workers: 4   branching factor: 2   sequential threshold: 256

As one worker is processing its tasks, another worker can steal a task from the back of its dequeue.

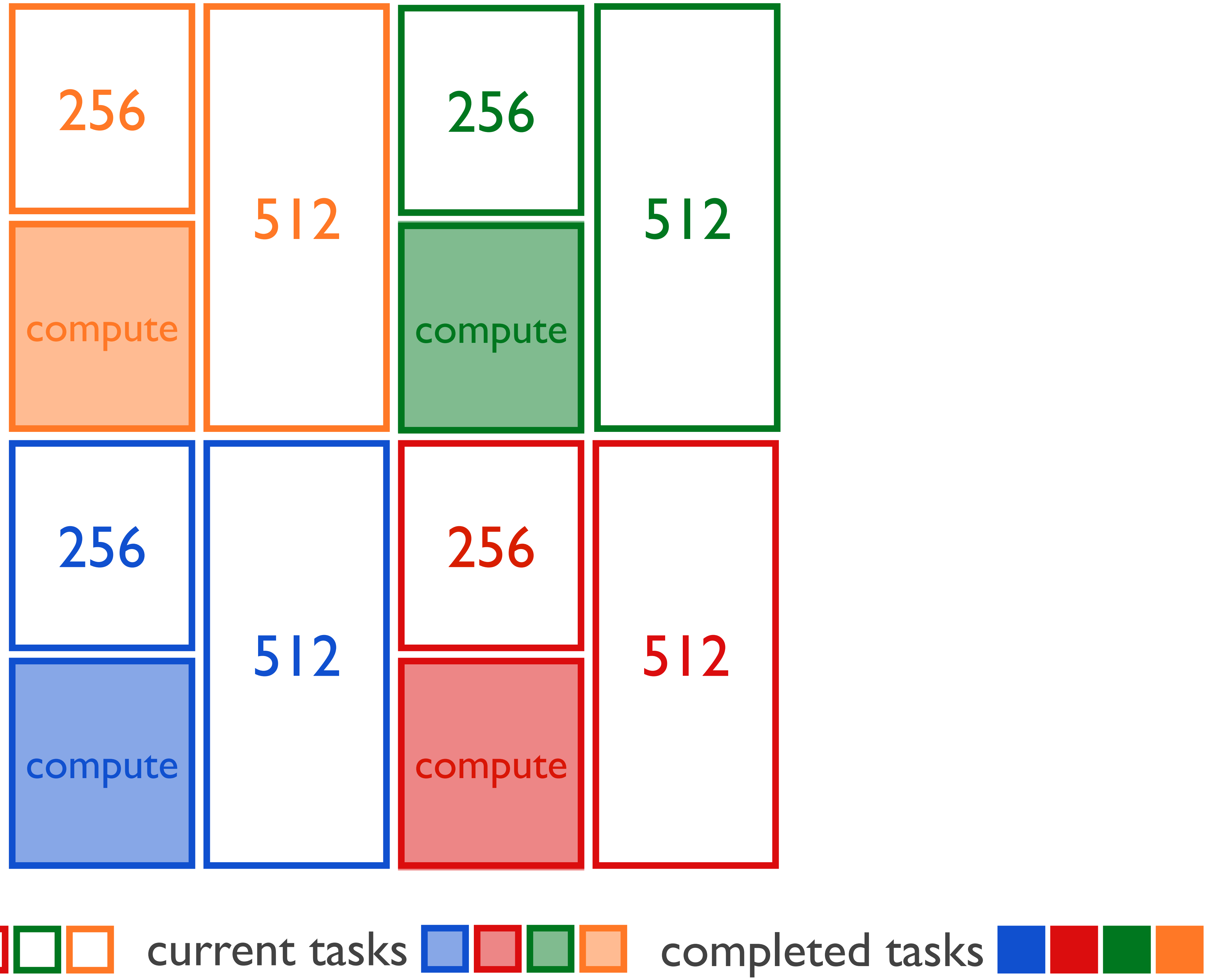


worker deques   current tasks   completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

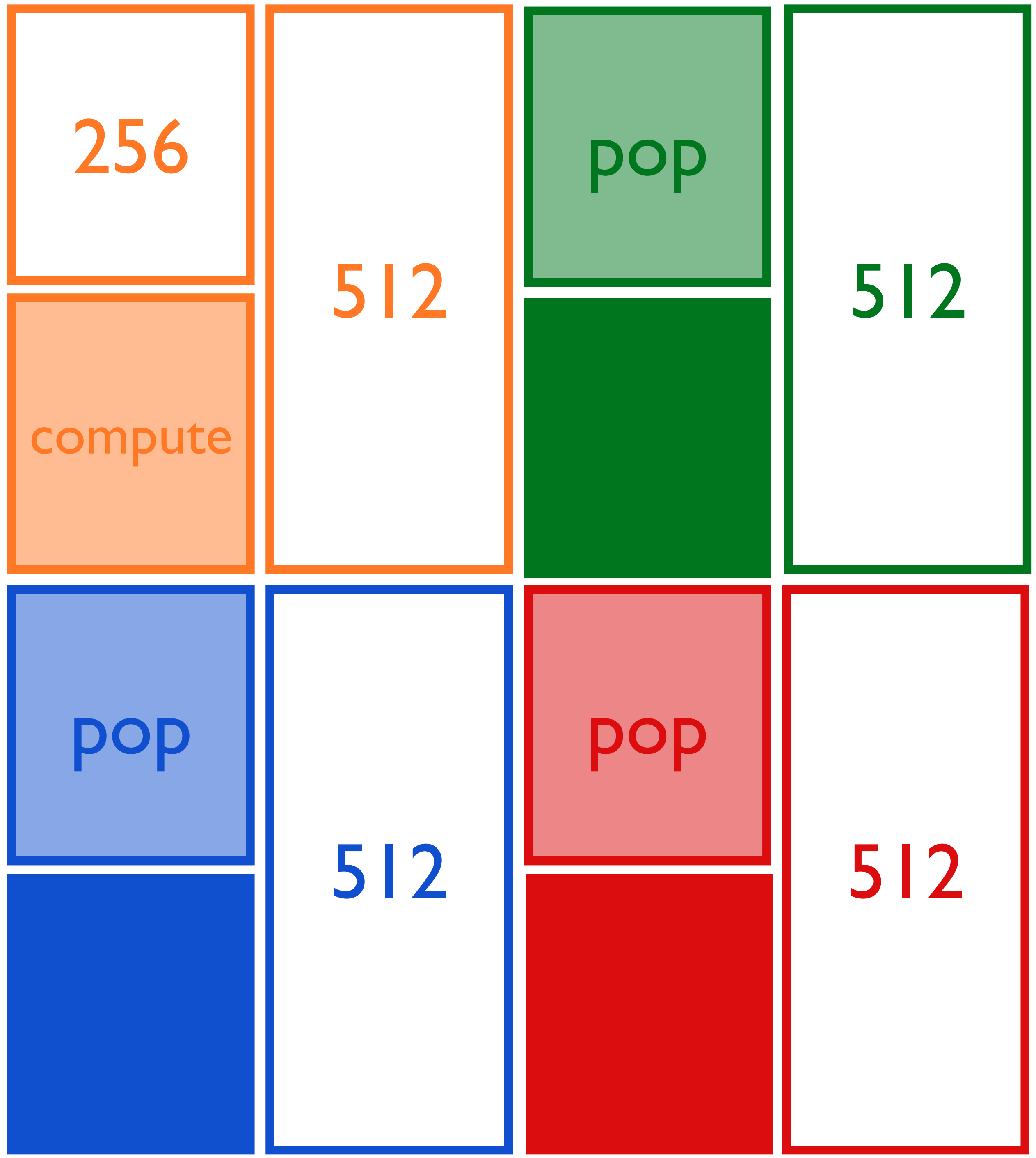
Once each *worker's* current *task* is smaller than the configured threshold, it will begin the intended computation.



fork-join

workers: 4 branching factor: 2 sequential threshold: 256

Once the computation is completed for the current *task*, the *worker* will *pop* another *task* off of its *deque*...

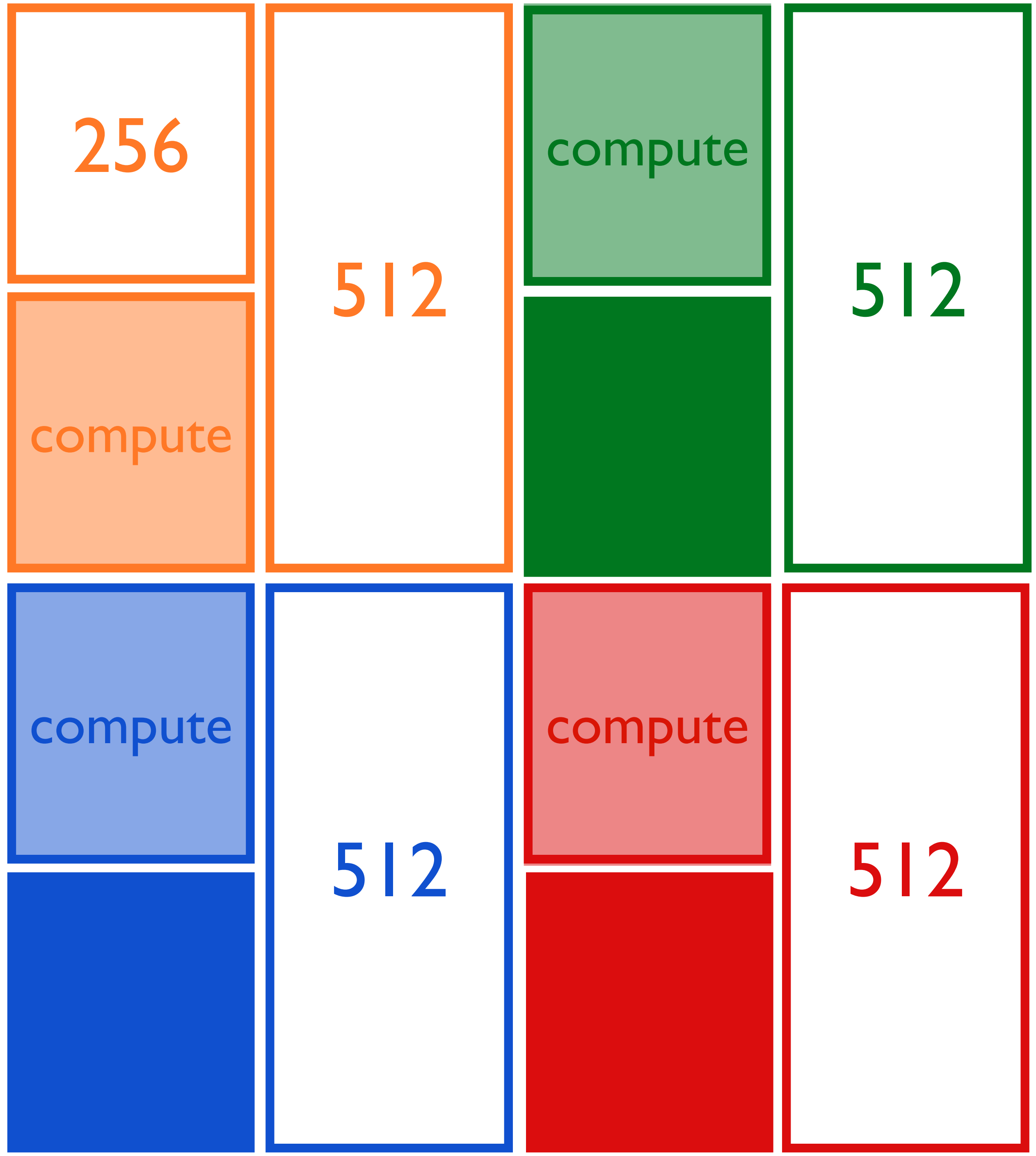


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

... and perform the computation on it.

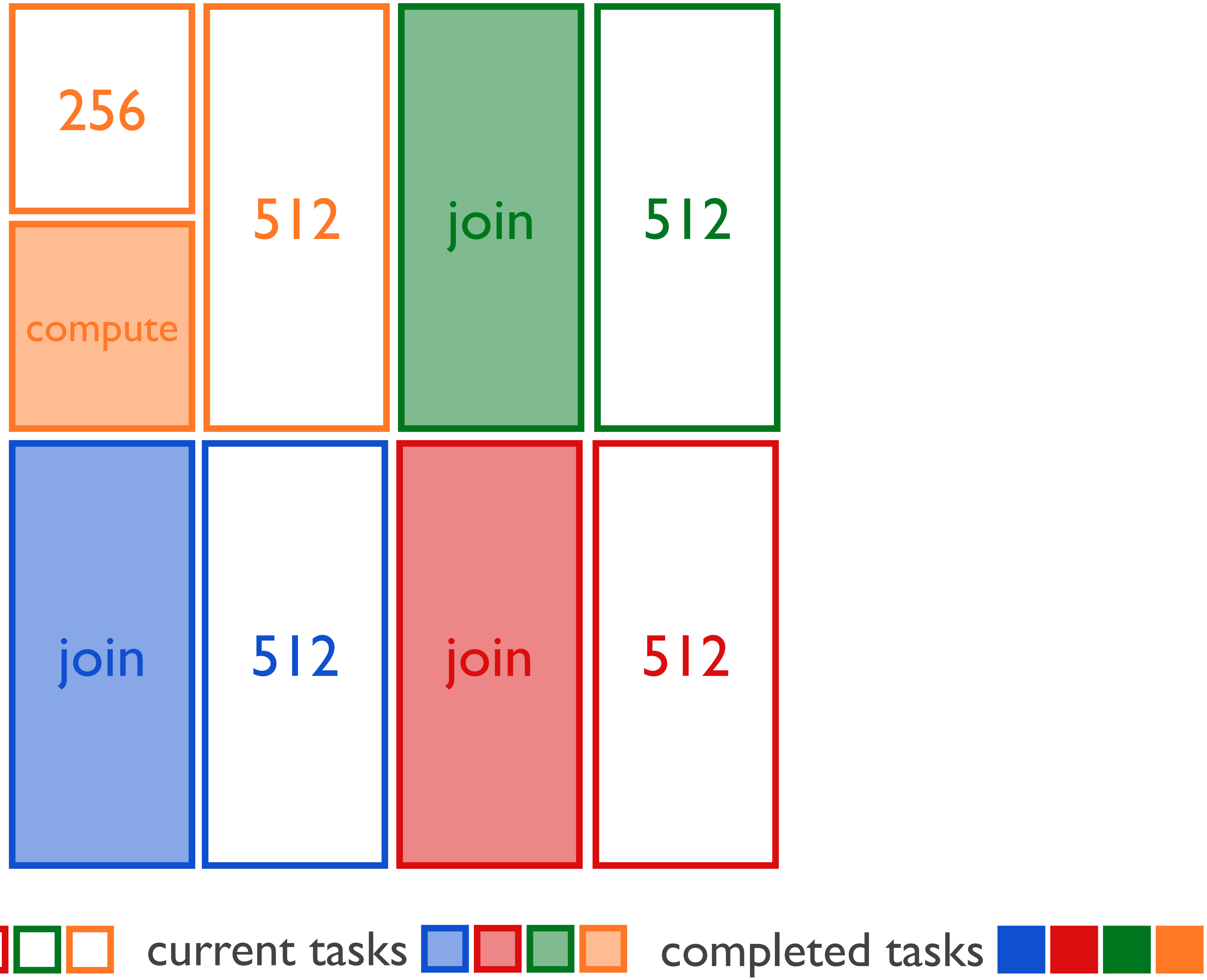


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

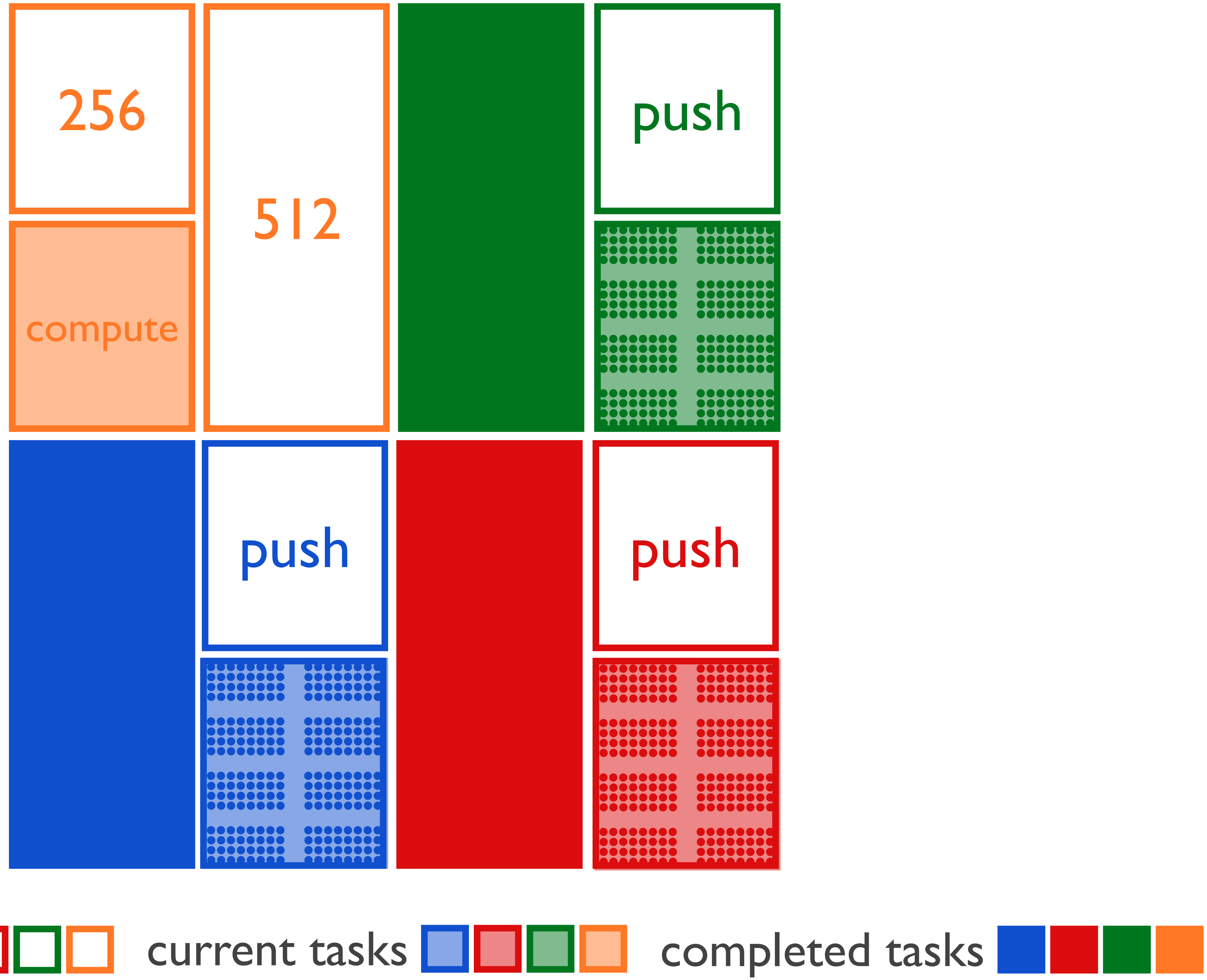
After the computation has been completed for at least two tasks, the results can be combined with a *join* function.



fork-join

workers: 4 branching factor: 2 sequential threshold: 256

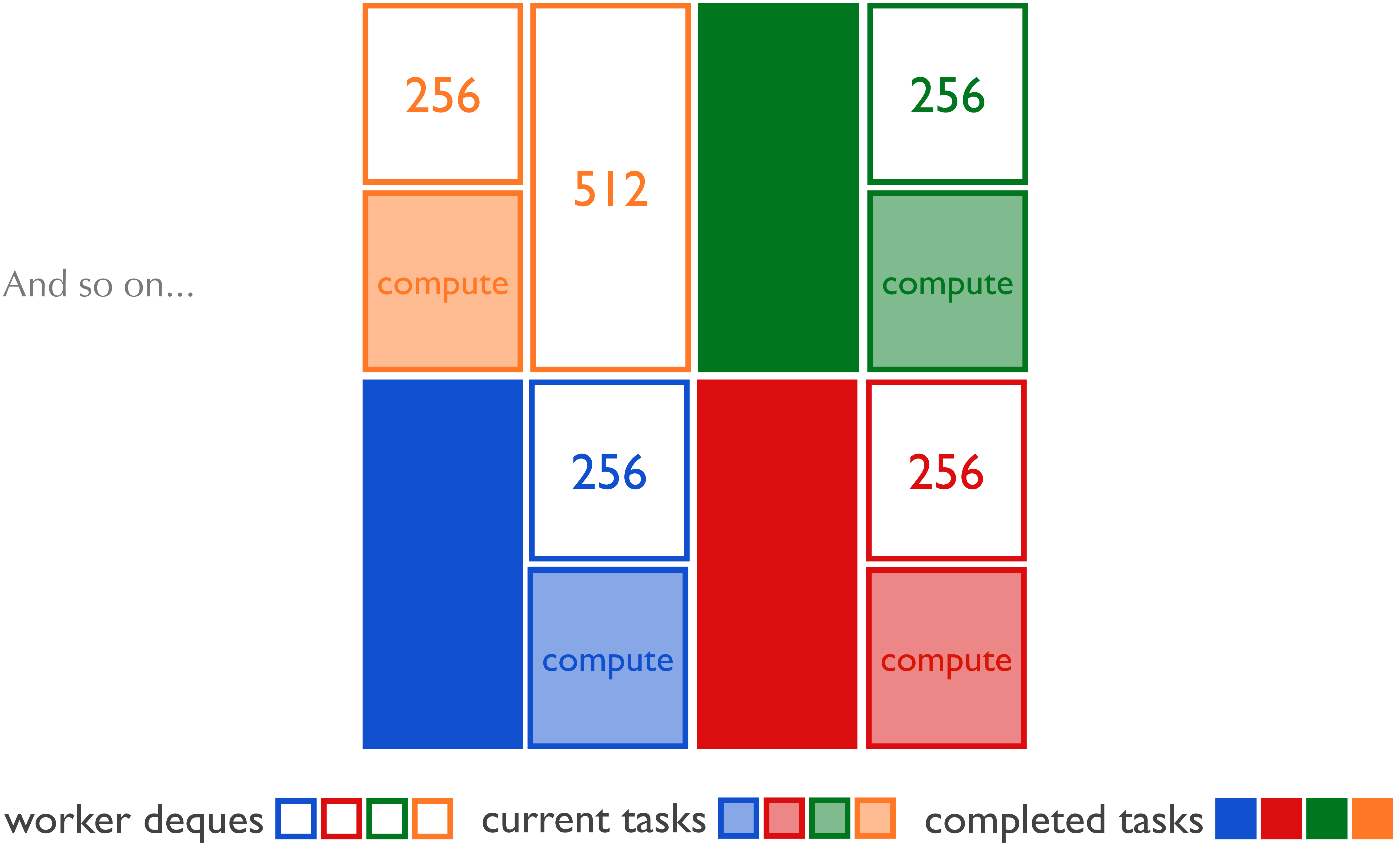
And so on...



fork-join

workers: 4 branching factor: 2 sequential threshold: 256

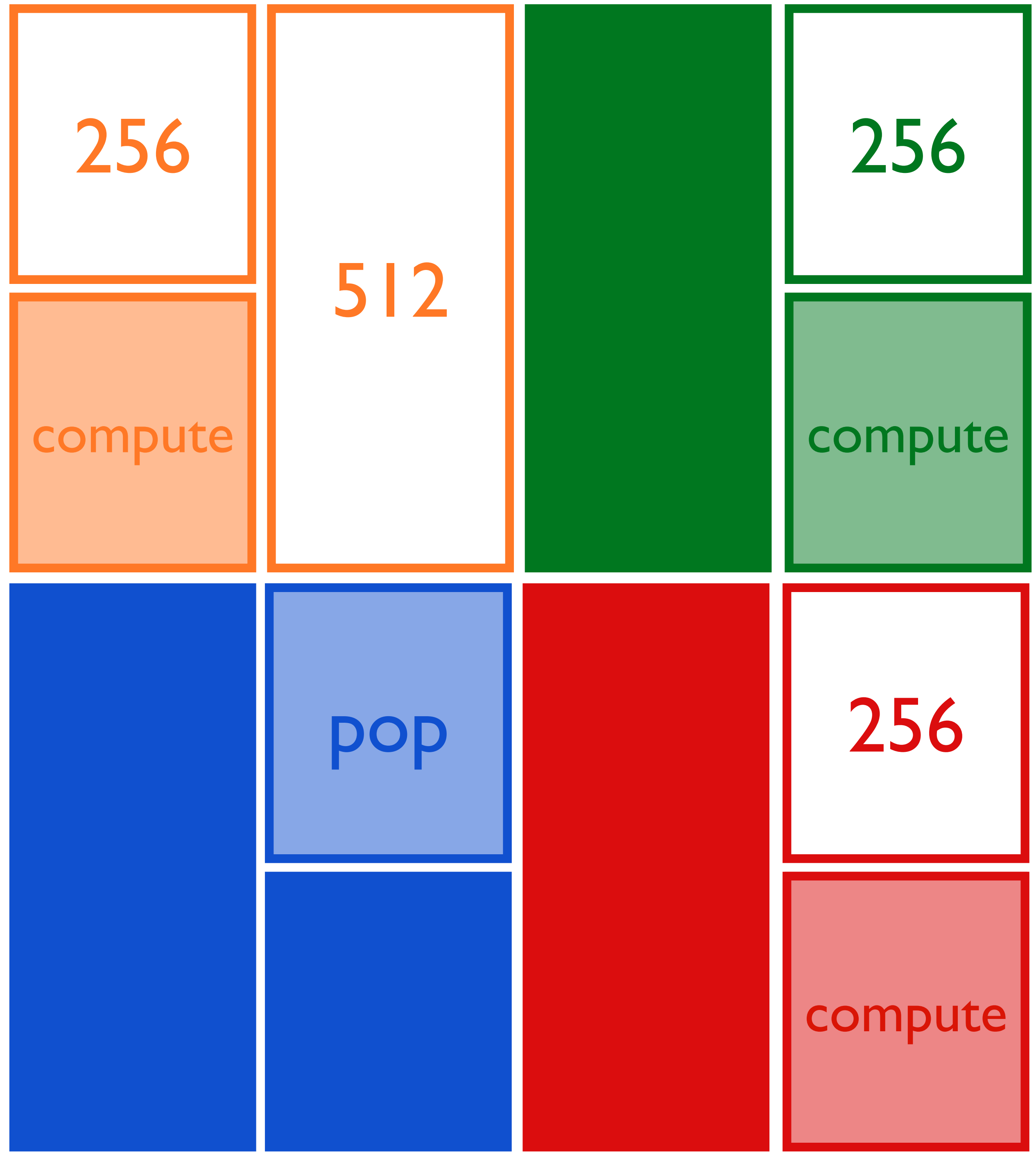
And so on…



fork-join

workers: 4 branching factor: 2 sequential threshold: 256

And so on…

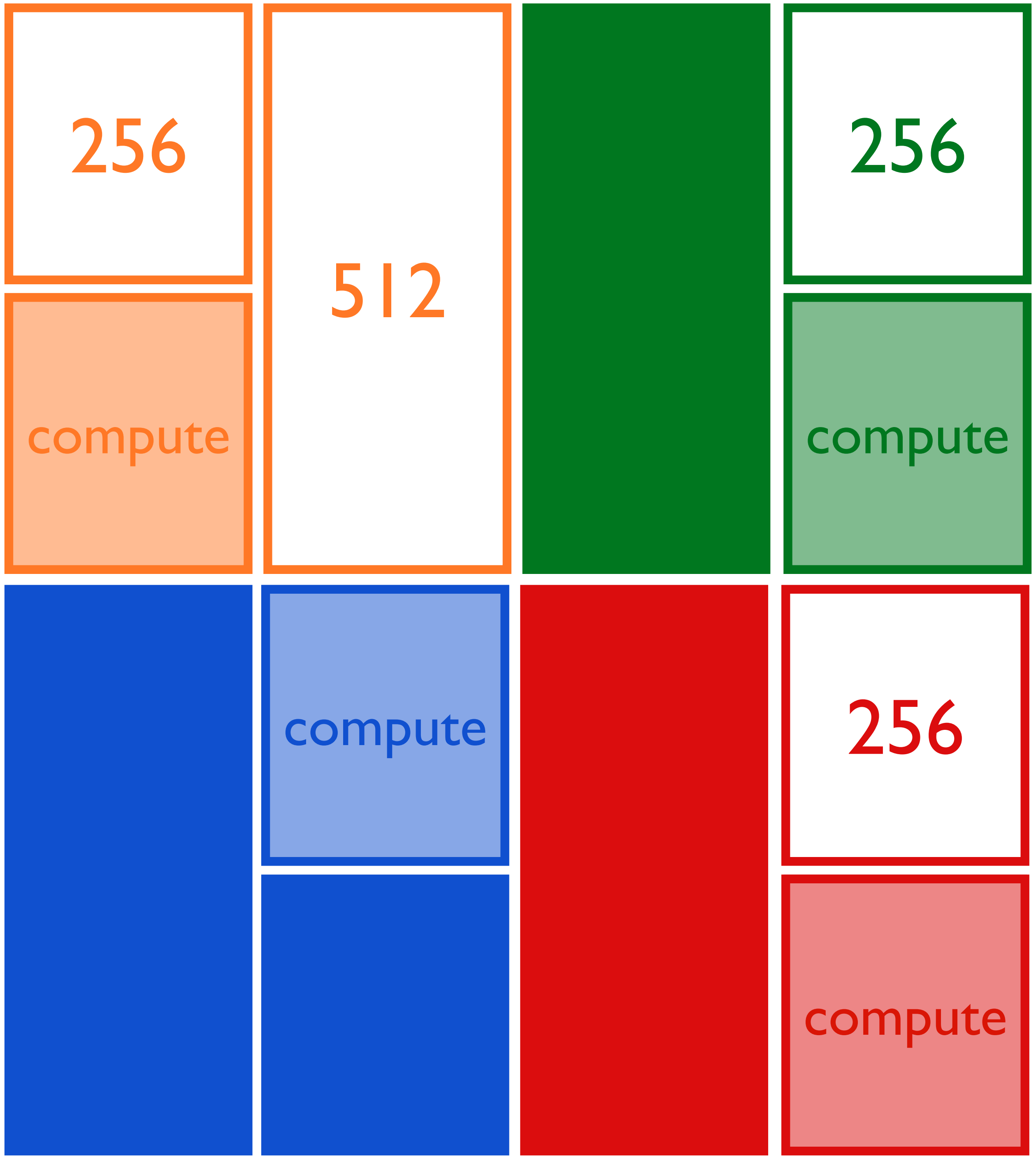


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

And so on...

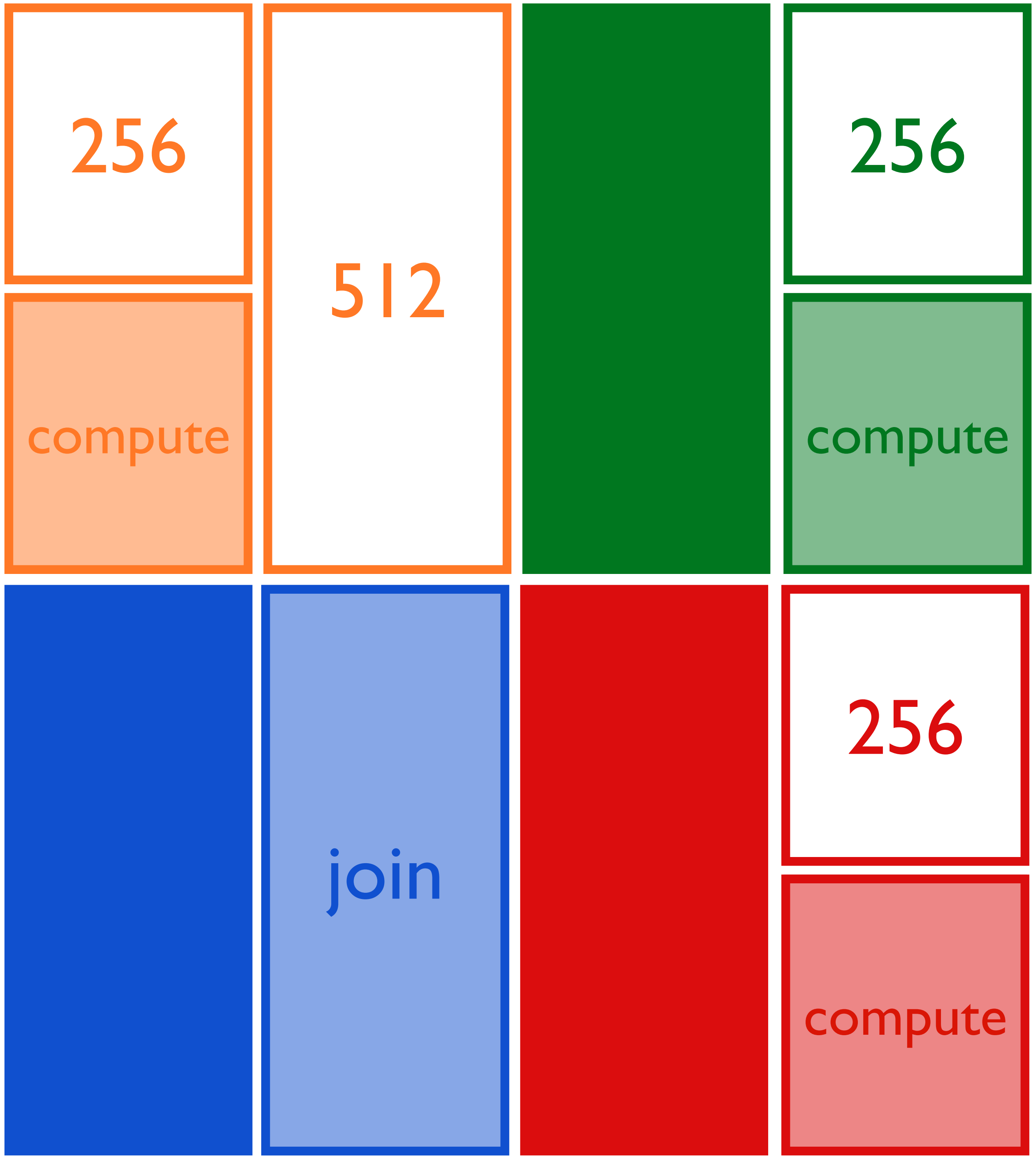


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

And so on...

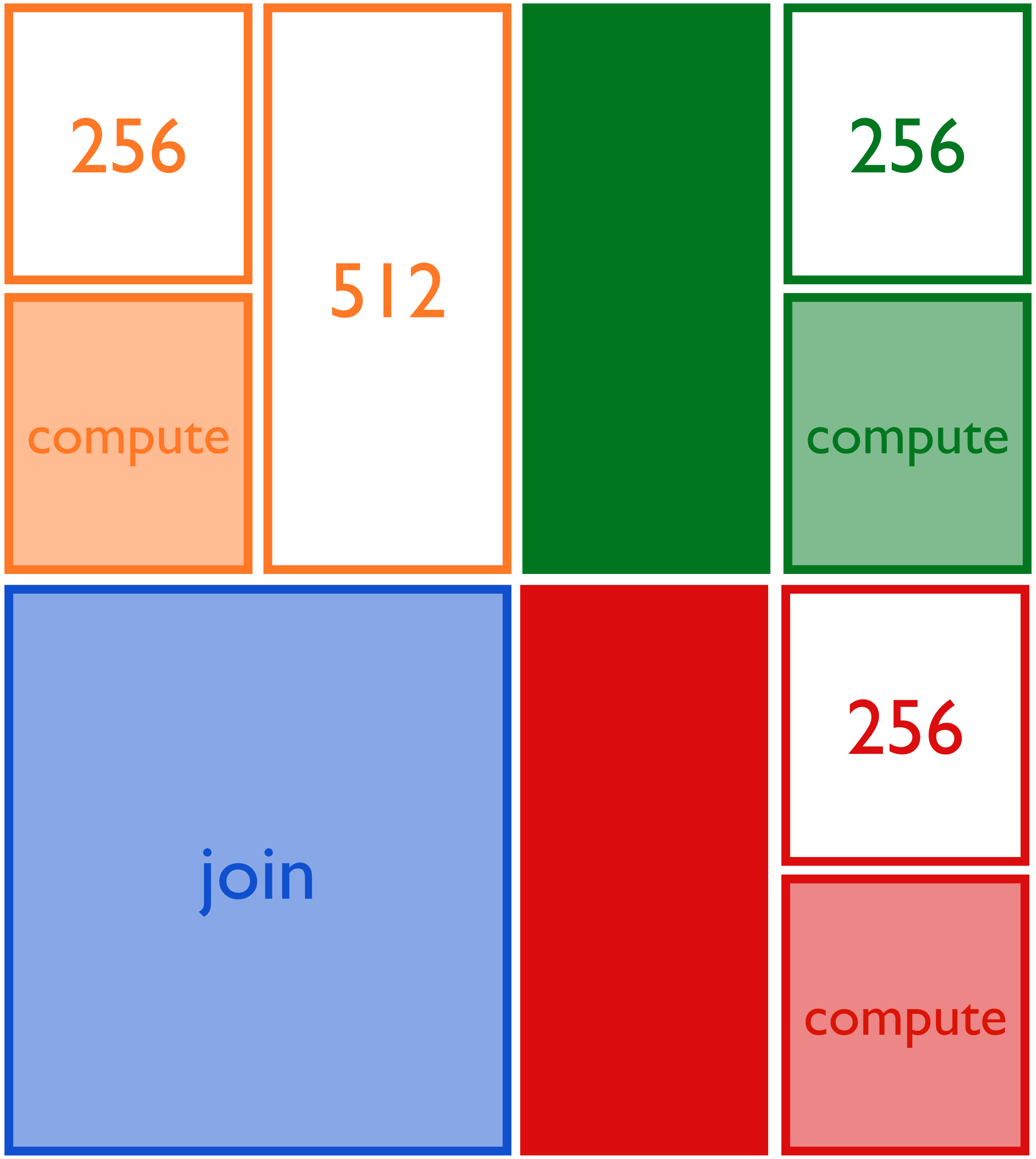


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

And so on...

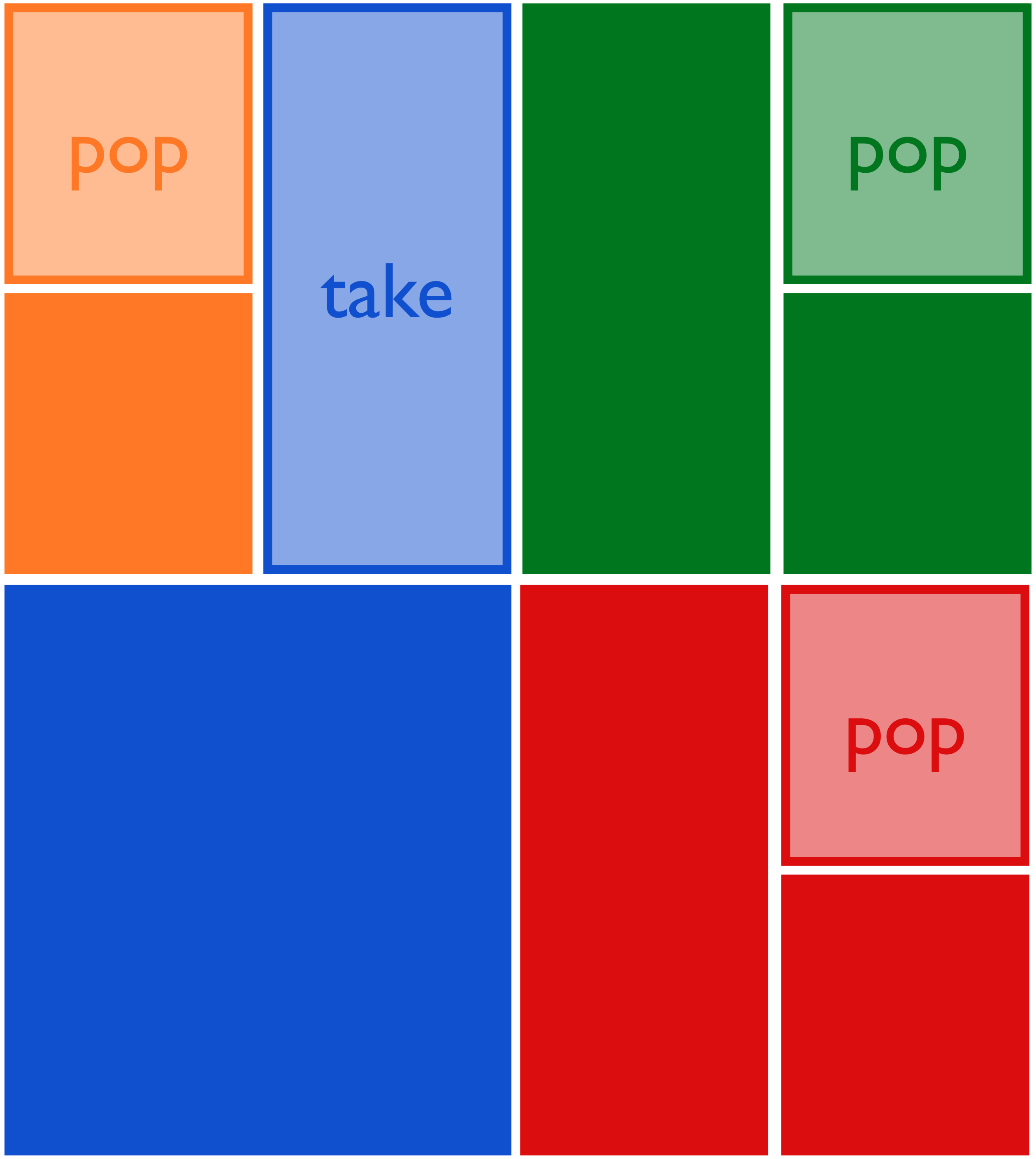


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

If one of the workers falls behind, another worker can *take* tasks from its dequeue.

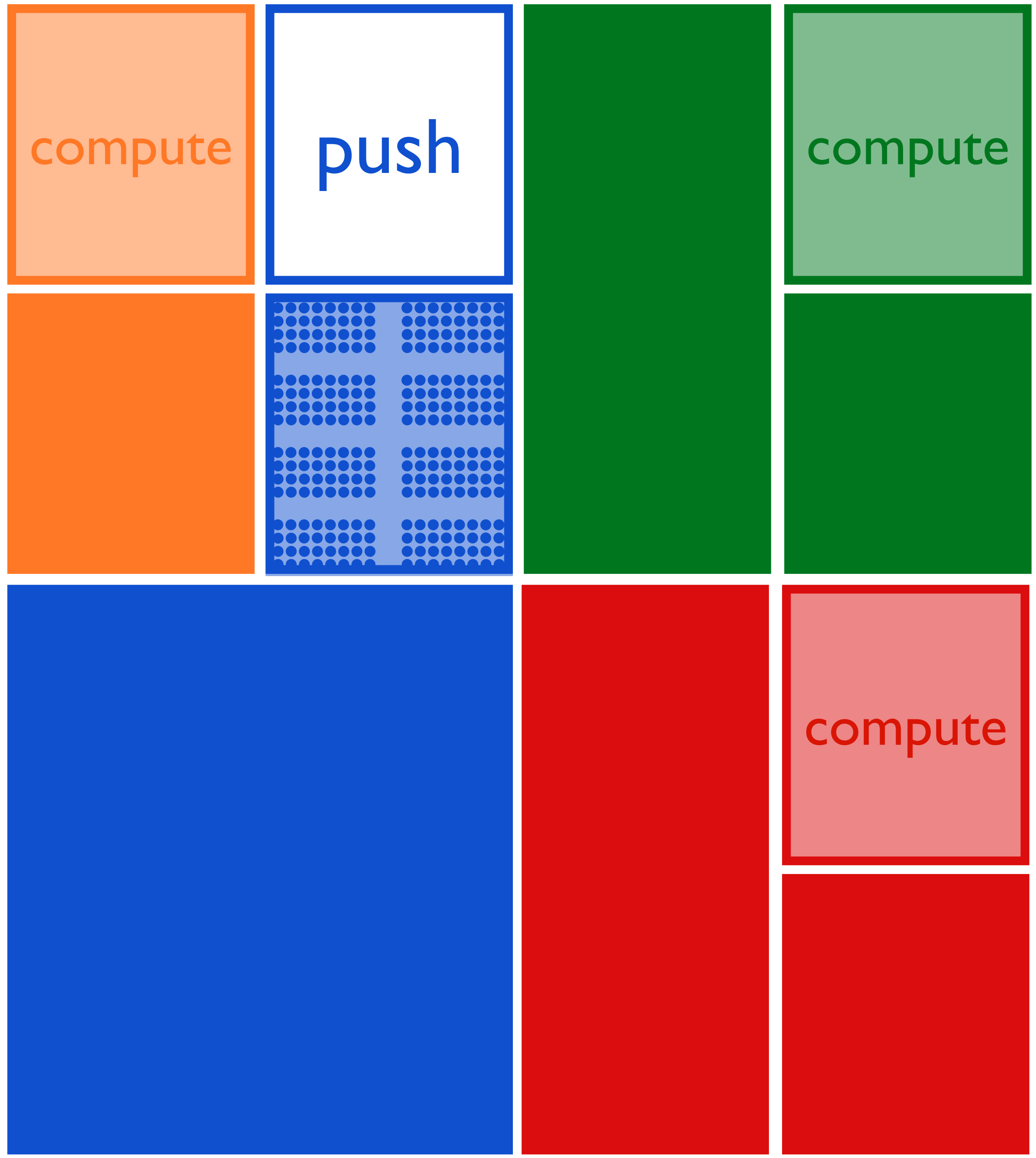


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

If one of the workers falls behind, another worker can *take* tasks from its dequeue.

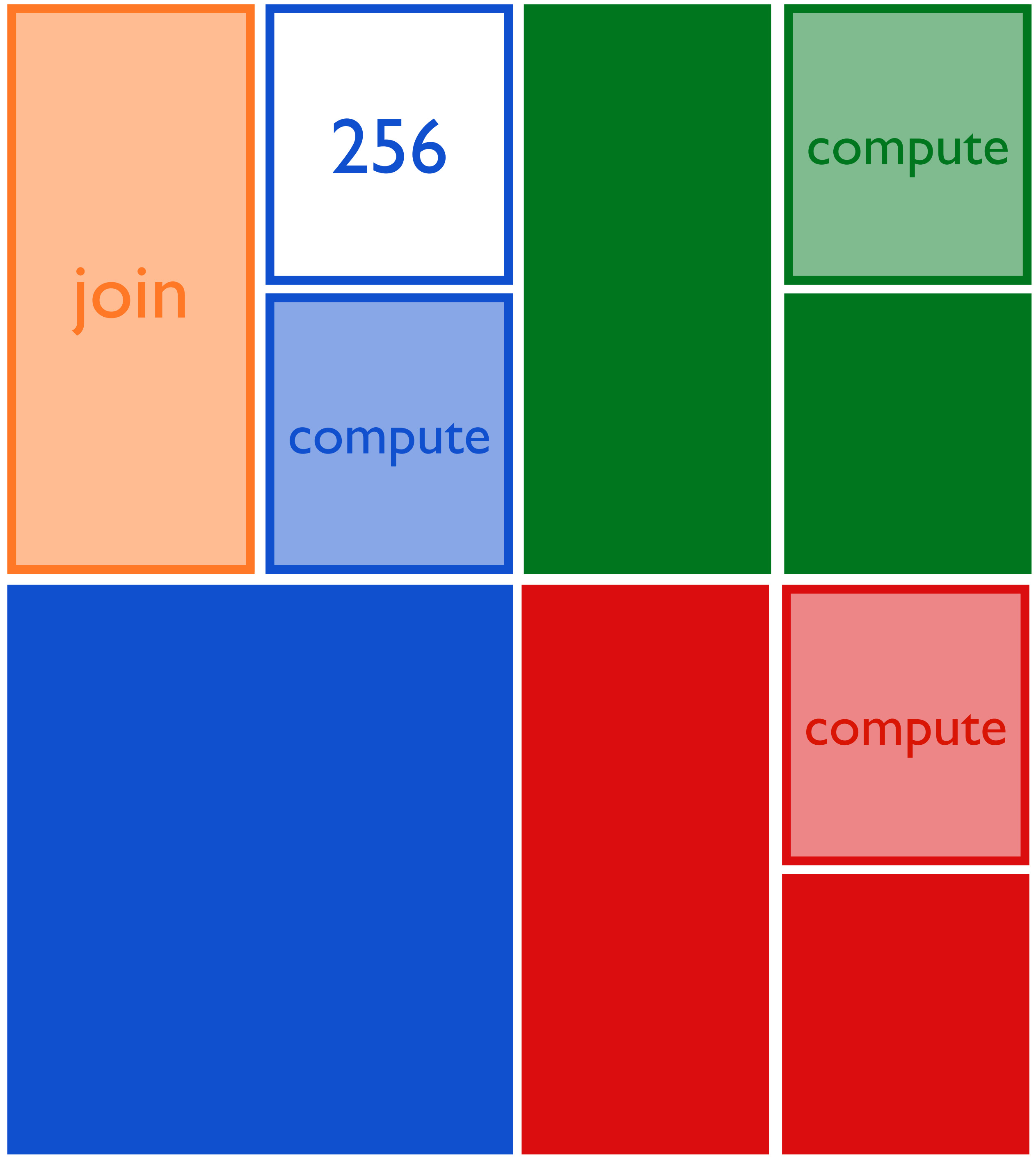


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

If one of the workers falls behind, another worker can *take* tasks from its dequeue.

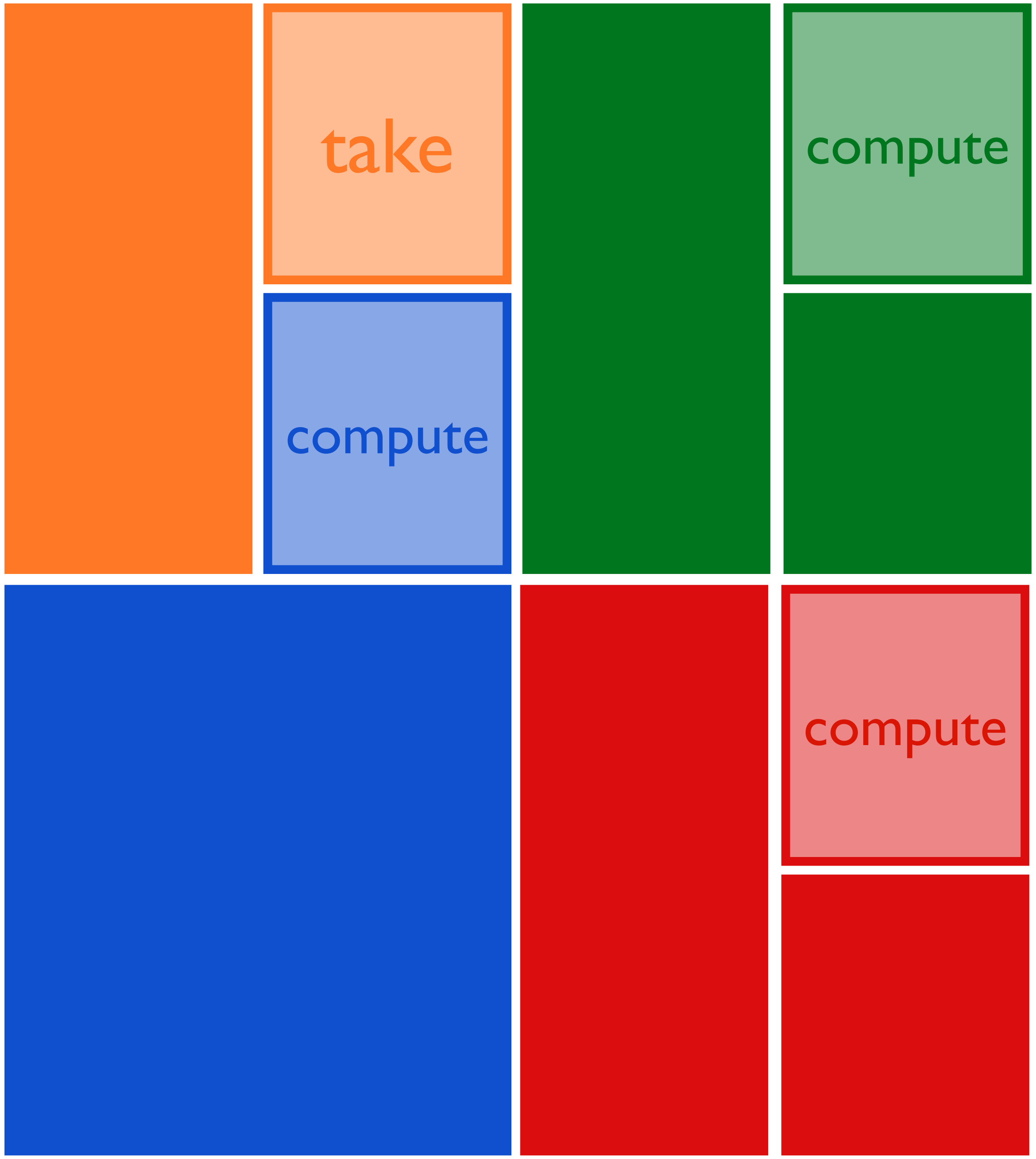


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

If one of the workers falls behind, another worker can *take* tasks from its dequeue.

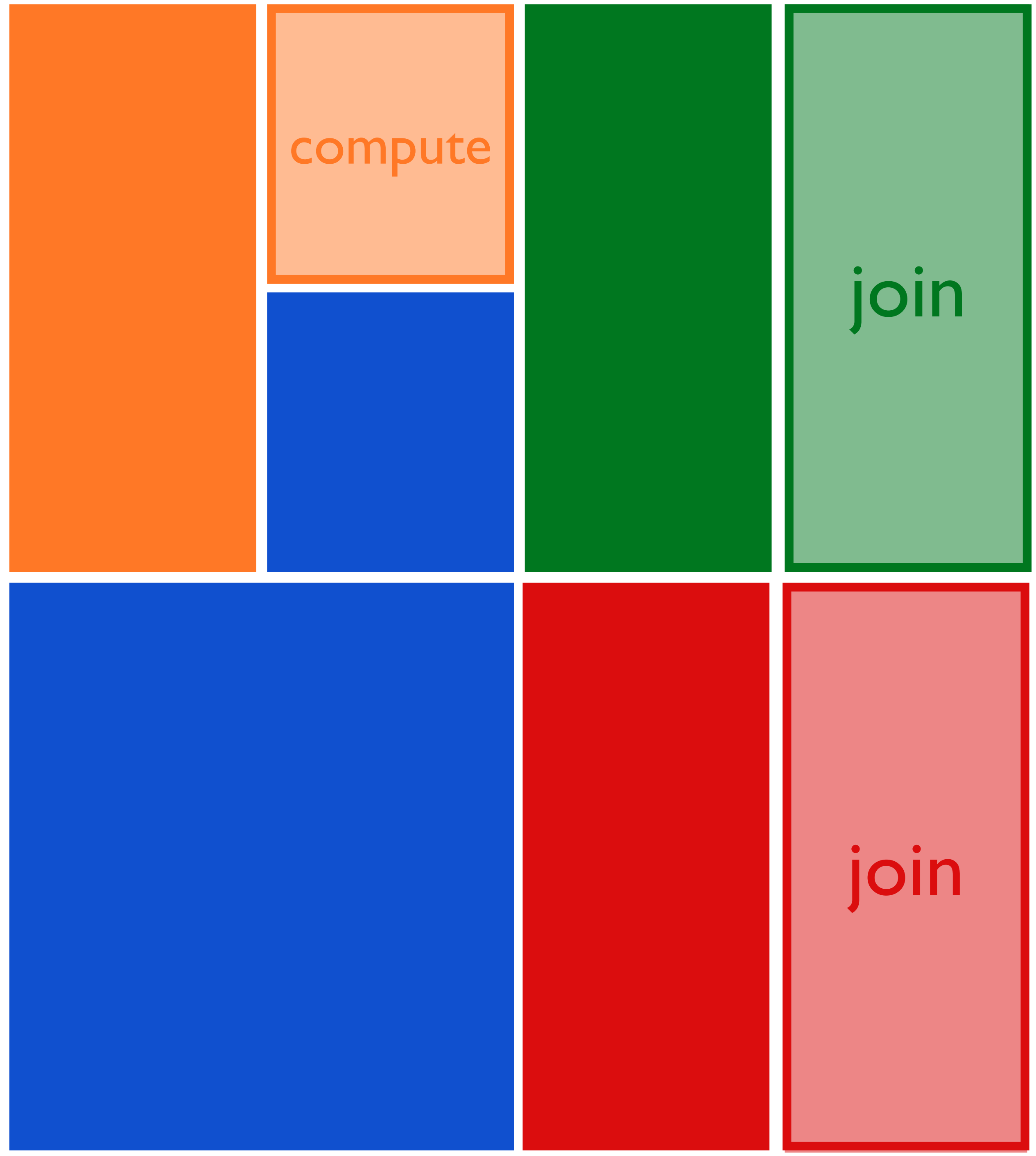


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

If one of the workers falls behind, another worker can *take* tasks from its dequeue.

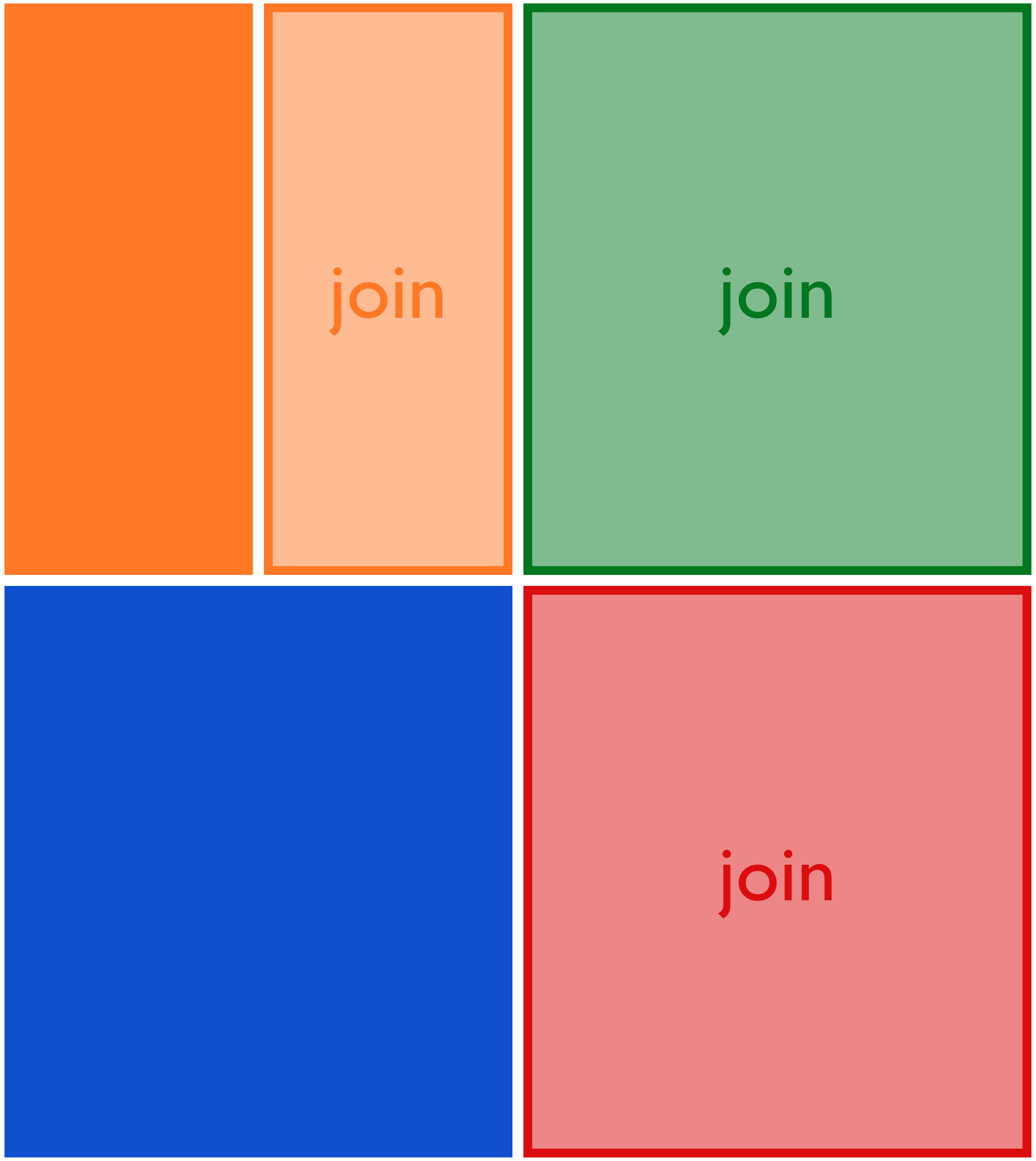


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

And so on...

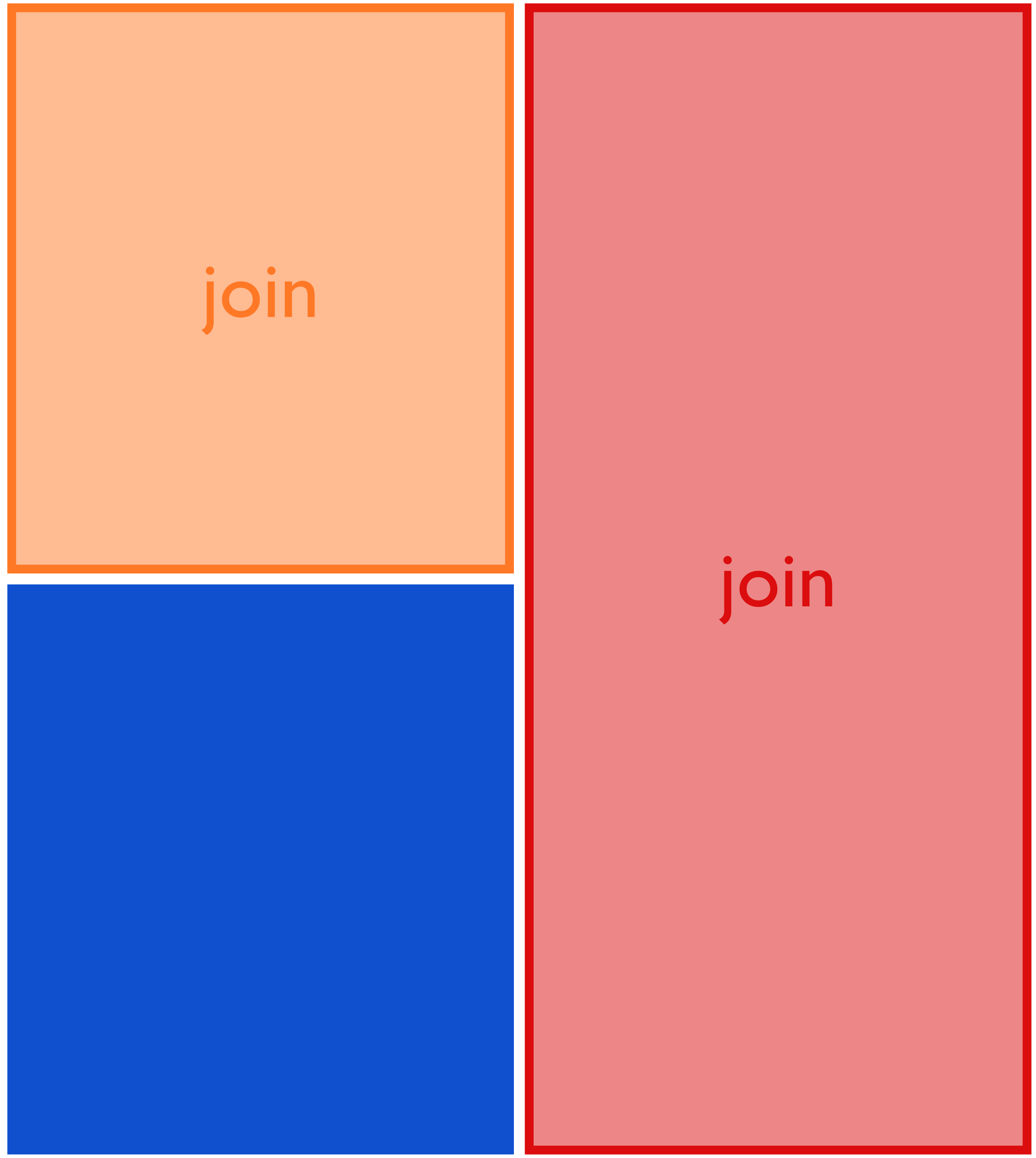


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

And so on...

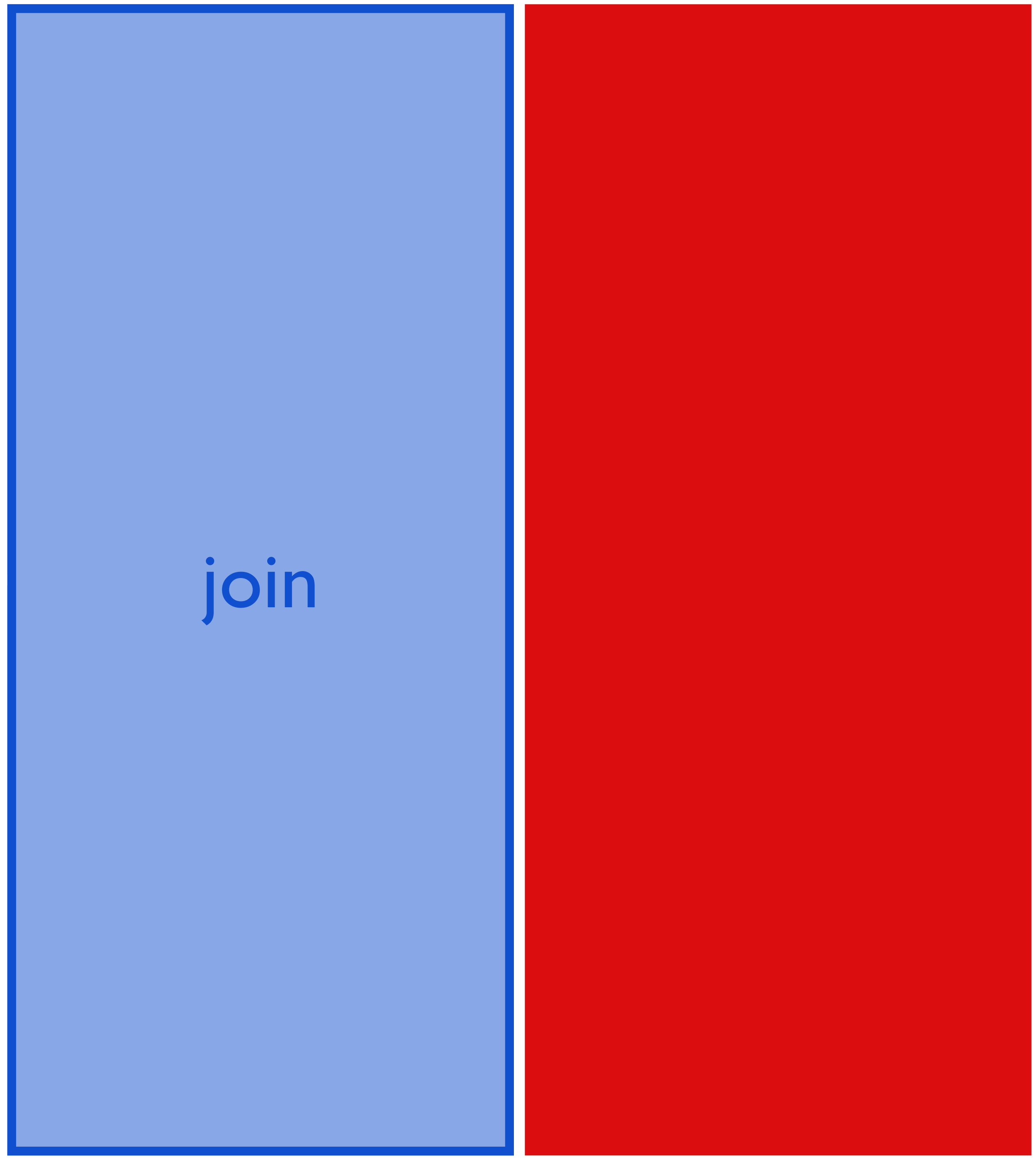


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

And so on…

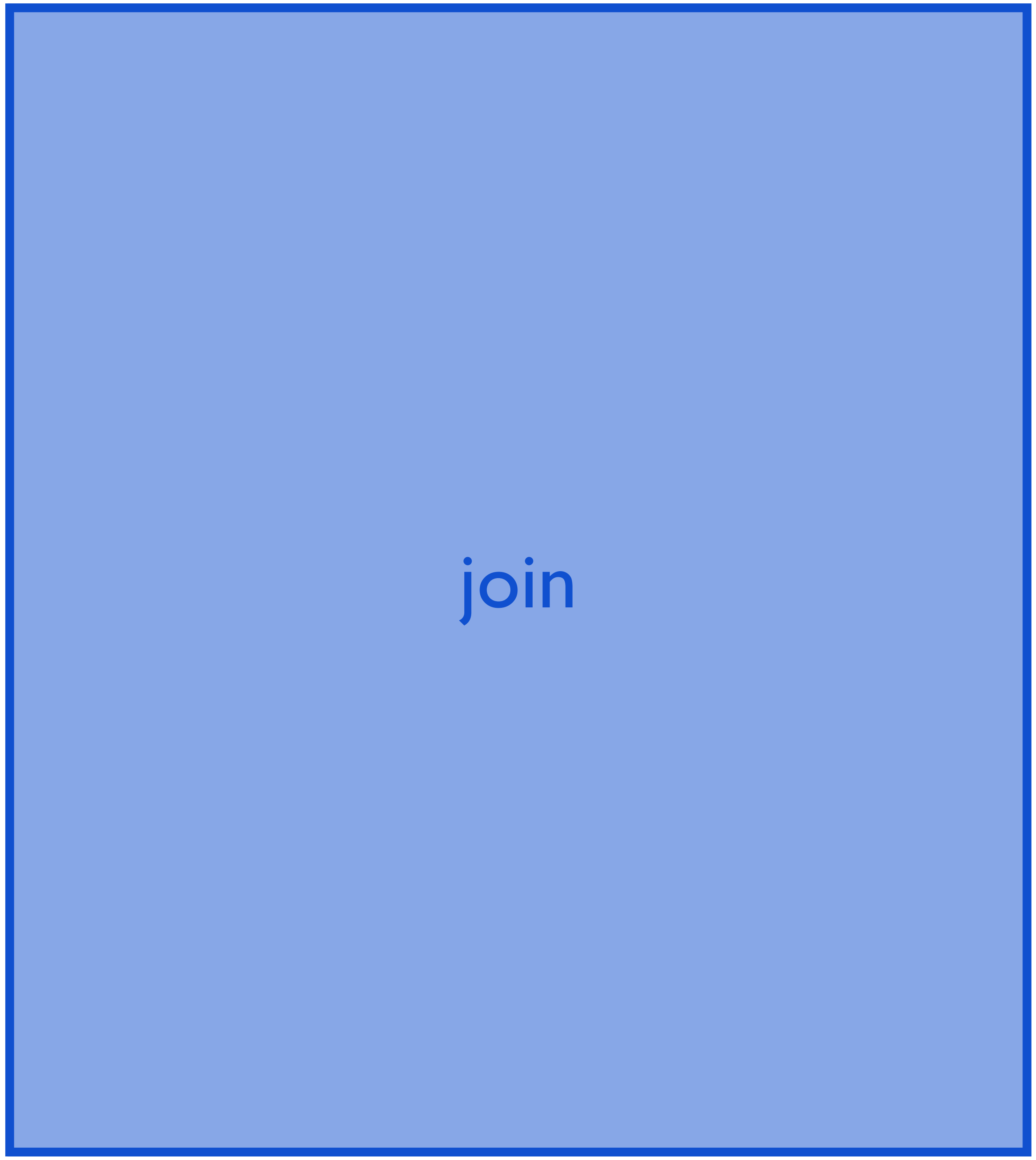


worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

And so on...



worker deques current tasks completed tasks

fork-join

workers: 4 branching factor: 2 sequential threshold: 256

... until the job is complete.

worker deques current tasks completed tasks

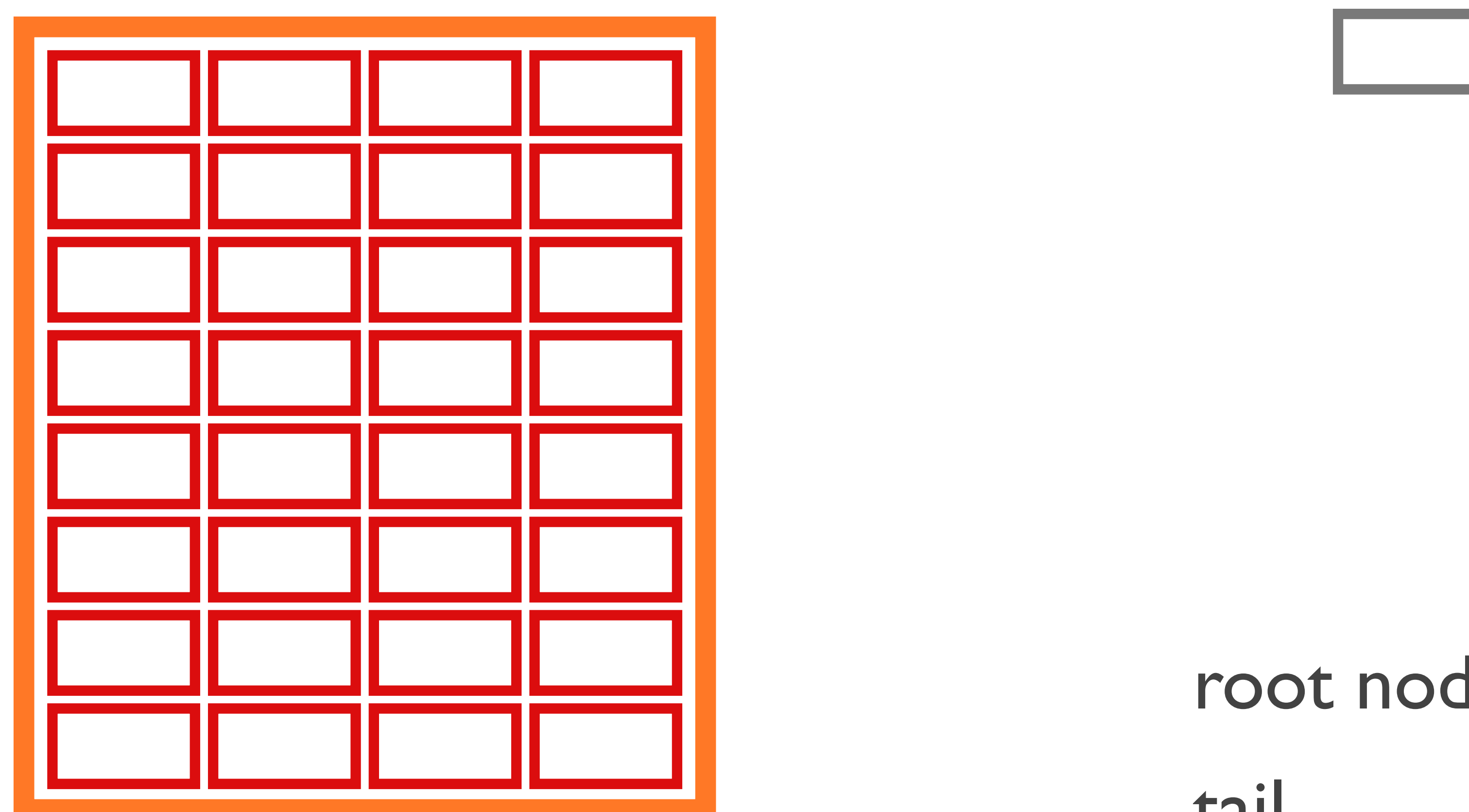
persistent vector

parallelism with existing data structures

persistent-vector

00000 00000 00000

count: 0
shift: 5



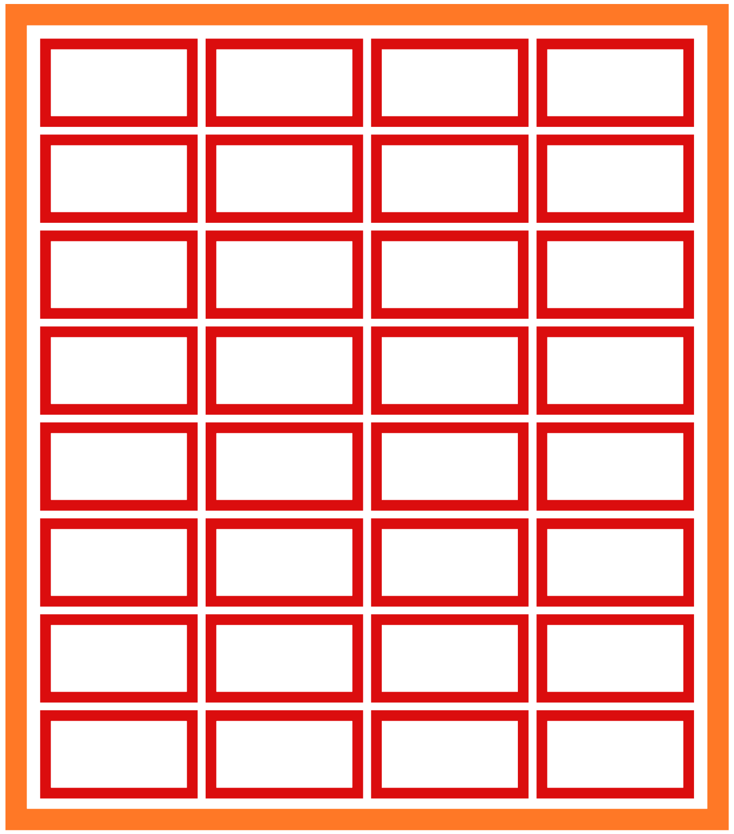
A PersistentVector contains a root and a tail; the tail is an array that can contain up to 32 Object references, and the root is a Node that can contain up to 32 child Nodes.

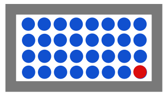
root node
tail
nodes
obj refs
selected

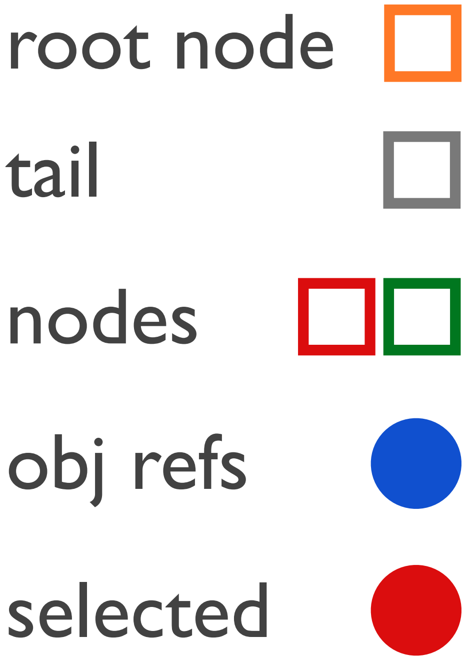
persistent-vector

count: 32
shift: 5

Values are added to the tail...

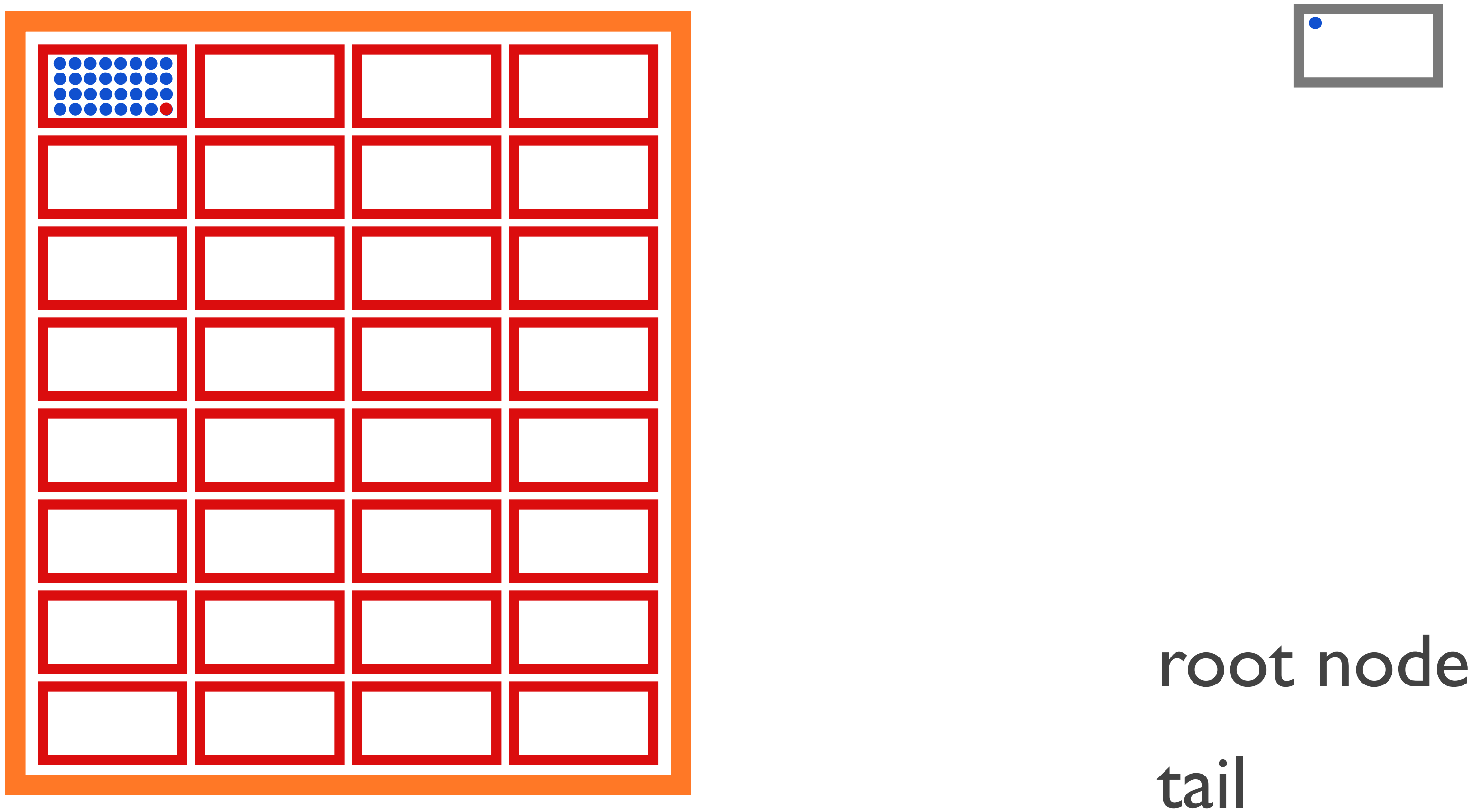


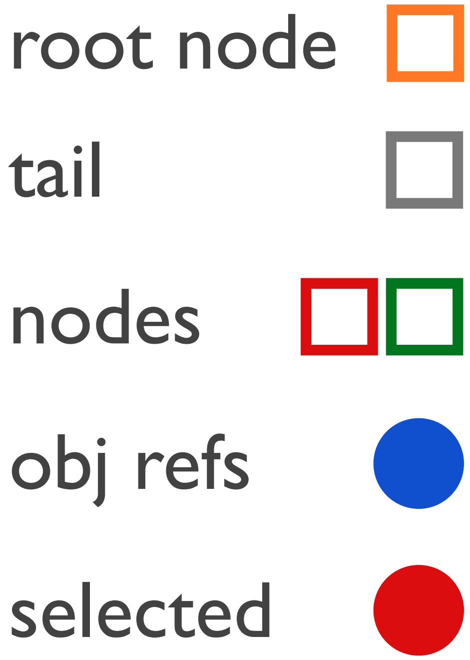




count: 33
shift: 5

... until the tail is full, then a new Node is created, containing the 32 Object references from the tail, and inserted as a child of the root.

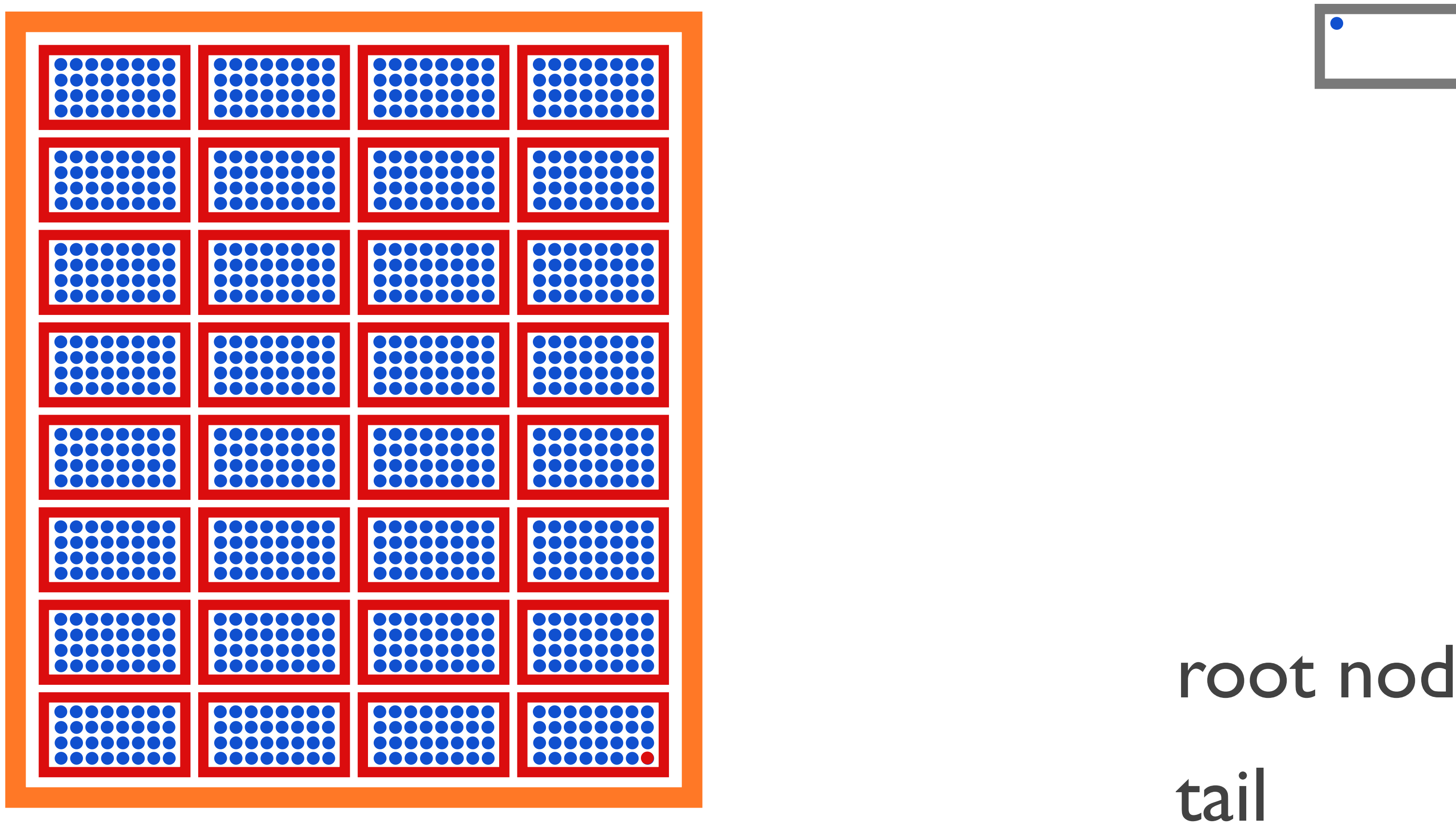




00000 11111 11111

count: 1025
shift: 5

Once the root is full, a new root Node is created, and the existing one is added as a child.



root node

tail

nodes

obj refs

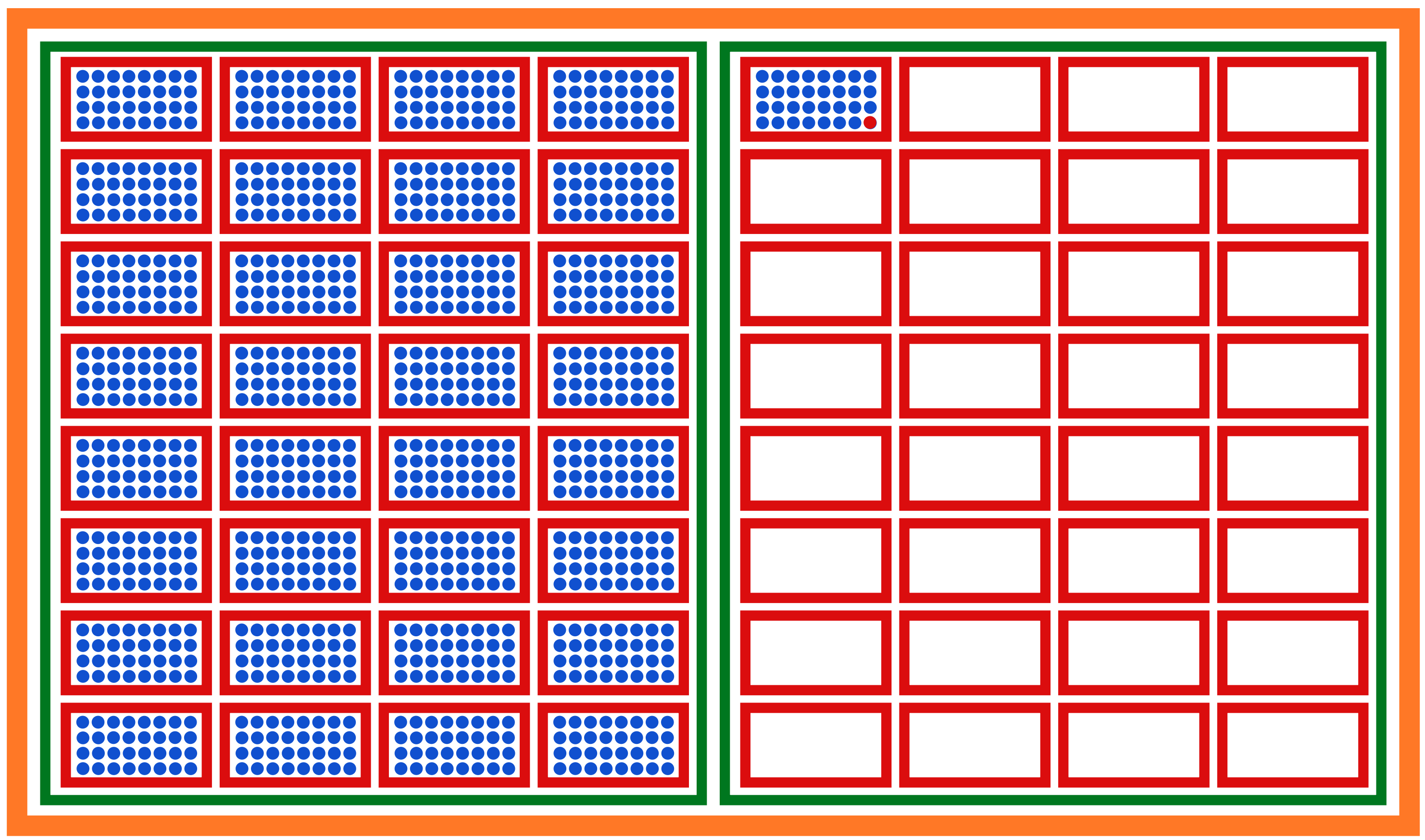
selected

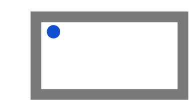
persistent-vector

00001 00000 11111

count: 1057
shift: 10

Once the root is full, a new root Node is created, and the existing one is added as a child.





root node

tail

nodes

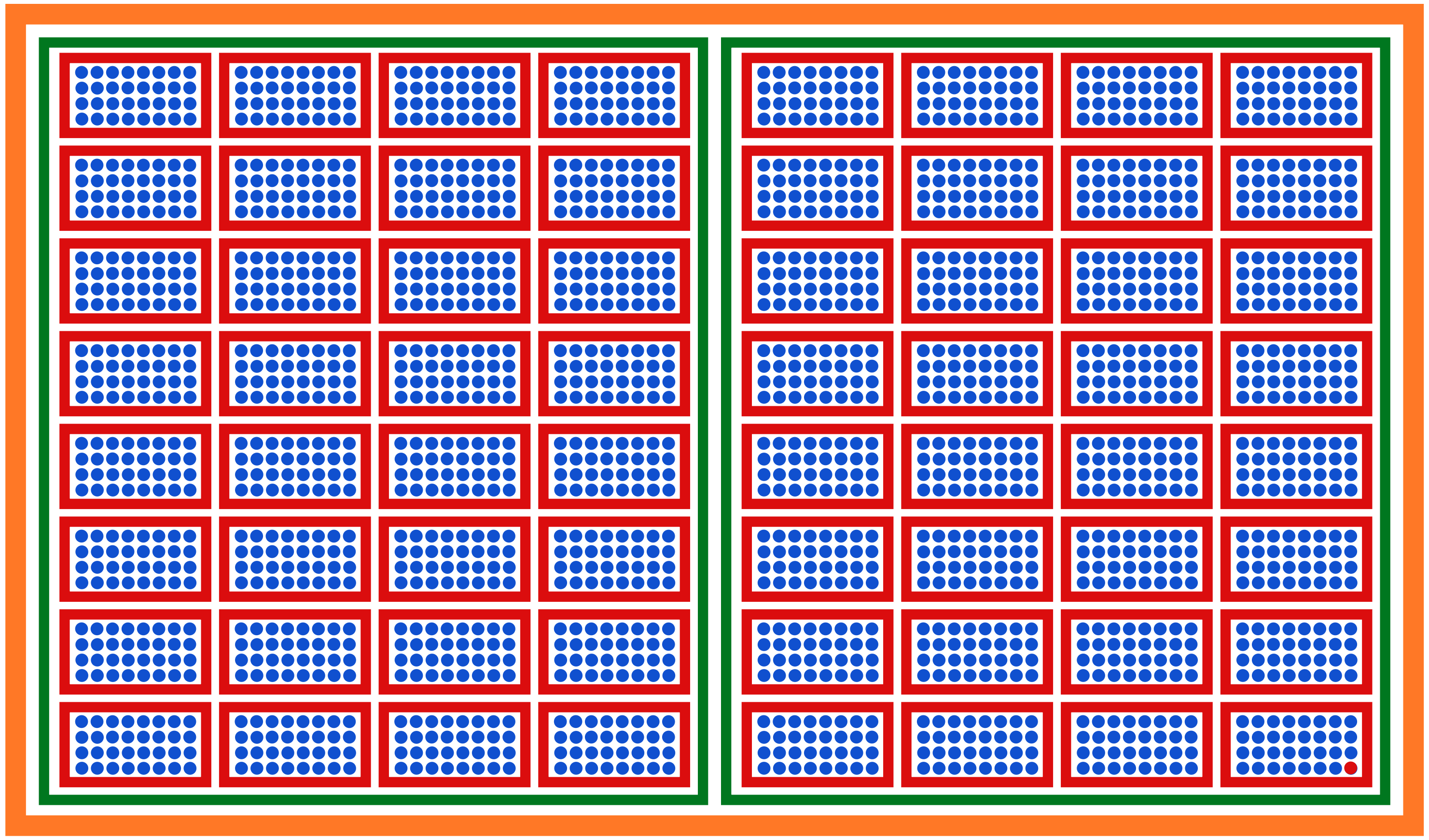
obj refs

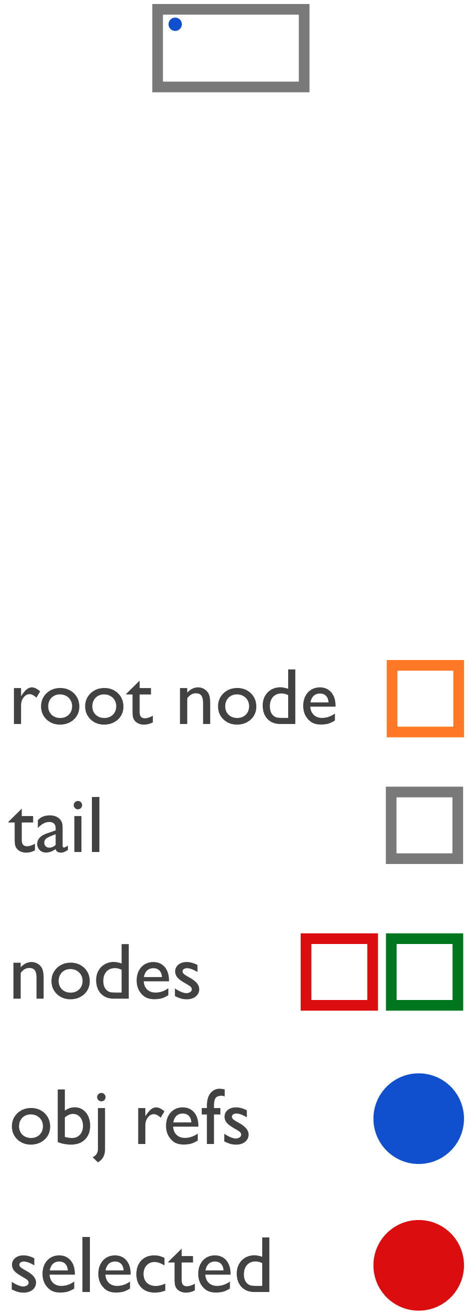
selected

persistent-vector 00001 11111 11111

count: 2049
shift: 10

And so on...

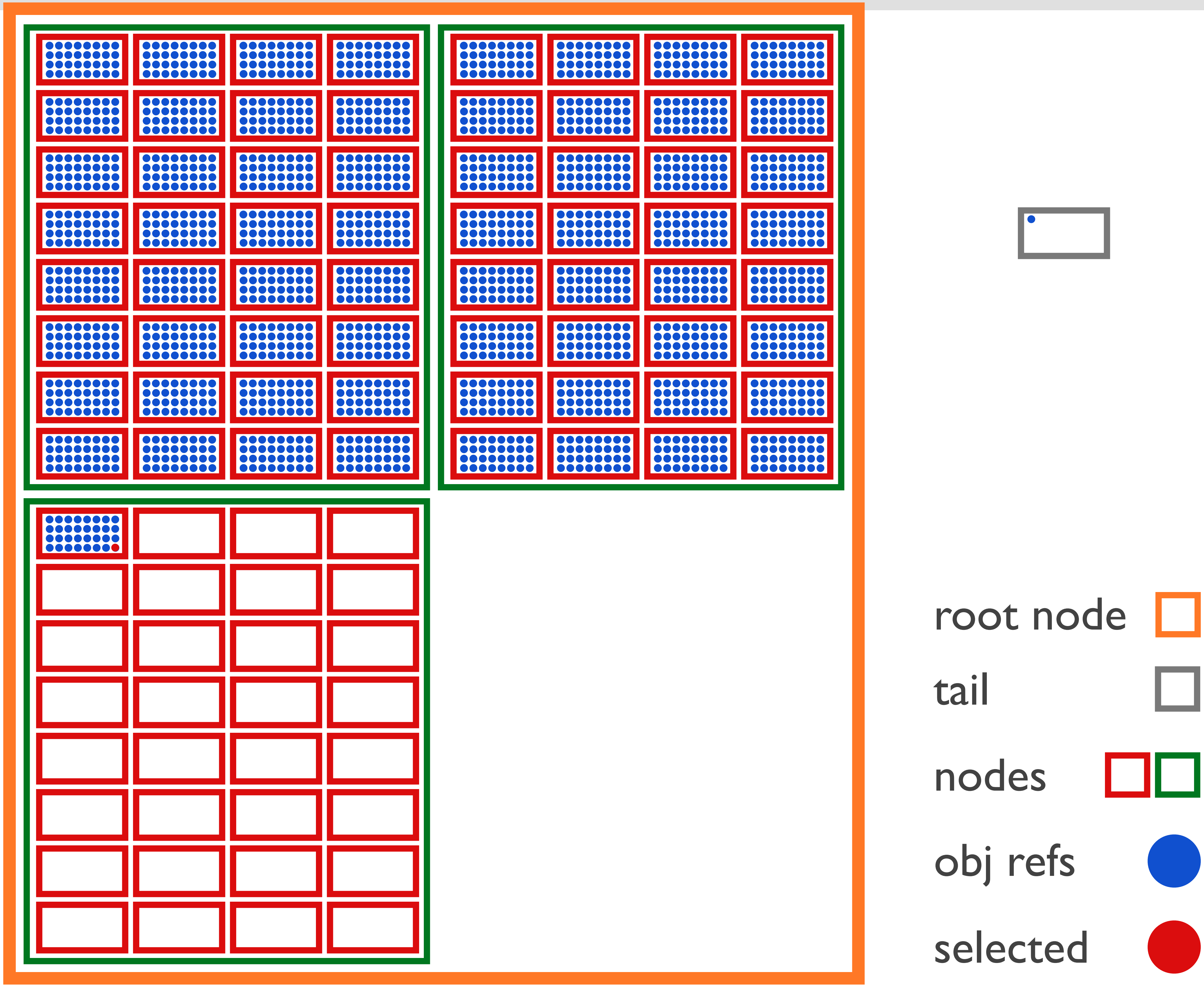




persistent-vector

count: 2081
shift: 10

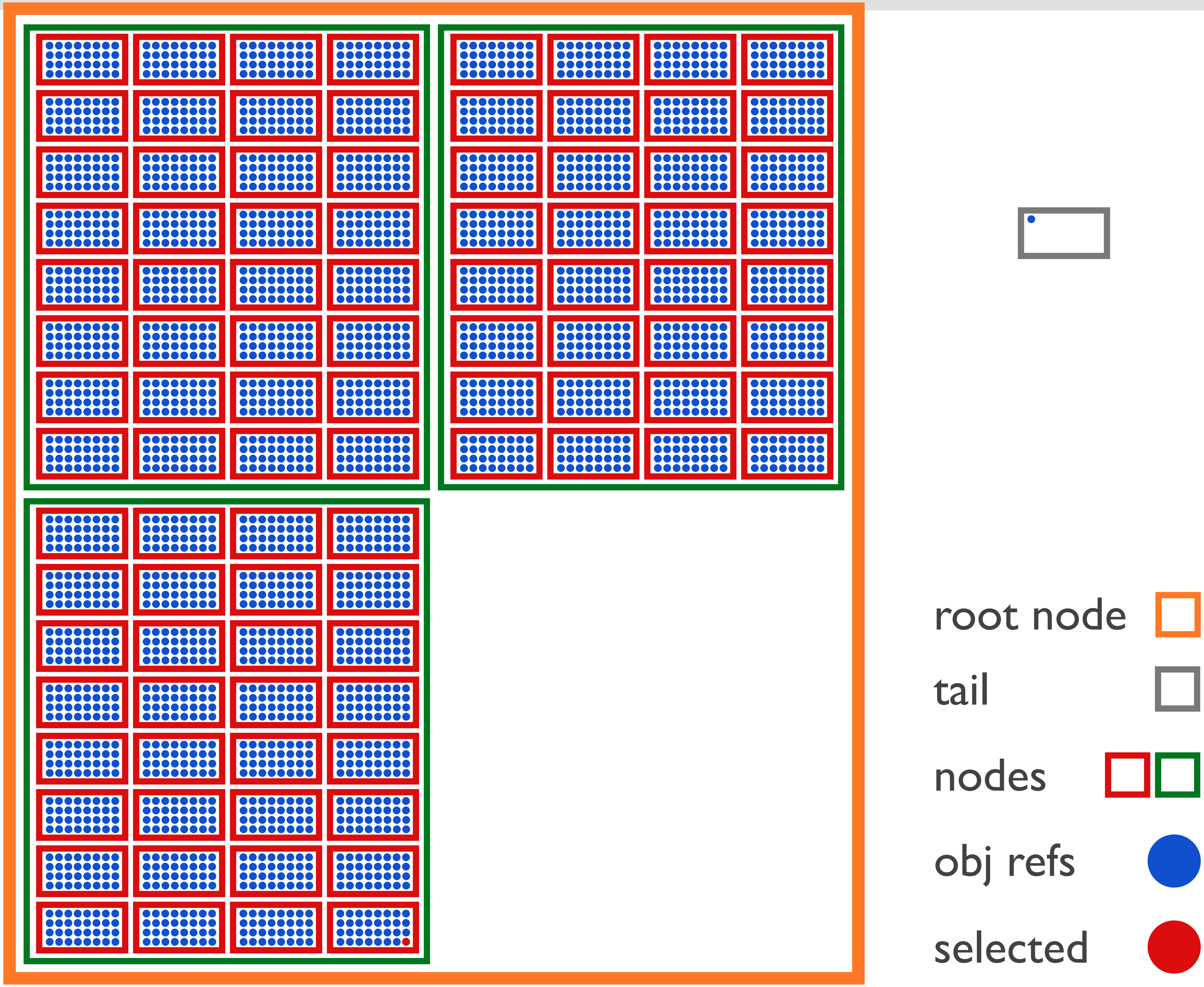
And so on...



persistent-vector

count: 3073
shift: 10

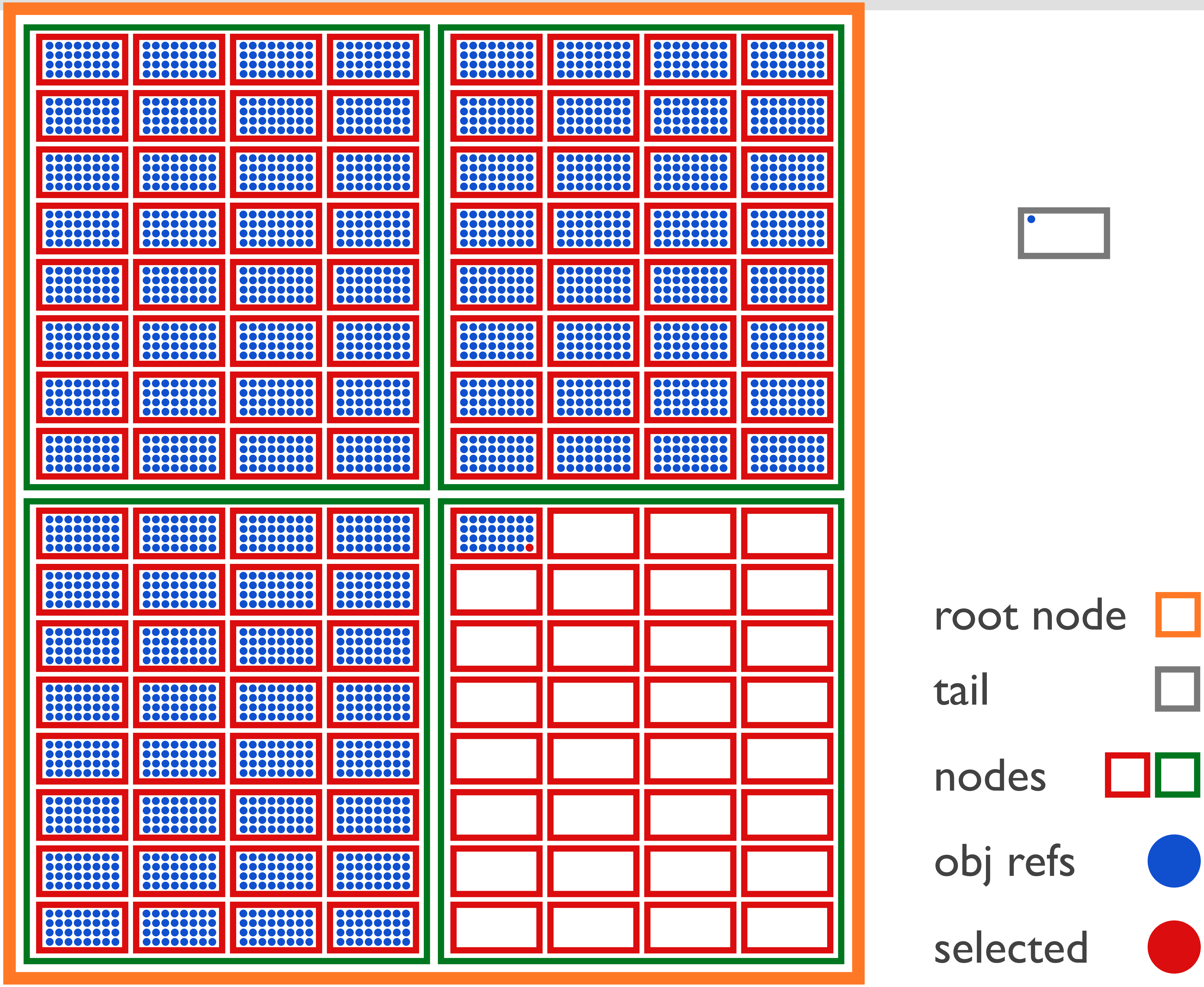
And so on...



persistent-vector

count: 3105
shift: 10

And so on...

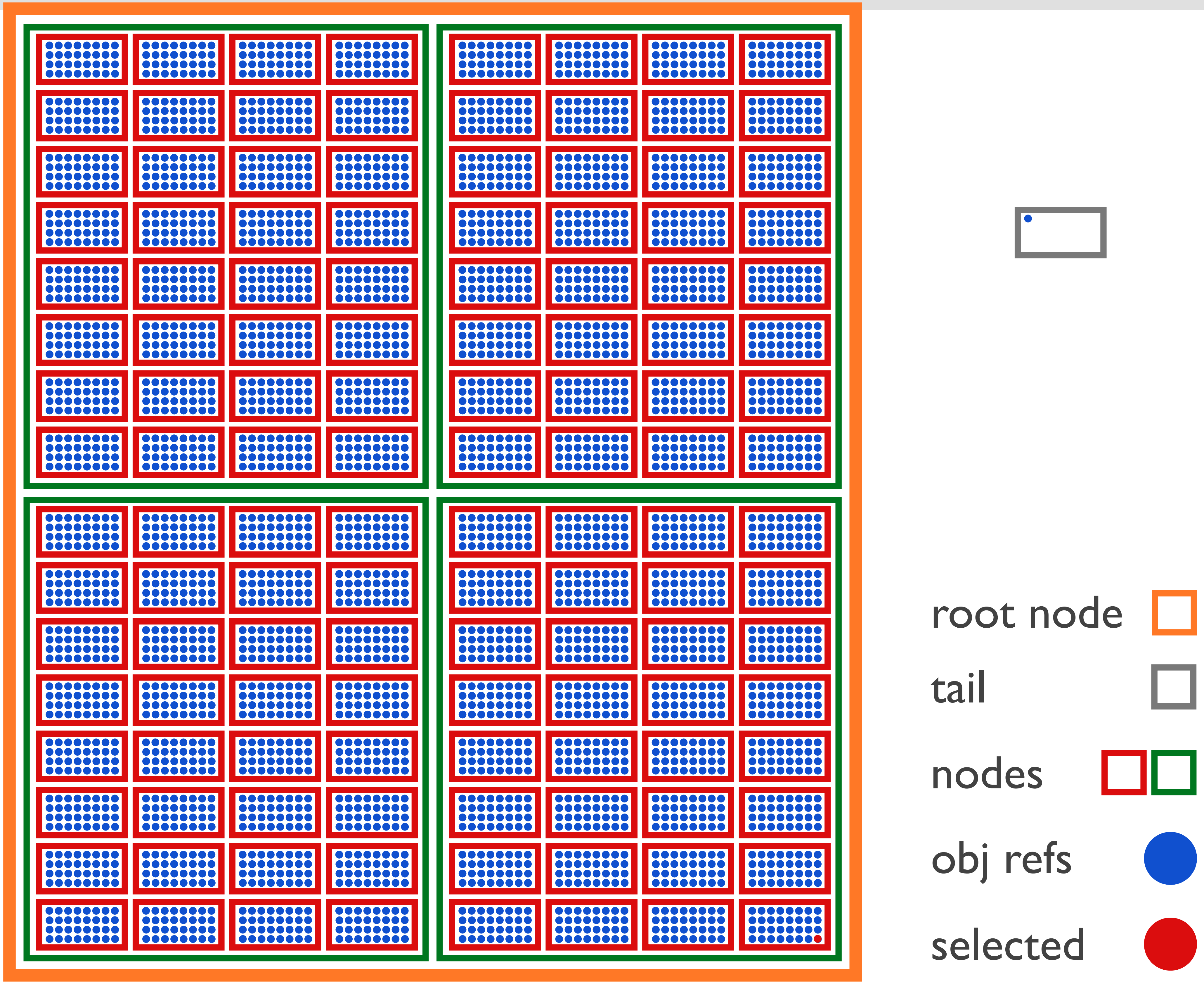


persistent-vector

00011 11111 11111

count: 4097
shift: 10

And so on...



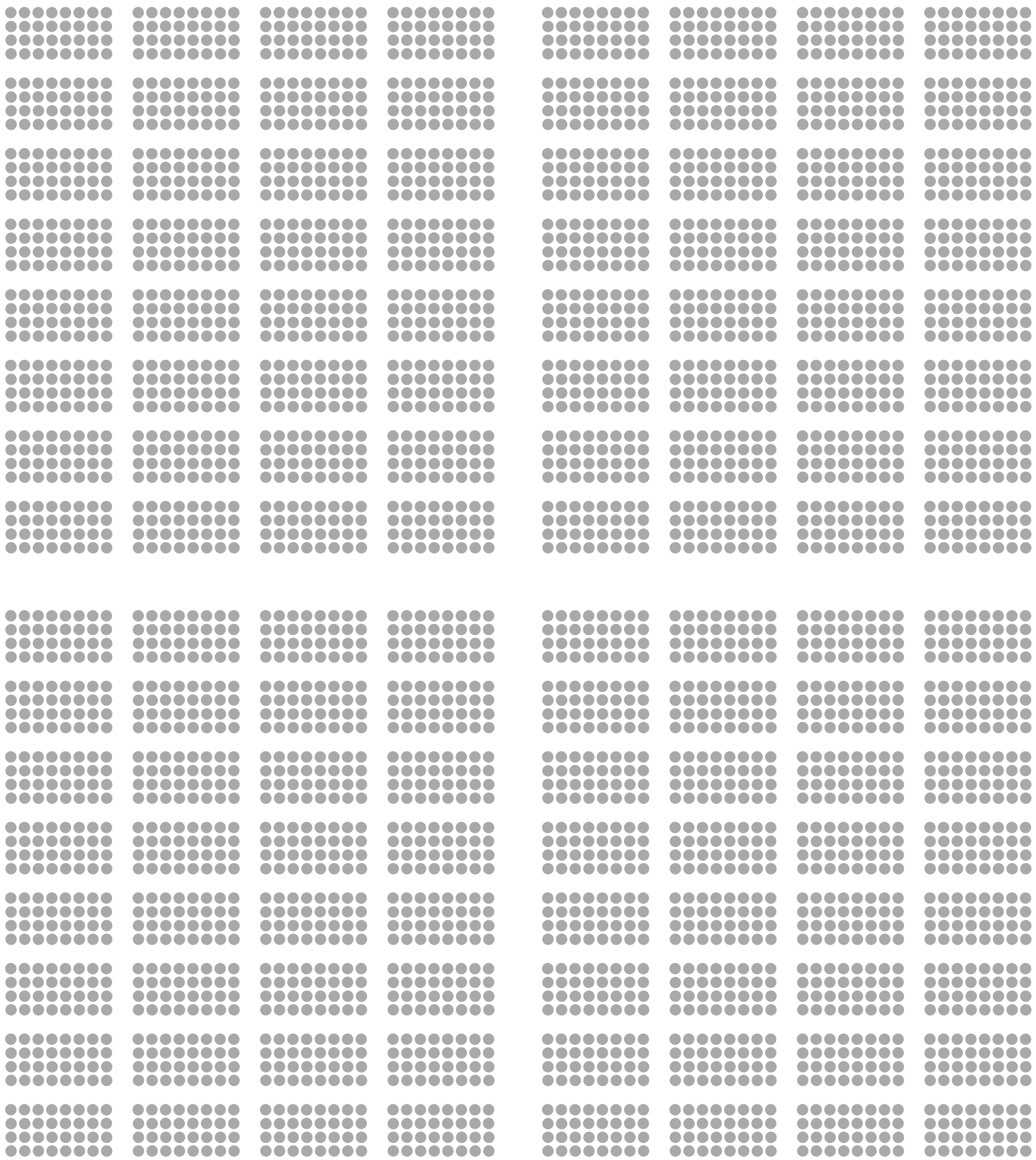
fork-join on persistent vectors

fjvtree, pvmap, and pvreduce

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

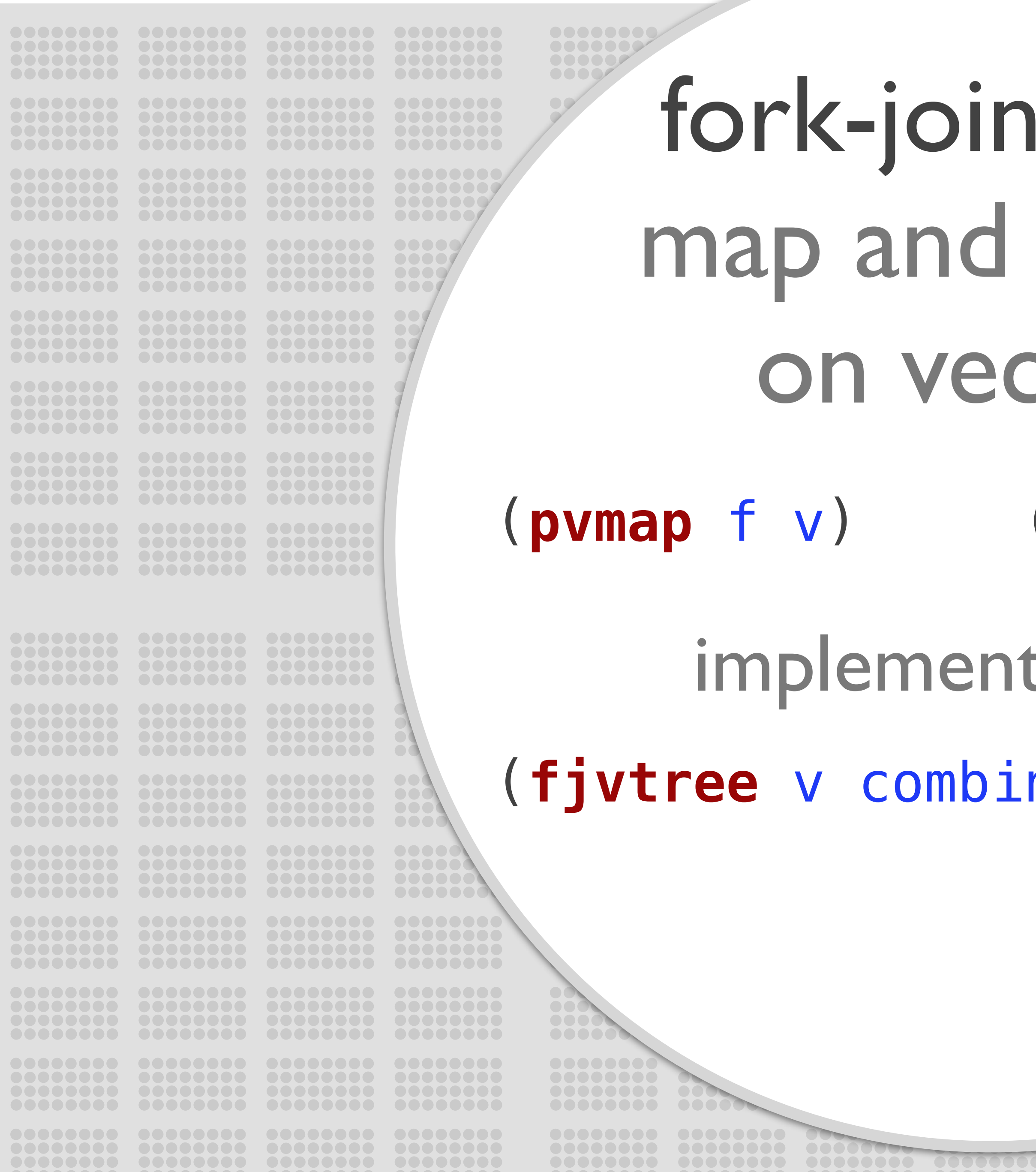


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.



fork-join based

map and reduce on vectors

```
(pvmap f v)        (pvreduce f v)
```

implemented with

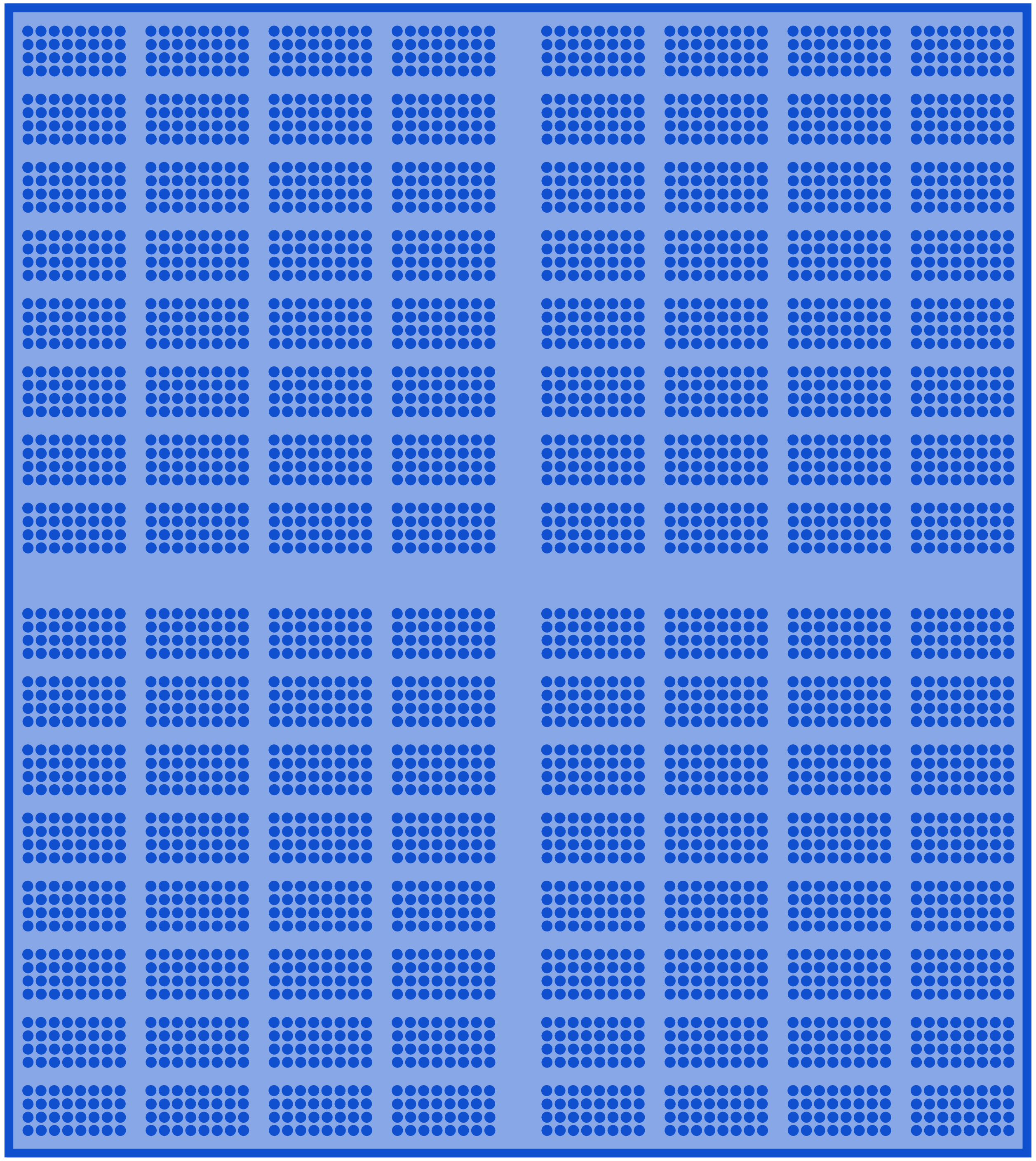
```
(fjvtree v combine-fn leaf-fn)
```

worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

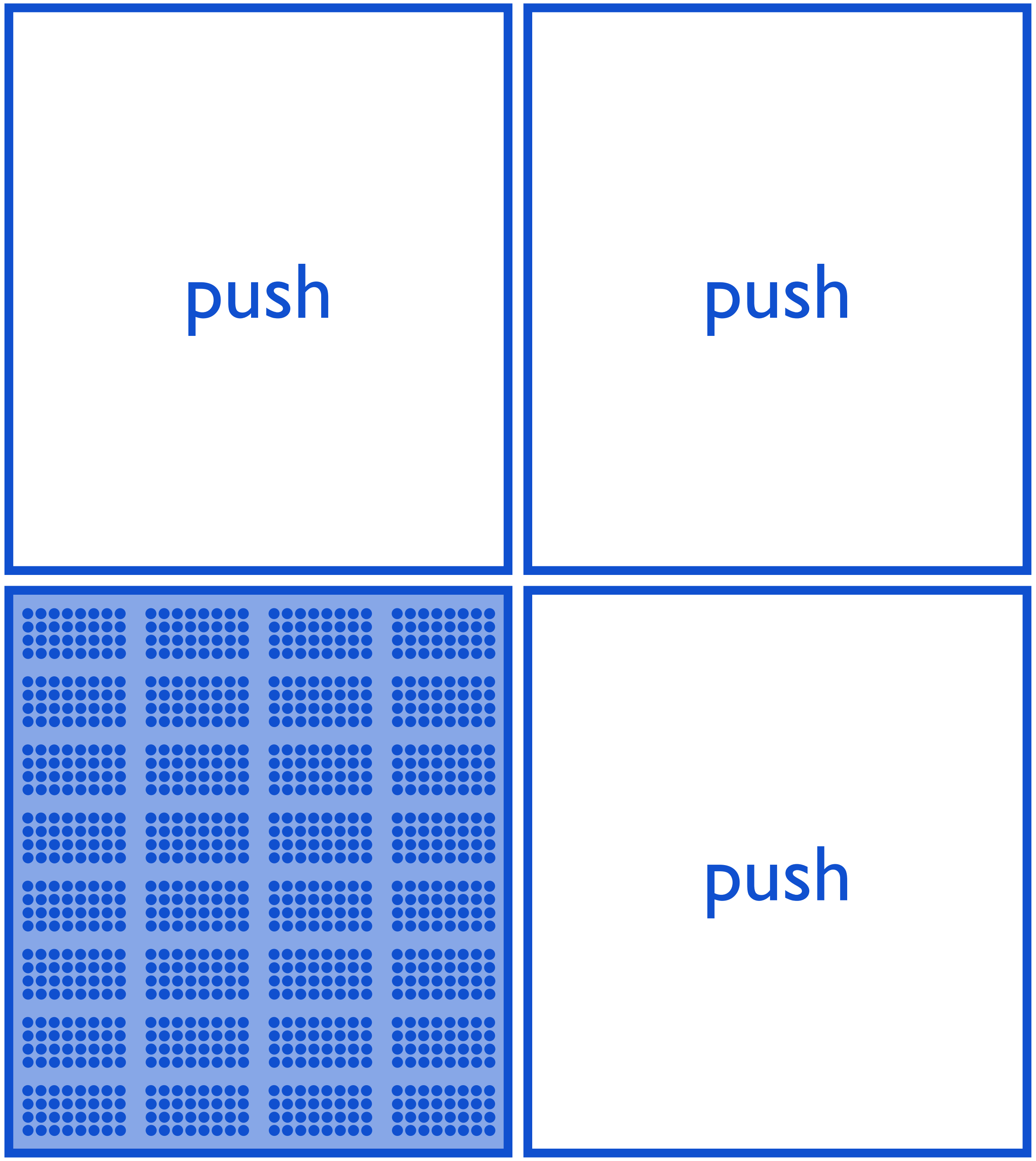


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

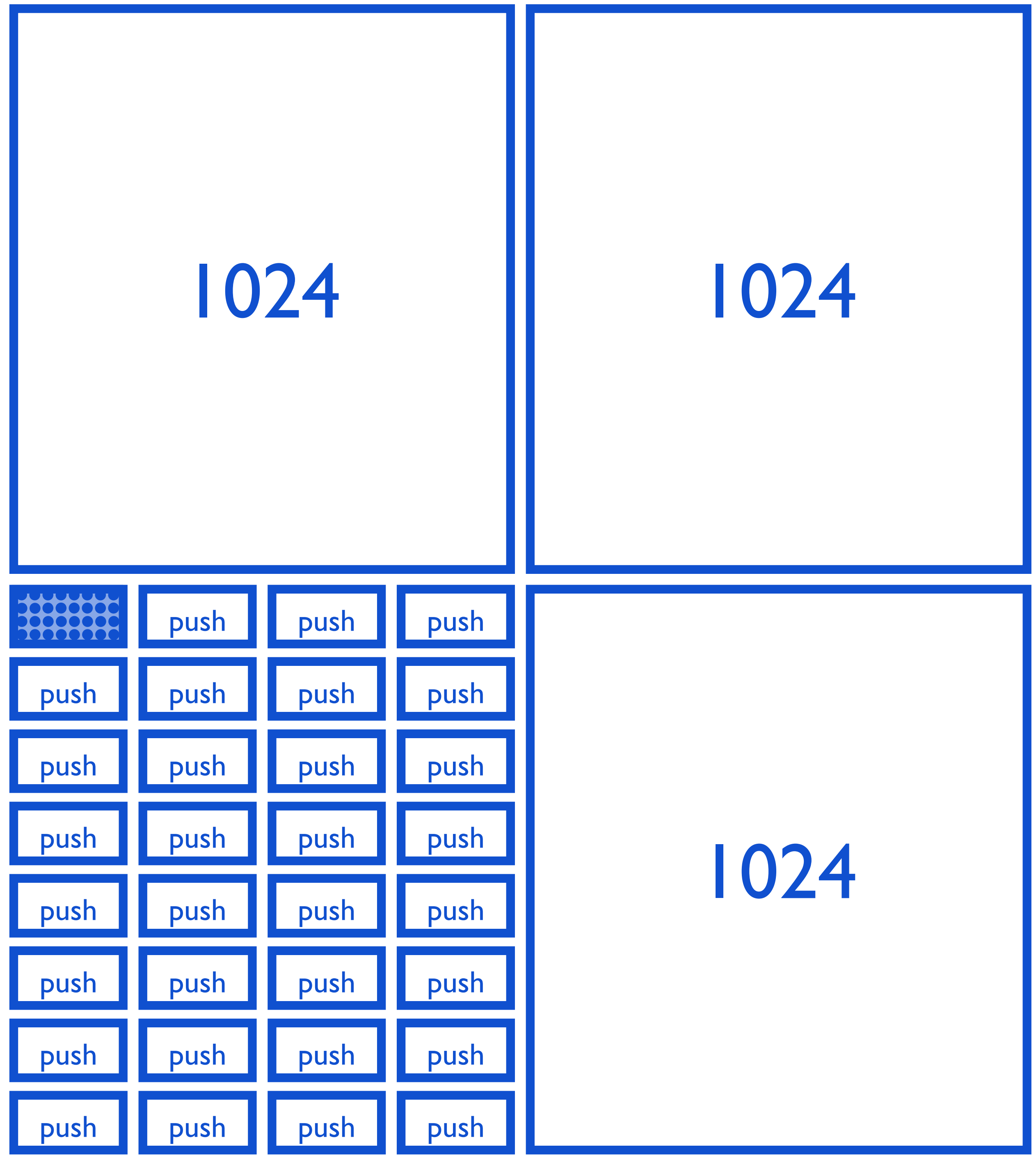


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

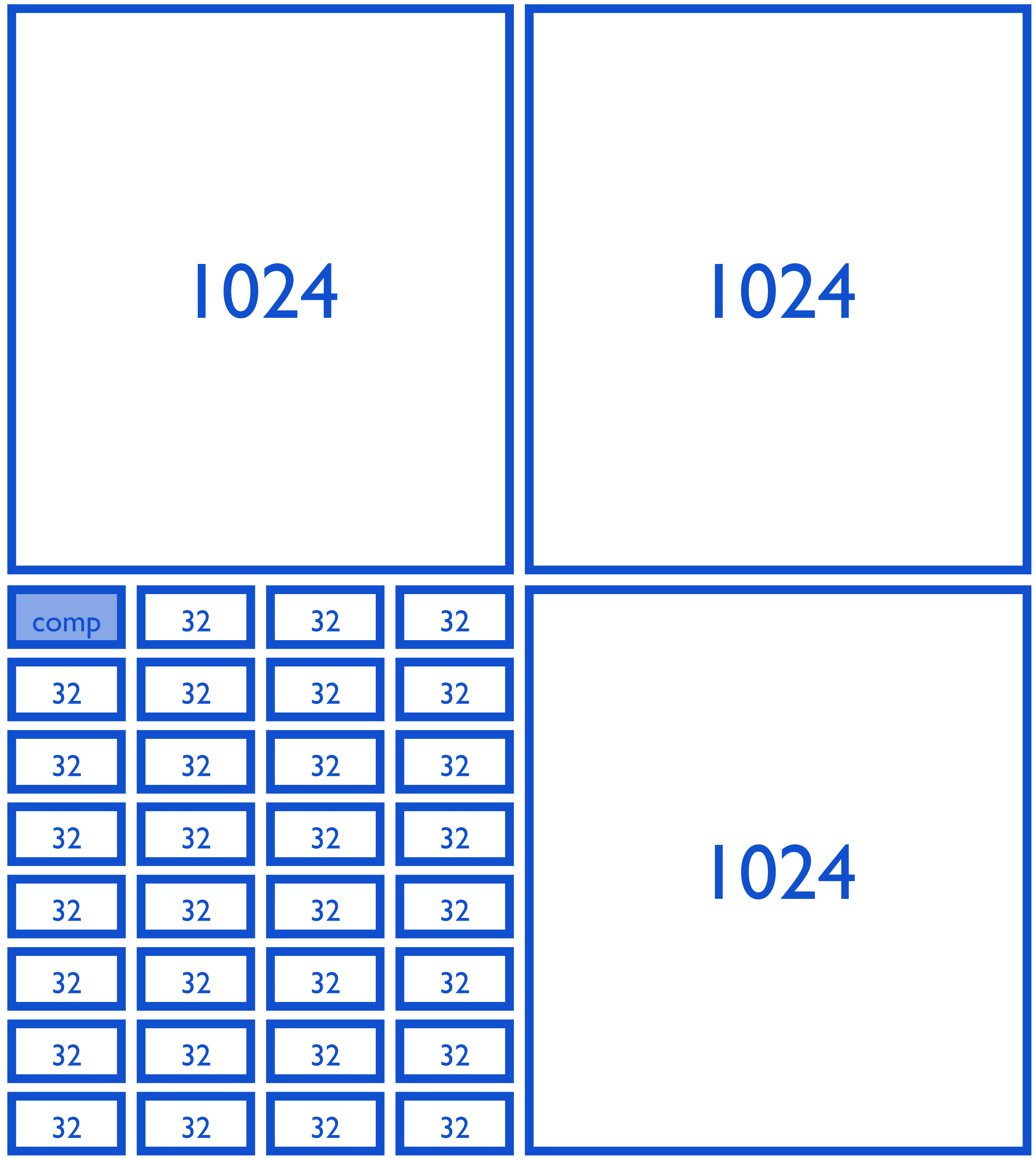


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

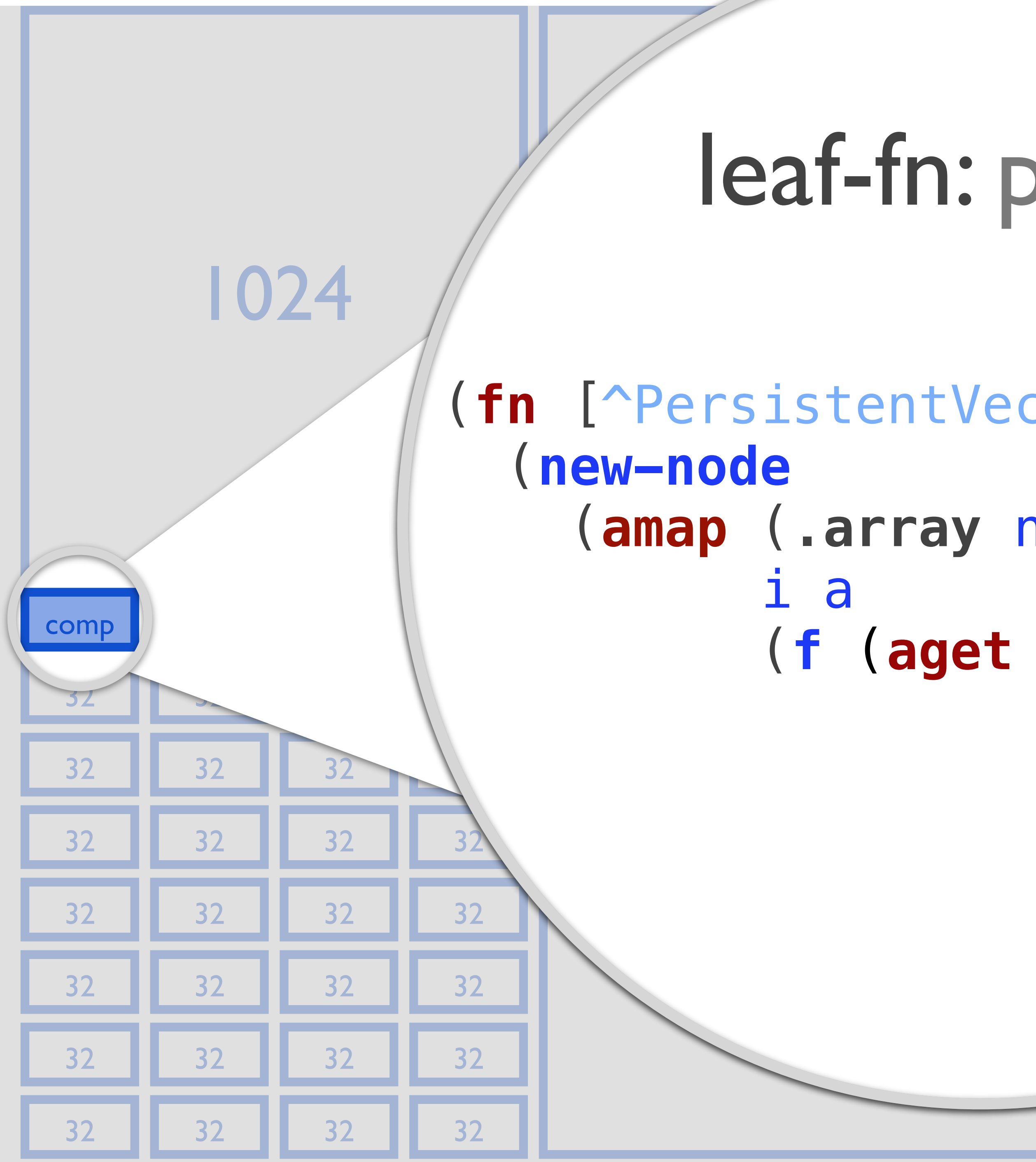


worker deques current tasks completed tasks

fjvtree

workers: 4   branching factor: 32   sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.



leaf-fn: pvmap

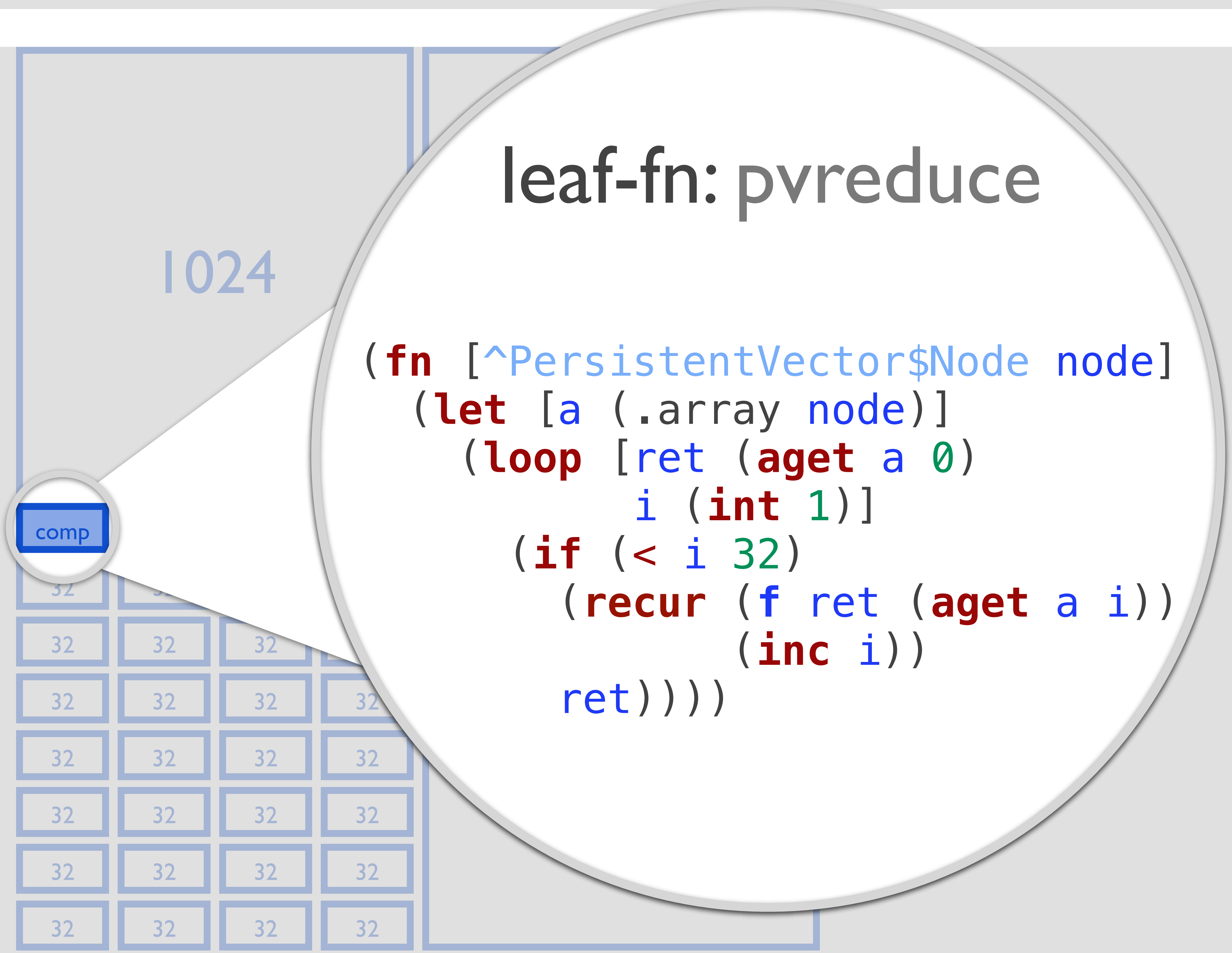
```
(fn [^PersistentVector$Node node]
  (new-node
    (amap (.array node)
          i a
          (f (aget a i)))))
```

worker deques   current tasks   completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.



leaf-fn: pvreduce

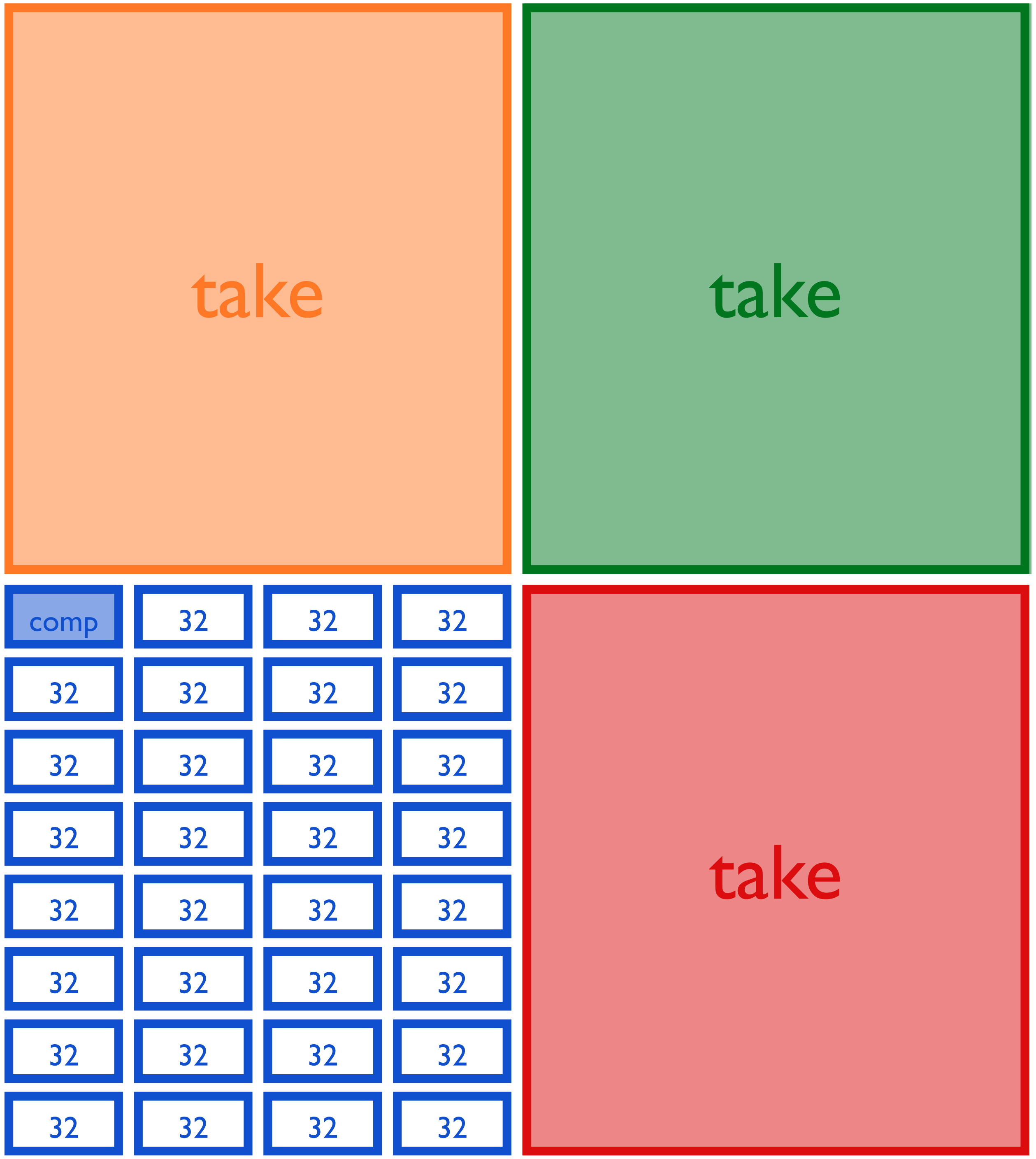
```
(fn [^PersistentVector$Node node]
  (let [a (.array node)]
    (loop [ret (aget a 0)
           i (int 1)]
      (if (< i 32)
        (recur (f ret (aget a i))
               (inc i))
        ret))))
```

worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

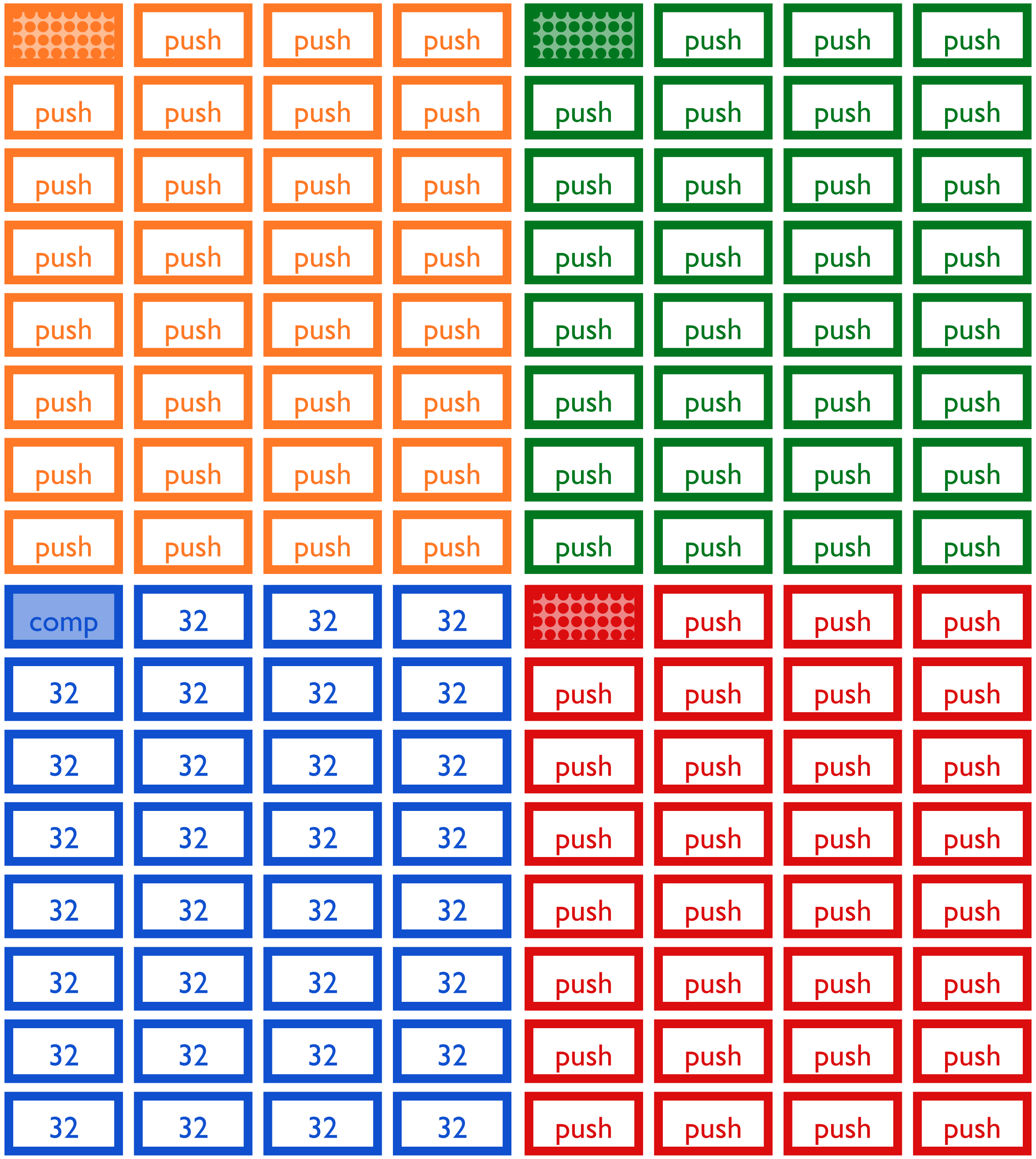


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

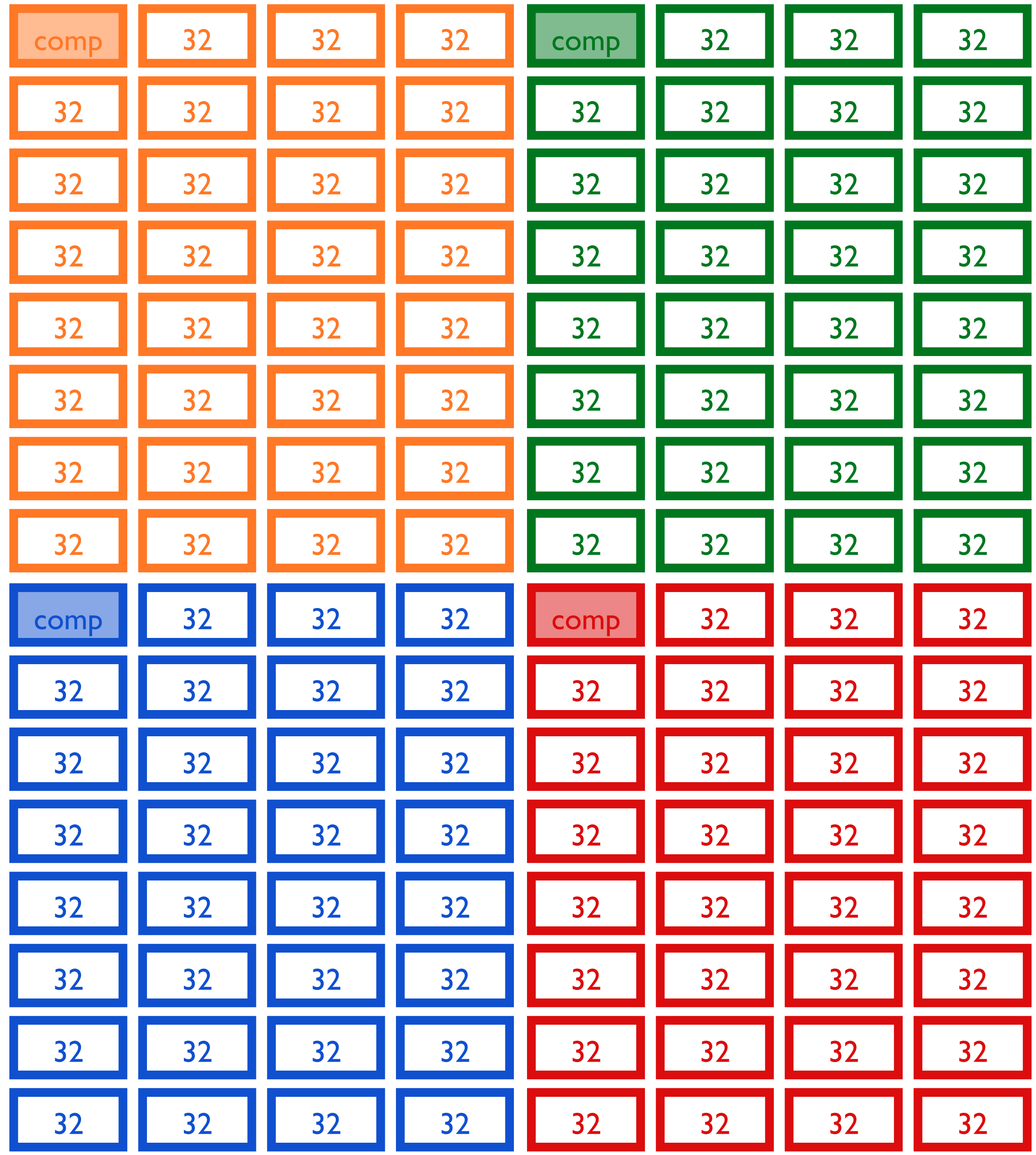


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

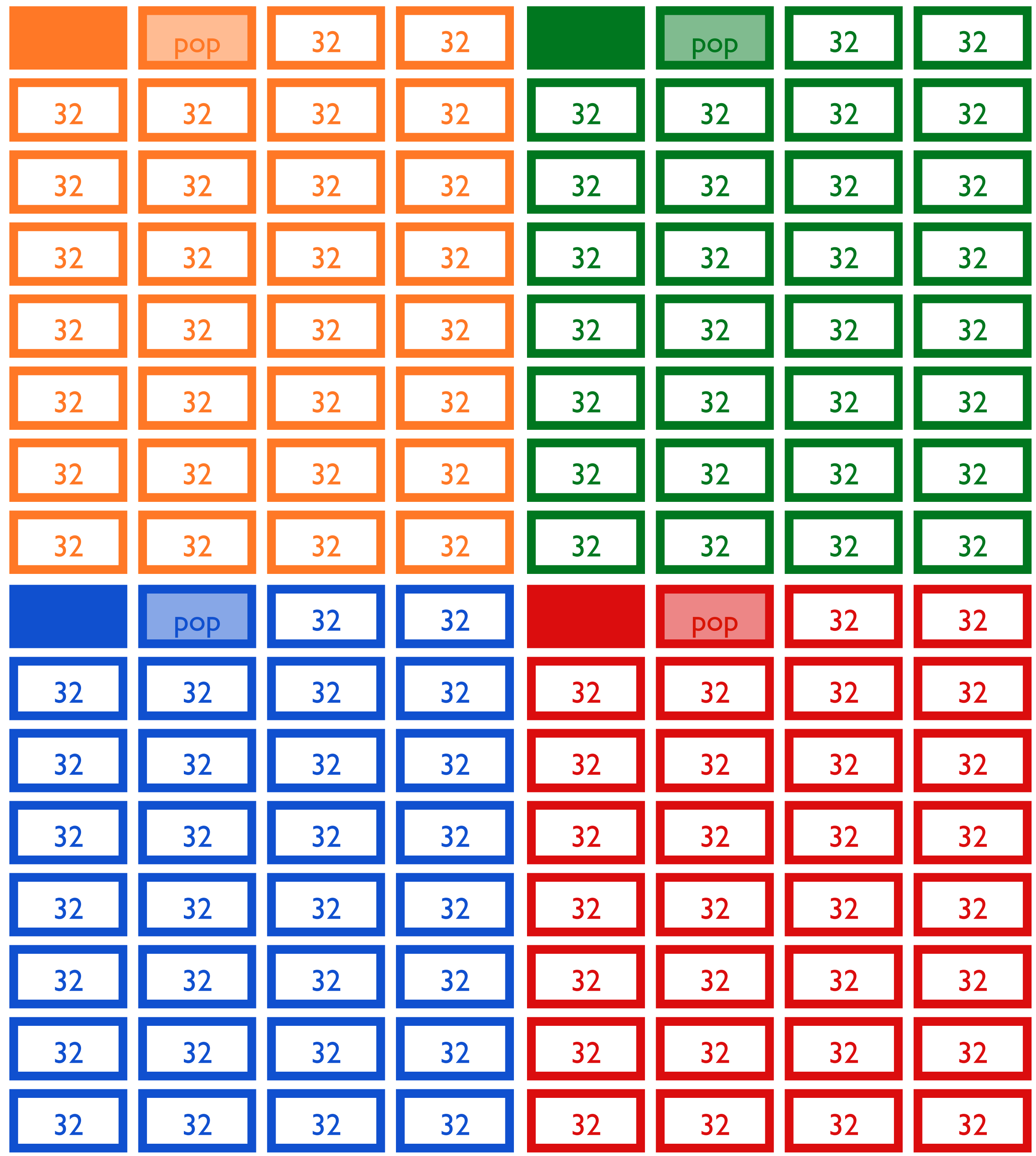


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

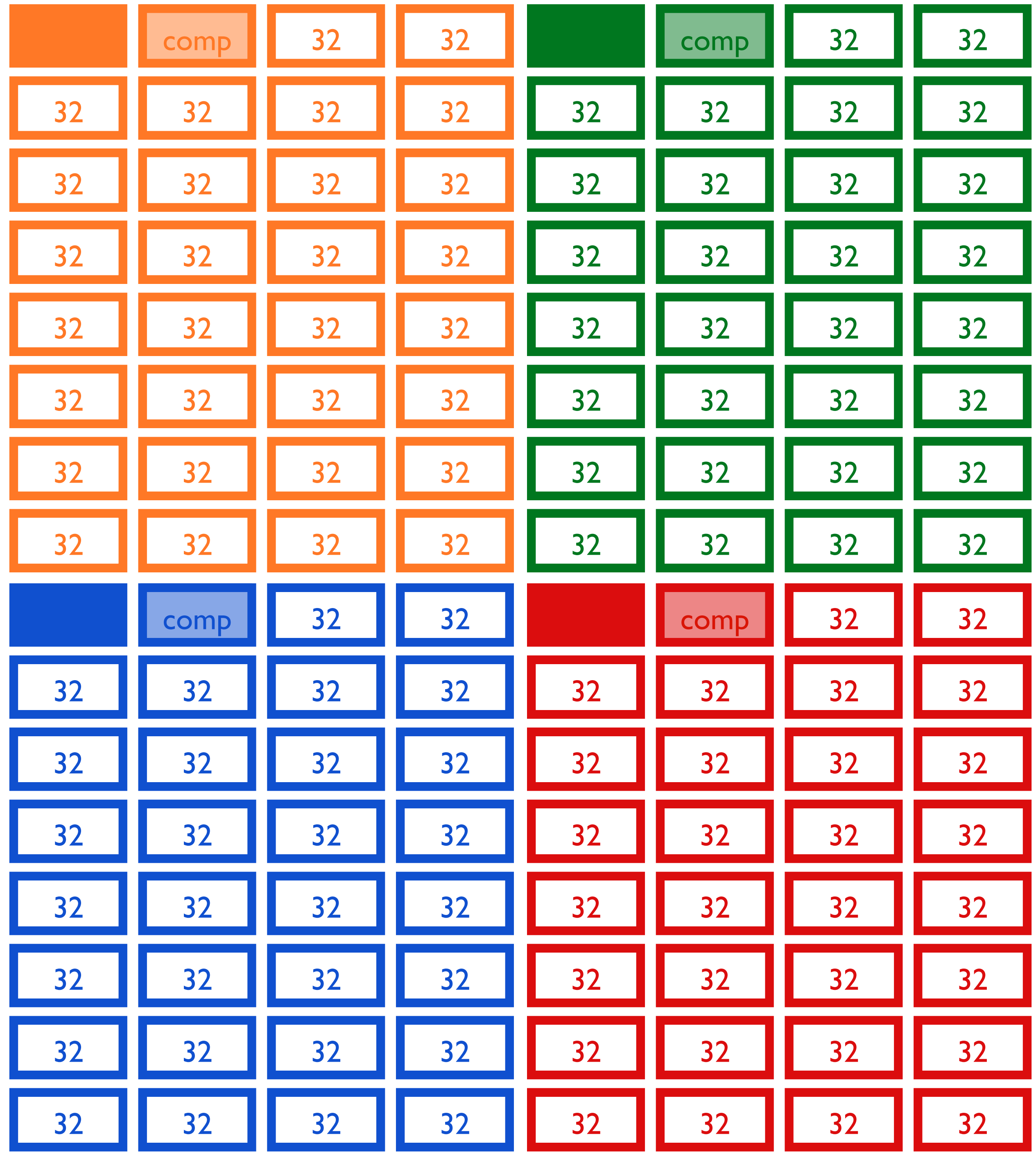


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

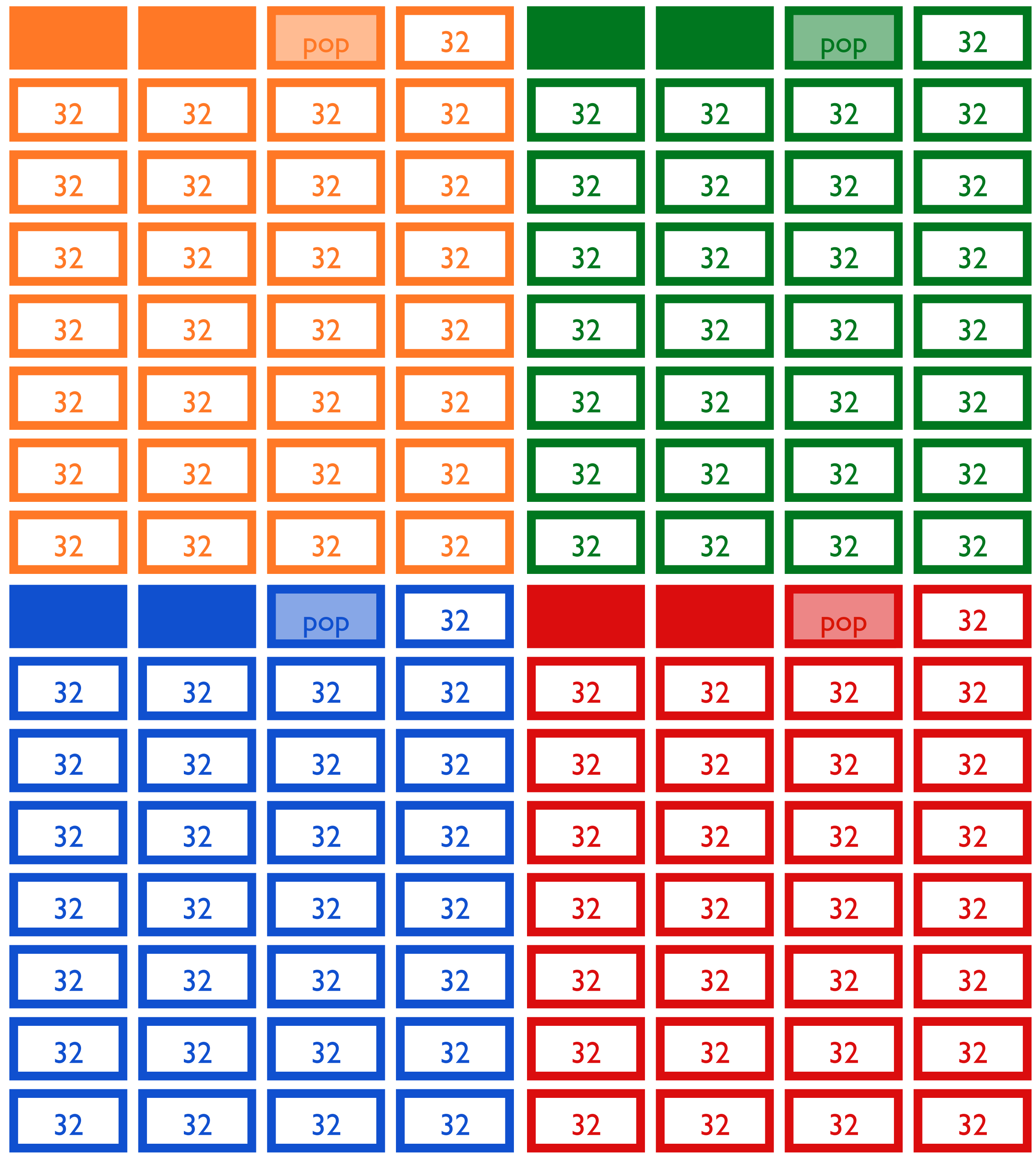


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

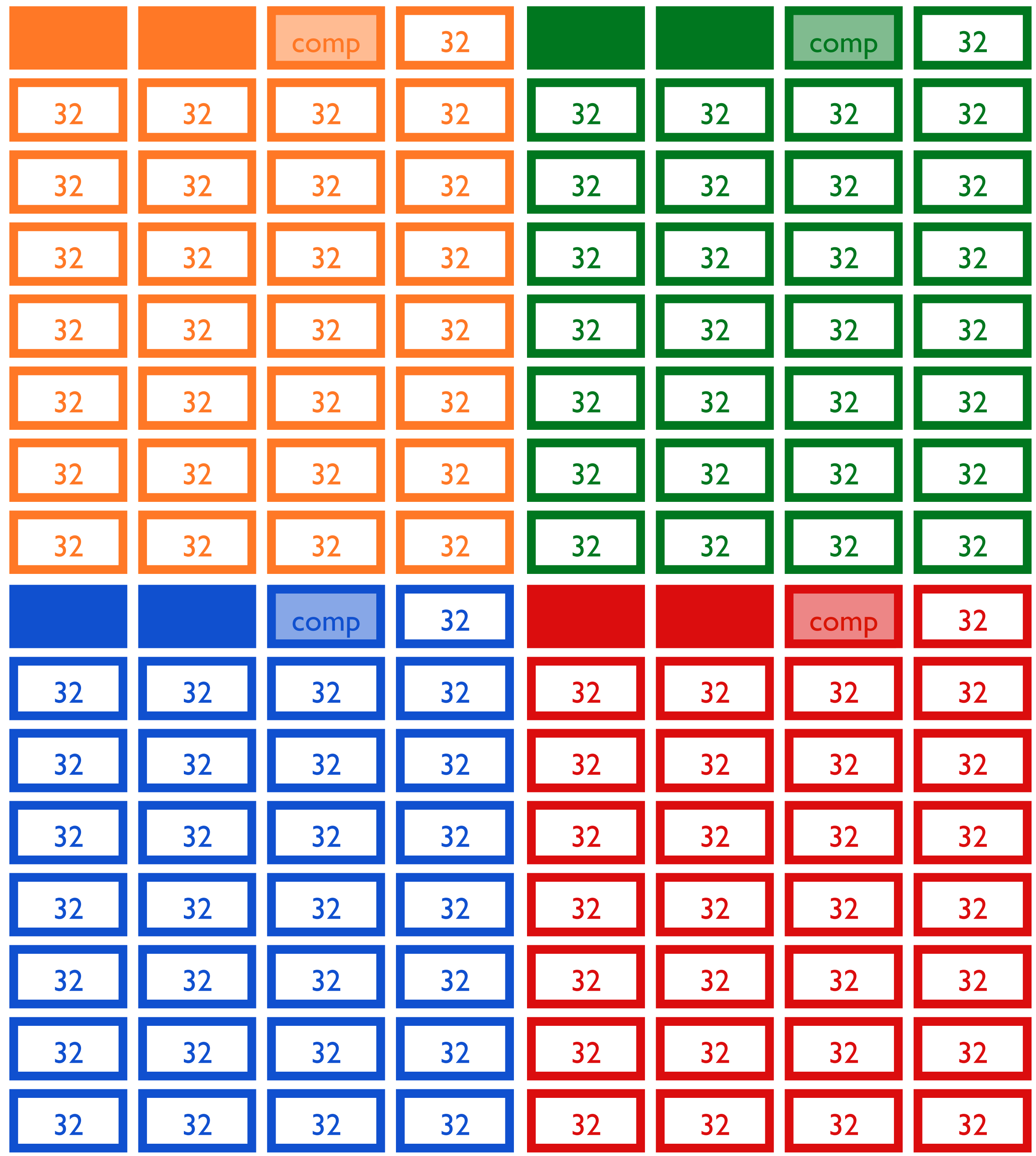


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

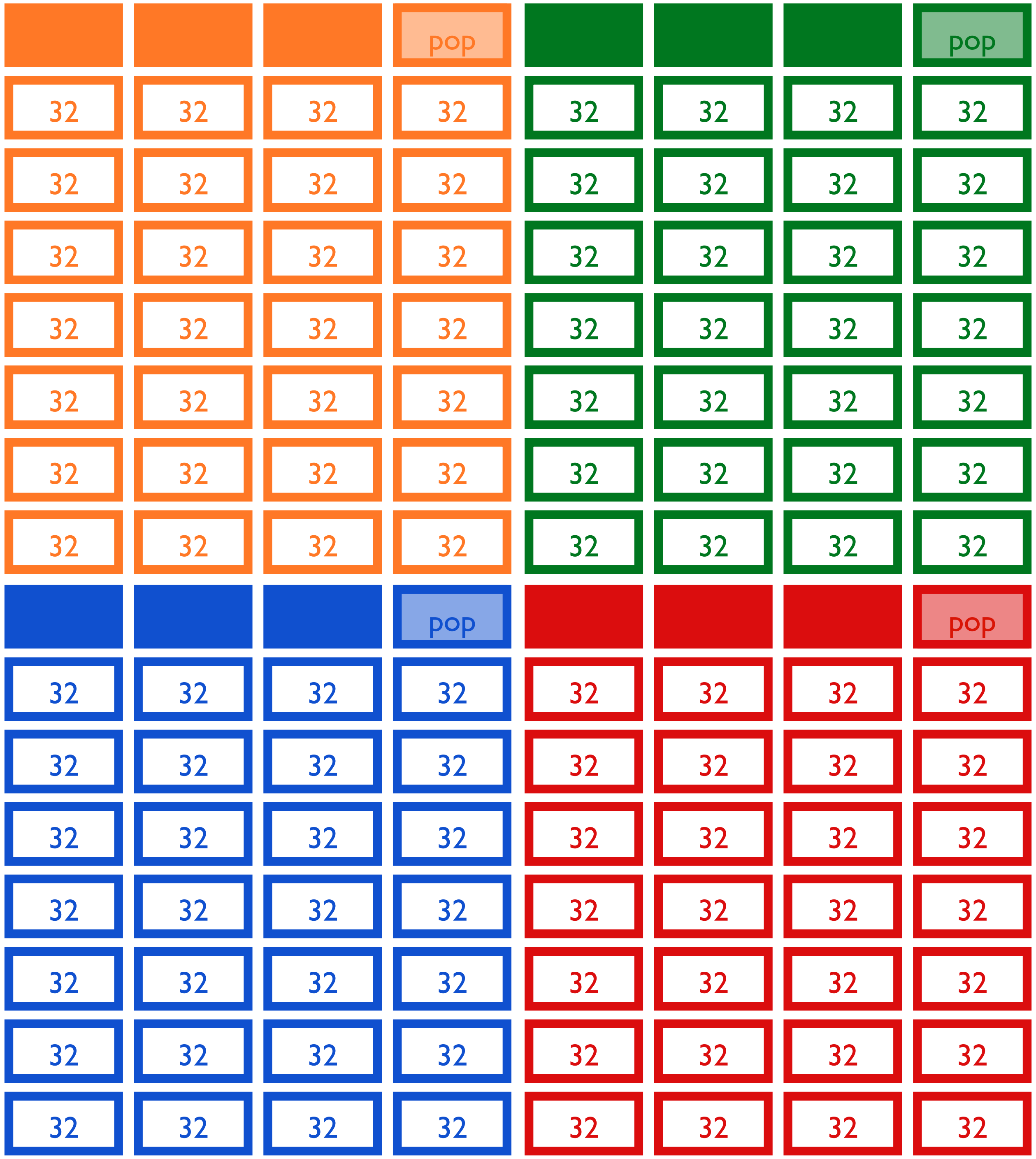


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

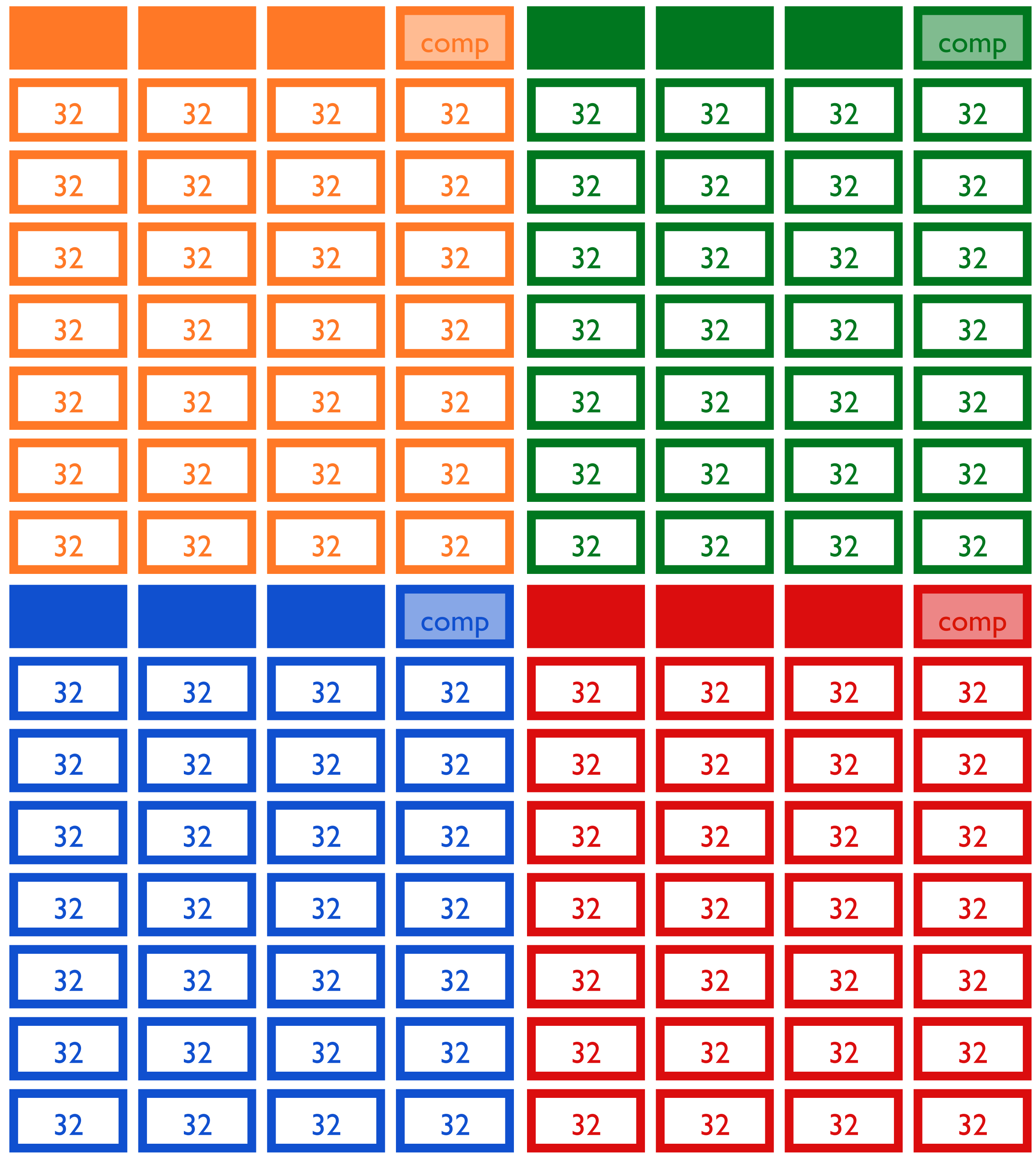


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

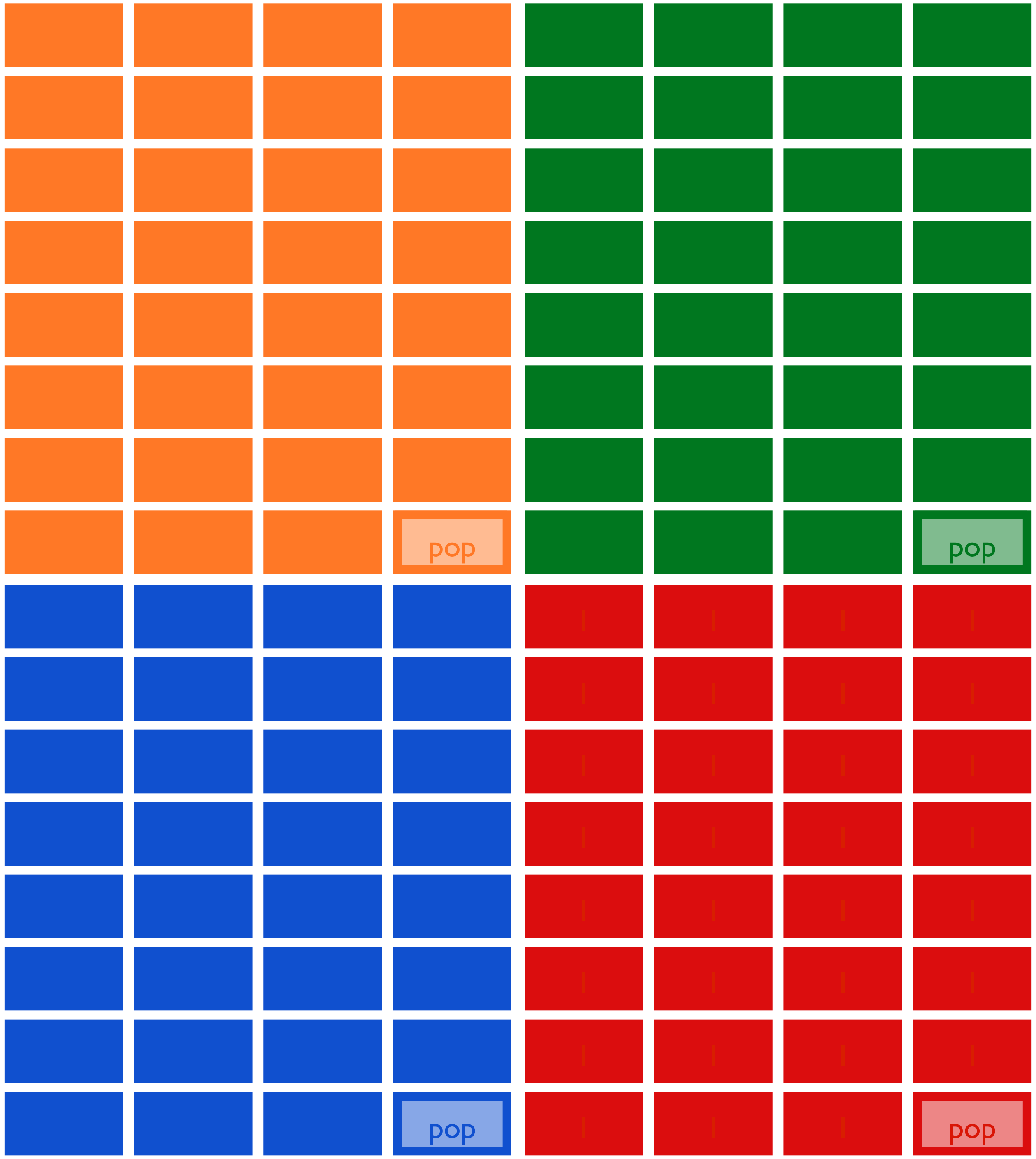


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

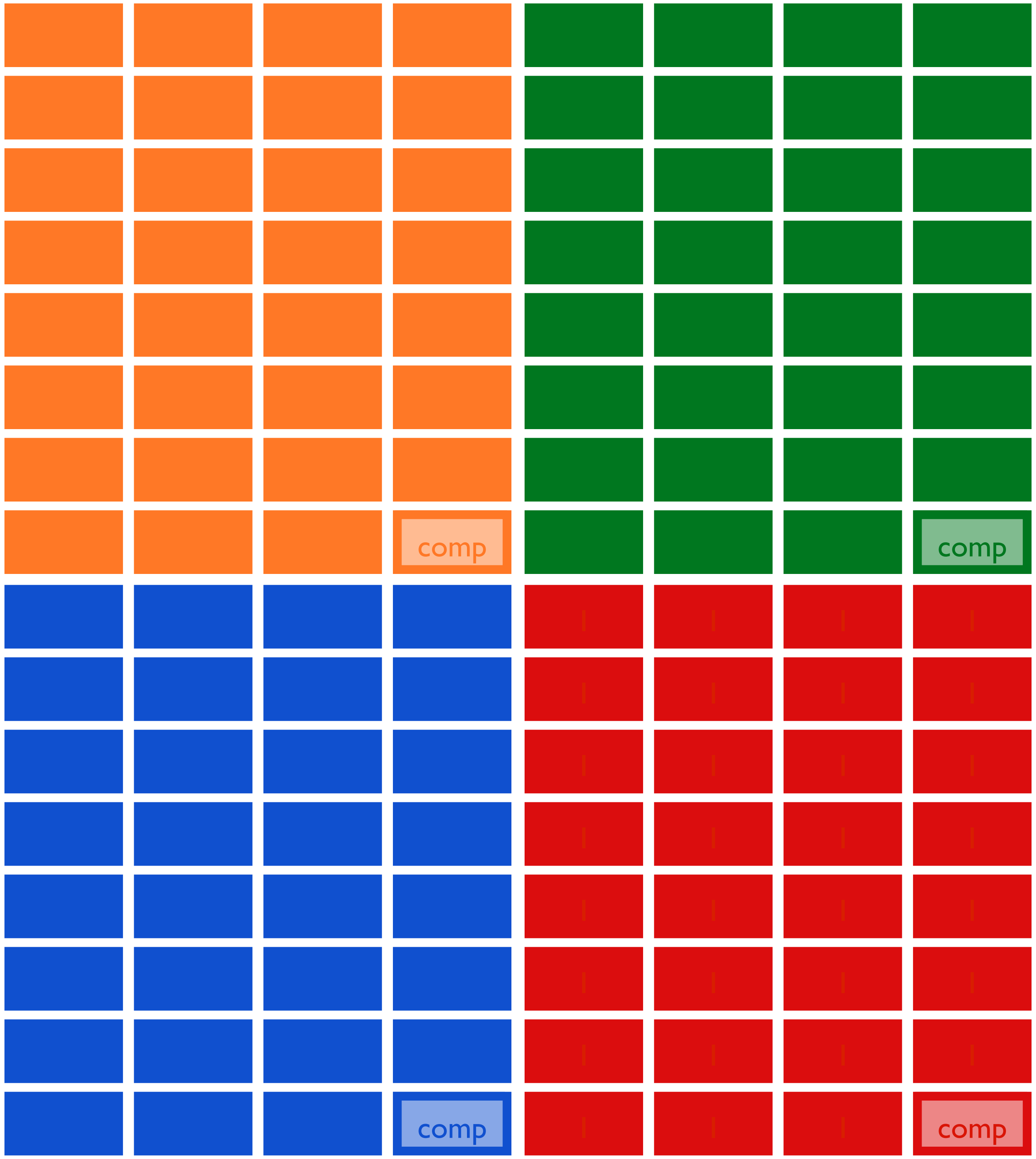


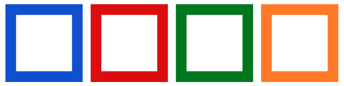
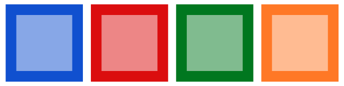
worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

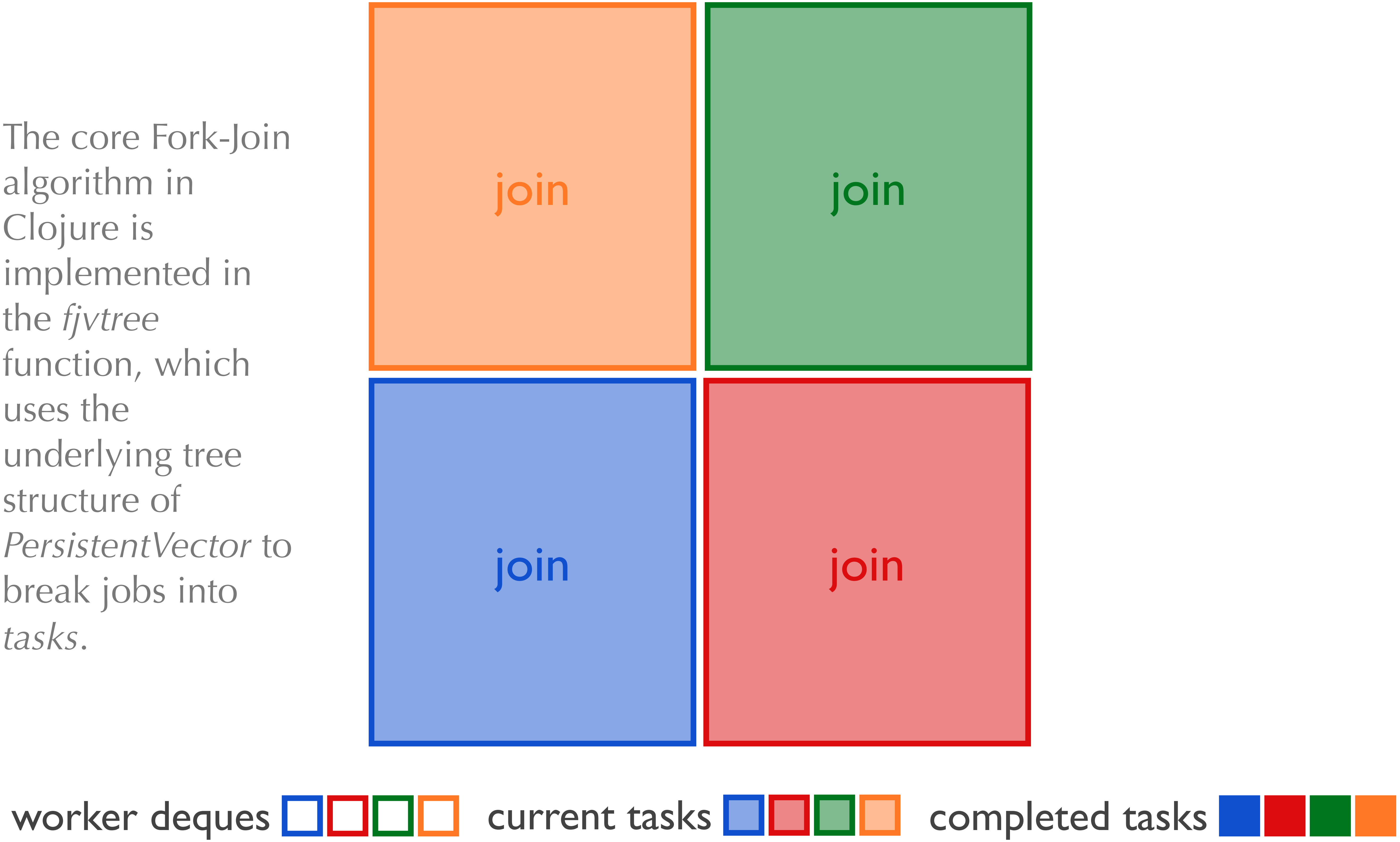


worker deques  current tasks  completed tasks 

fjvtree

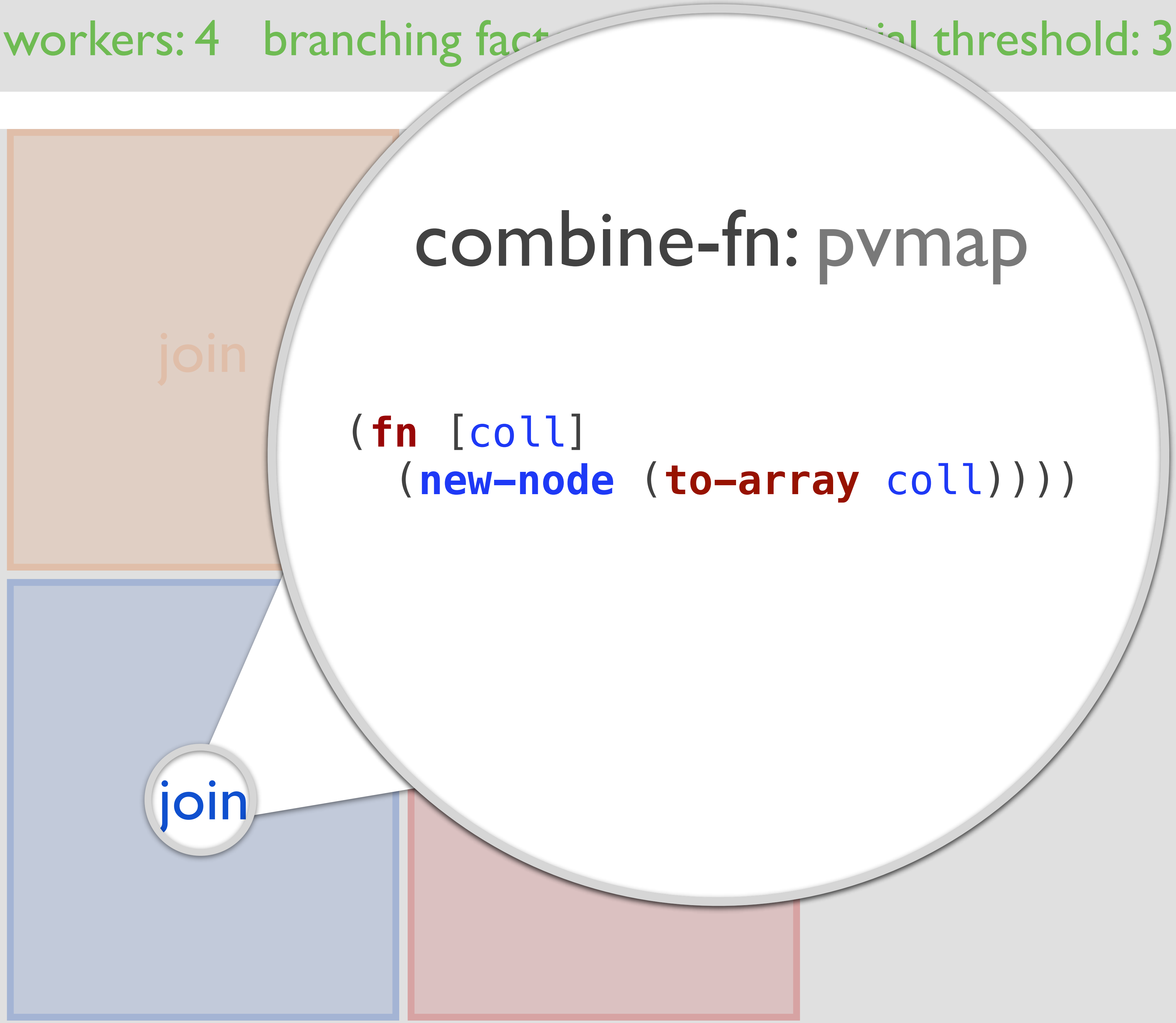
workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.



fjvtree   workers: 4   branching fact… …ial threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.



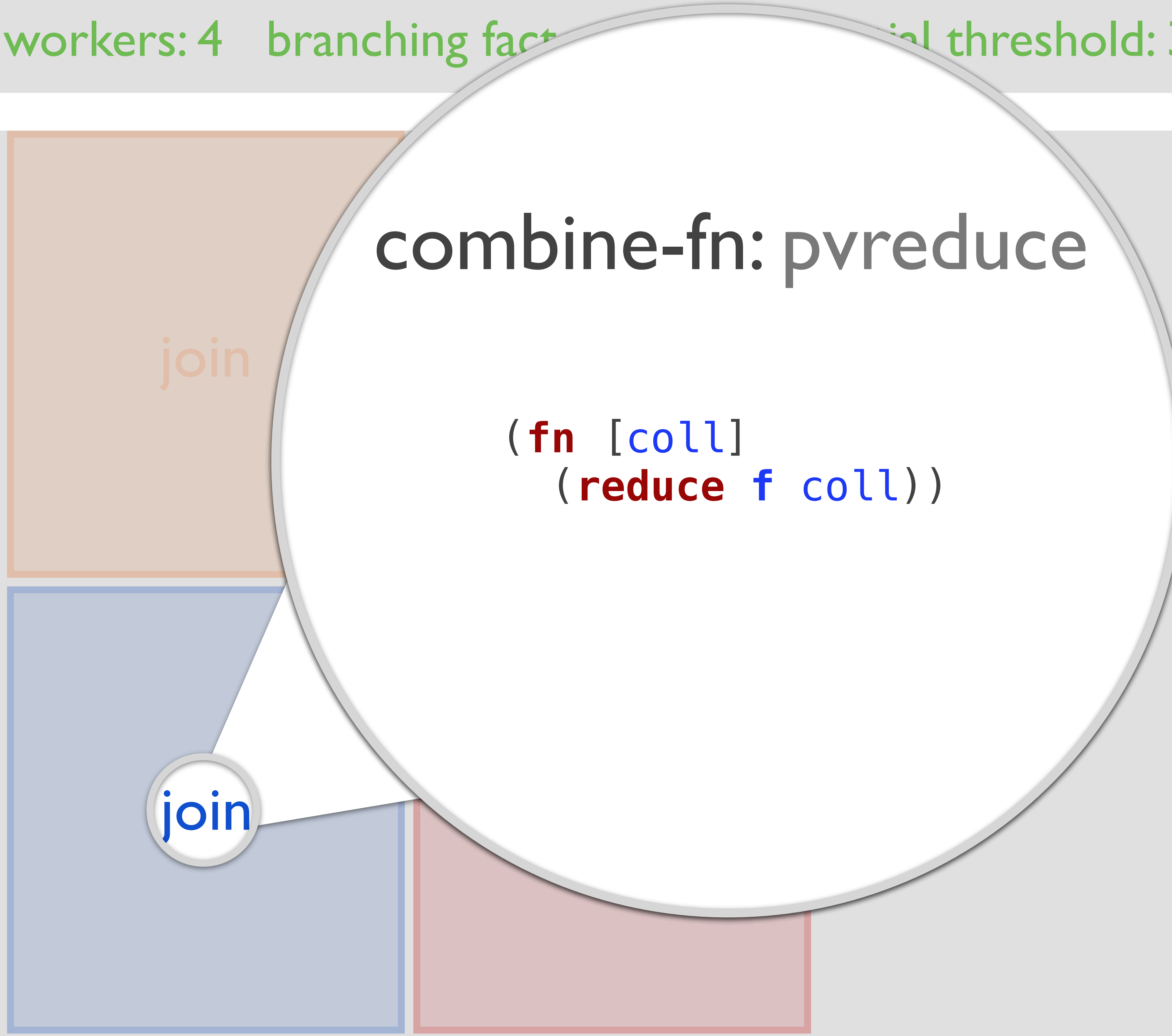

```
(fn [coll]
  (new-node (to-array coll)))
```

worker deques   current tasks   completed tasks

fjvtree

workers: 4 branching fact… …ial threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.



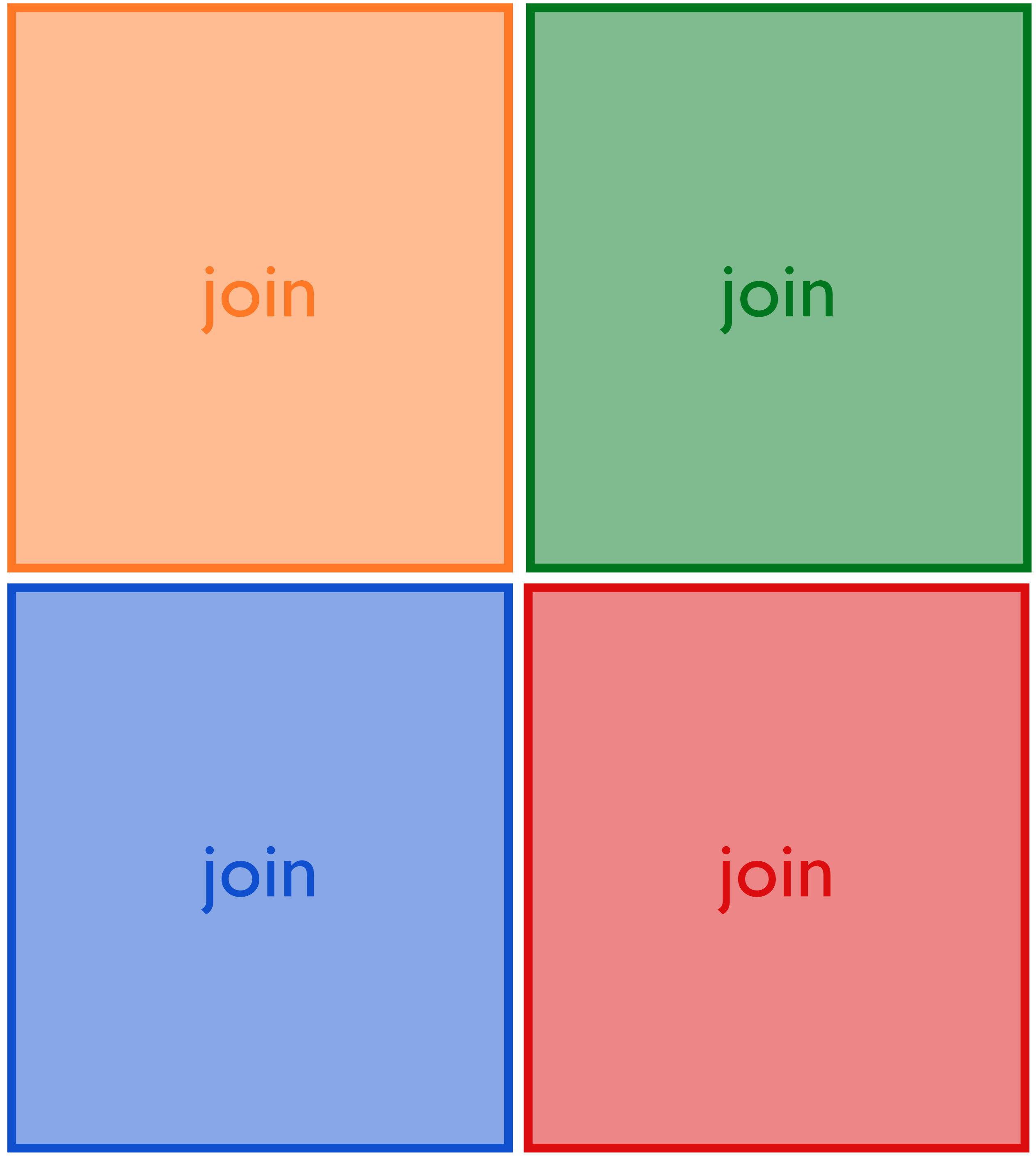

```
(fn [coll]
  (reduce f coll))
```

worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

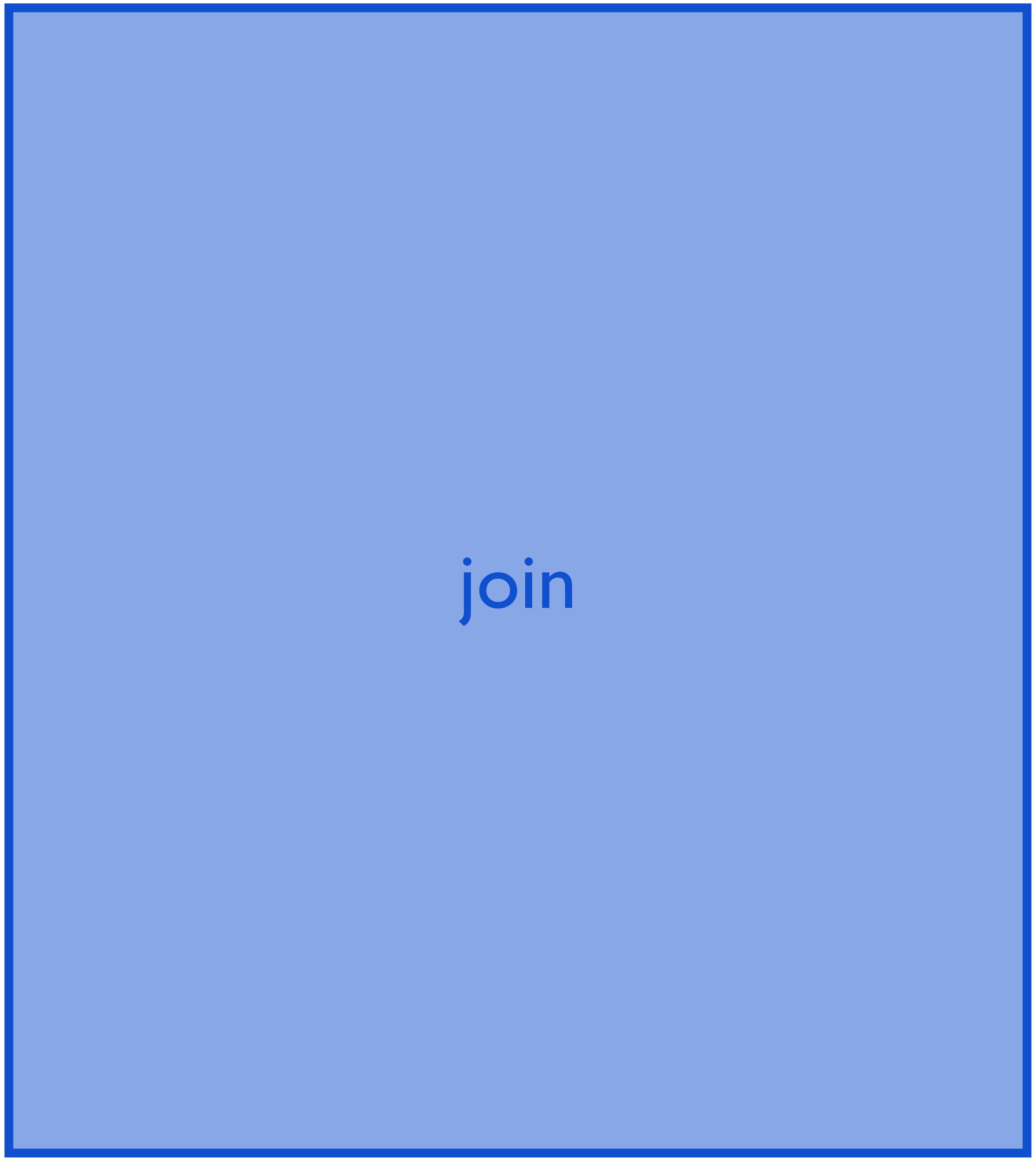


worker deques current tasks completed tasks

fjvtree

workers: 4 branching factor: 32 sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.

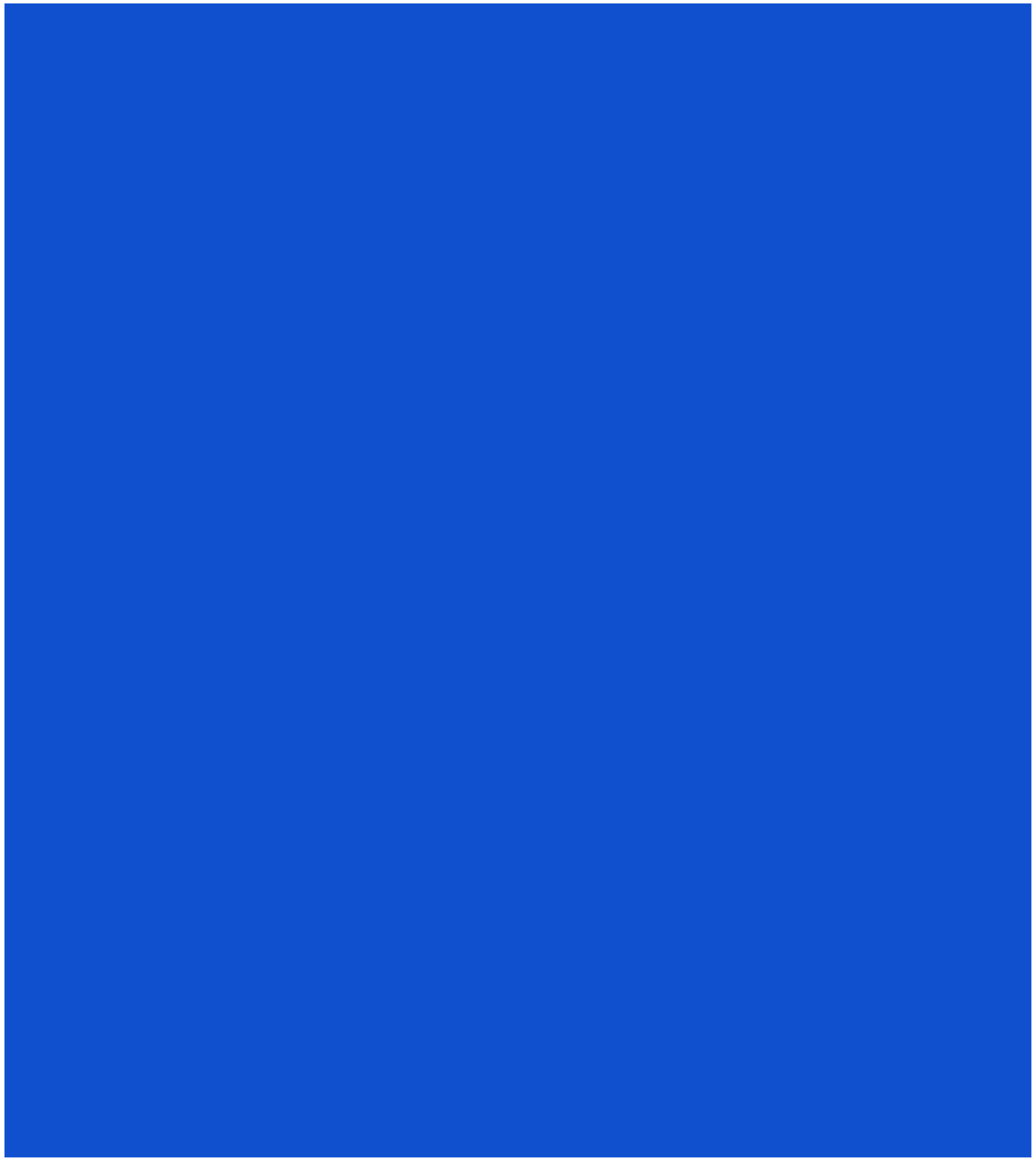


worker deques current tasks completed tasks

fjvtree

workers: 4   branching factor: 32   sequential threshold: 32

The core Fork-Join algorithm in Clojure is implemented in the *fjvtree* function, which uses the underlying tree structure of *PersistentVector* to break jobs into *tasks*.



worker deques   current tasks   completed tasks

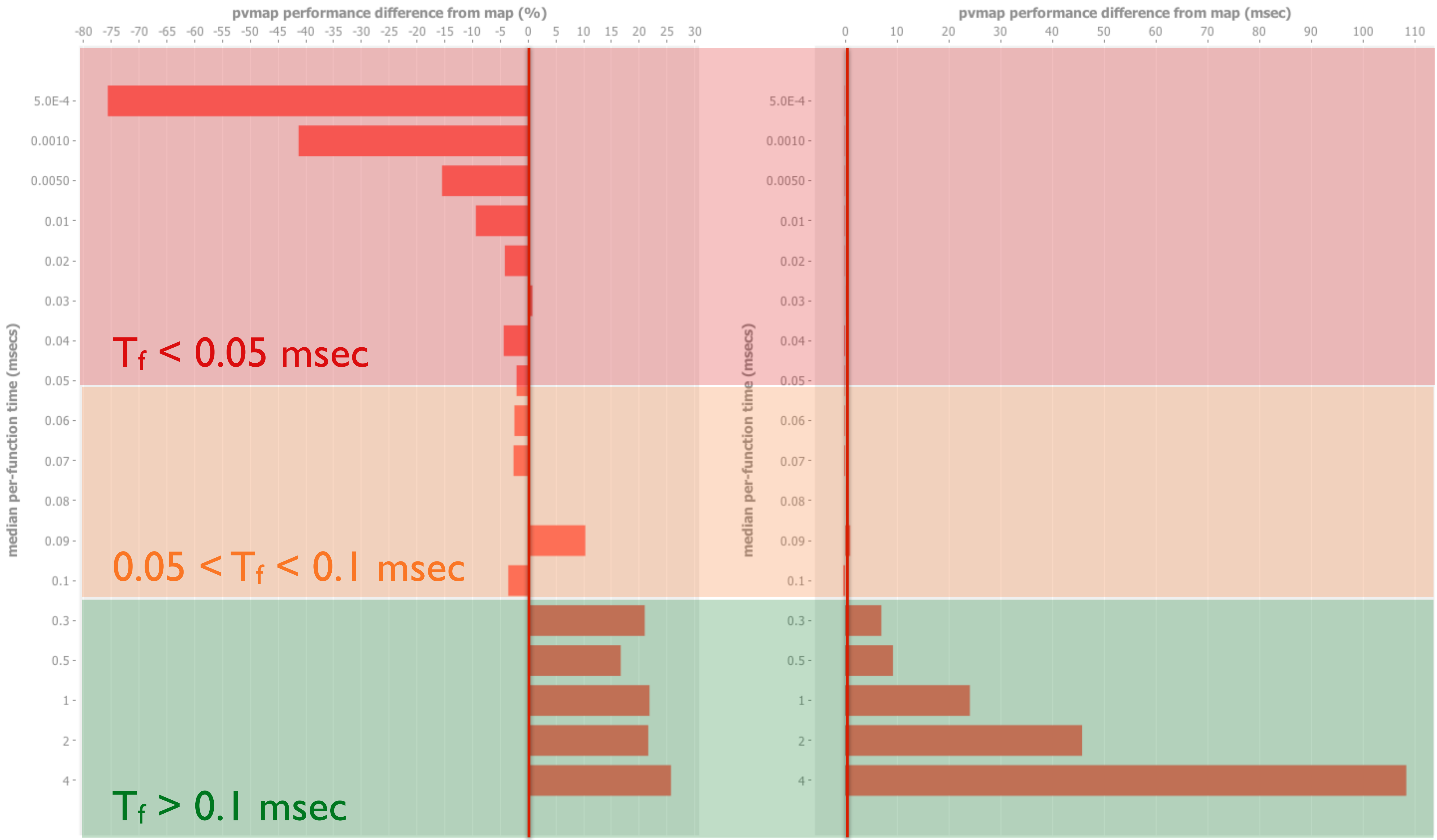
pvmap

performance characteristics

pvmap

compared to map (2 cores)

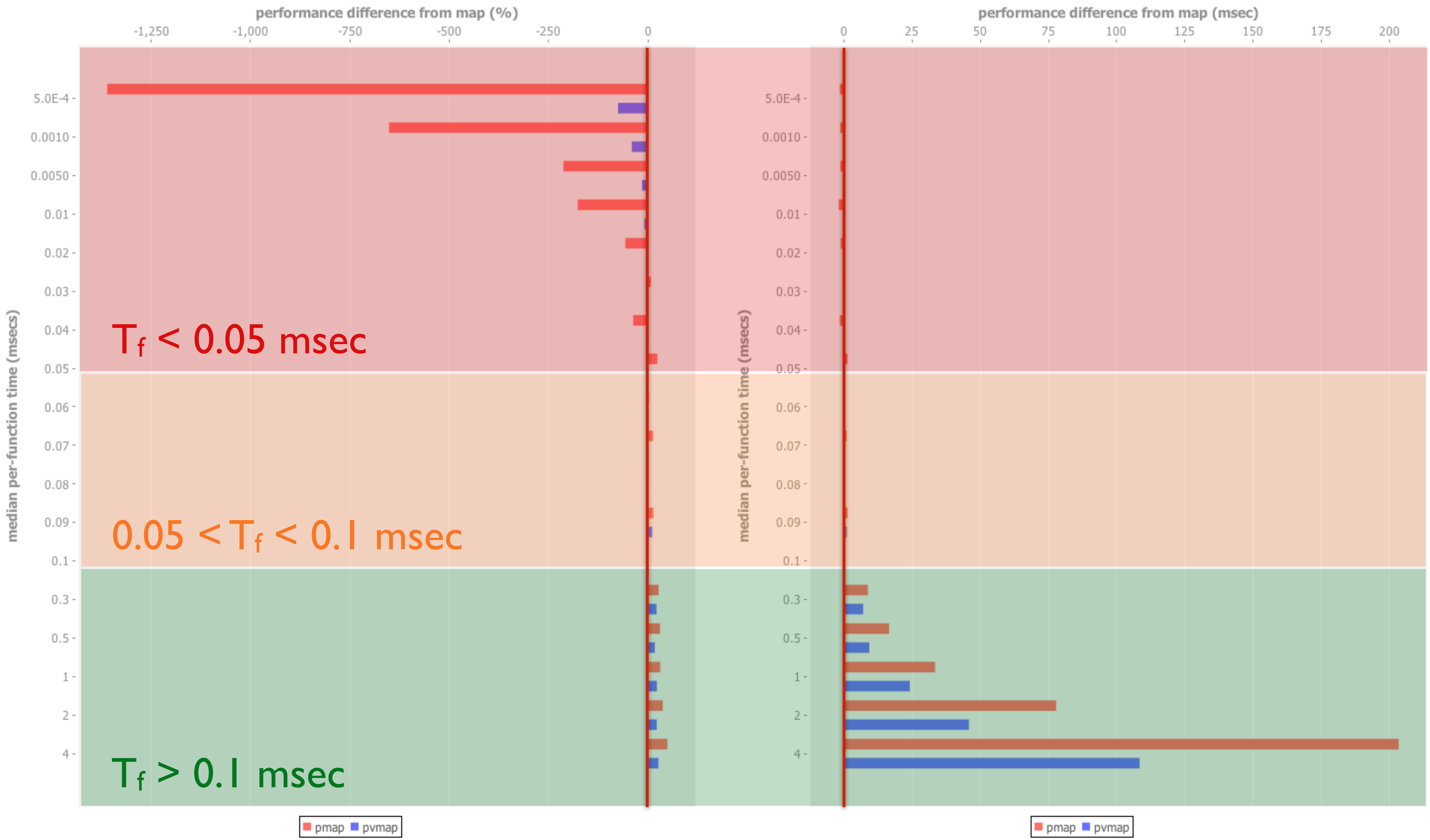
* results are the median of 50 samples, where a test function was pvmapped over a vector of 100 values



pvmap vs pmap

compared to map (2 cores)

* results are the median of 50 samples, where a test function was p*mapped over a vector of 100 values



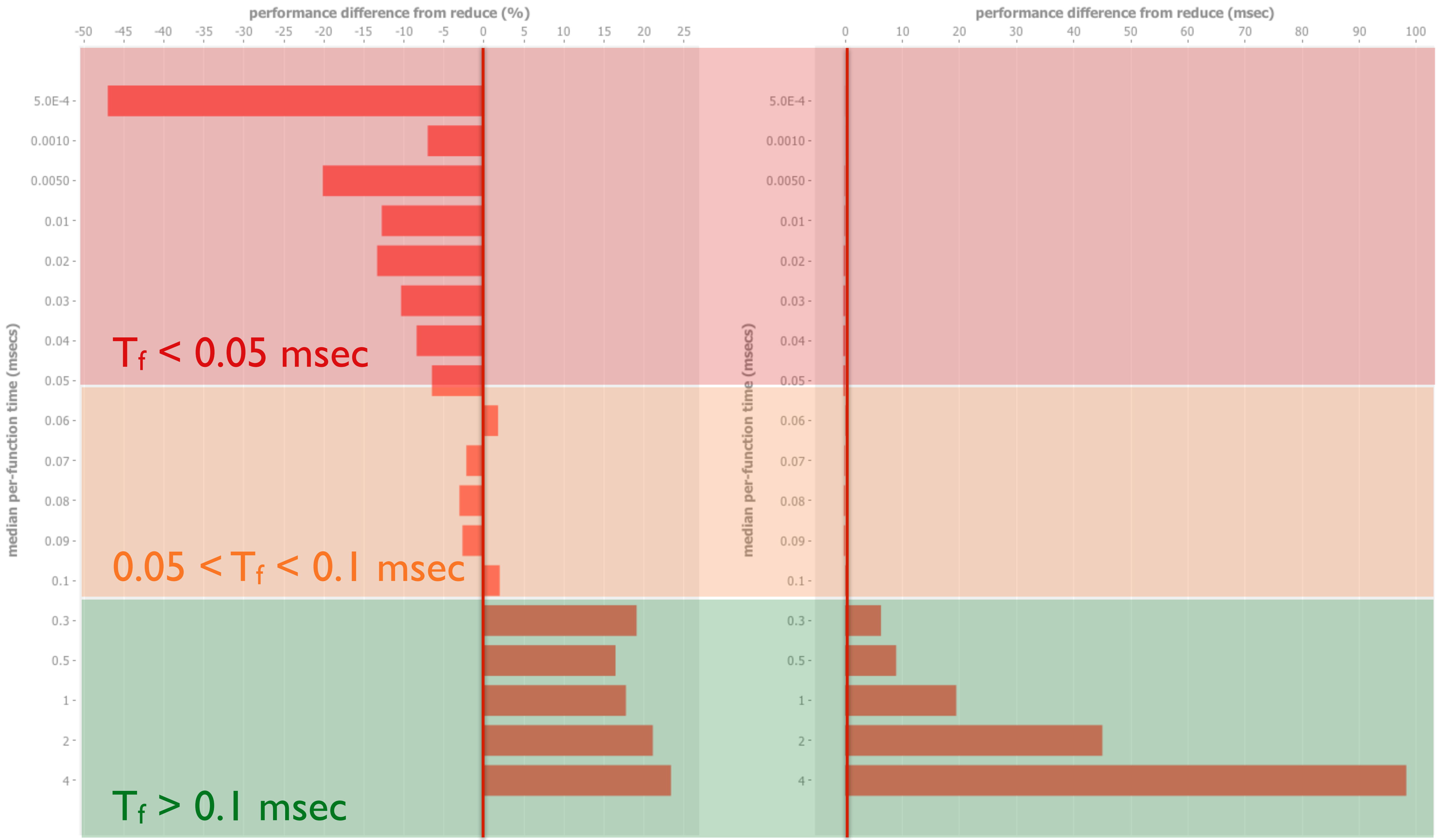
pvreduce

performance characteristics

pvreduce

compared to reduce (2 cores)

* results are the median of 50 samples, where a test function was pvreduced over a vector of 100 values



pvreduce examples

implementing pvfilter

parallel sum


```
(pvreduce + [1 2 3 4 5 6 7 8 9 .. 128])
```

parallel sum

using pvreduce

```
(pvreduce + [1 2 3 4 5 6 7 8 9 .. 128])
```

```
(reduce + (reduce + [1 2 3 4 5 .. 32])
          (reduce + [33 33 34 35 .. 64])
          (reduce + [65 66 67 68 .. 96])
          (reduce + [97 98 99 100 .. 128]))
```

parallel sum

using pvreduce

```
(pvreduce + [1 2 3 4 5 6 7 8 9 .. 128])

(reduce + (reduce + [1 2 3 4 5 .. 32])
          (reduce + [33 33 34 35 .. 64])
          (reduce + [65 66 67 68 .. 96])
          (reduce + [97 98 99 100 .. 128]))

(reduce + [528 1552 2576 3600])
```

parallel sum

using pvreduce

```
(pvreduce + [1 2 3 4 5 6 7 8 9 .. 128])

(reduce + (reduce + [1 2 3 4 5 .. 32])
          (reduce + [33 33 34 35 .. 64])
          (reduce + [65 66 67 68 .. 96])
          (reduce + [97 98 99 100 .. 128]))

(reduce + [528 1552 2576 3600])

8256
```

parallel filter

using pvreduce

```
(defn pvfilter [pred v]
  (letfn [(filt [v x] (if (pred x) (conj v x) v))]
    (pvreduce filt [] v)

(pvfilter even? [1 2 3 4 5 6 7 8 9 .. 128])
```

parallel filter

using pvreduce

```
(defn pvfilter [pred v]
  (letfn [(filt [v x] (if (pred x) (conj v x) v))]
    (pvreduce filt [] v)))

(pvfilter even? [1 2 3 4 5 6 7 8 9 .. 128])

(reduce filt (reduce filt [] [1 2 3 4 .. 32])
             (reduce filt [] [33 34 35 36 .. 64])
             (reduce filt [] [65 66 67 68 .. 96])
             (reduce filt [] [97 98 99 100 .. 128]))
```

parallel filter

using pvreduce

```
(defn pvfilter [pred v]
  (letfn [(filt [v x] (if (pred x) (conj v x) v))]
    (pvreduce filt [] v)))

(pvfilter even? [1 2 3 4 5 6 7 8 9 .. 128])

(reduce filt (reduce filt [] [1 2 3 4 .. 32])
             (reduce filt [] [33 34 35 36 .. 64])
             (reduce filt [] [65 66 67 68 .. 96])
             (reduce filt [] [97 98 99 100 .. 128]))

(reduce filt [[2 4 6 8 .. 32]
              [34 36 38 40 .. 64]
              [66 68 70 72 .. 96]
              [98 100 102 104 .. 128]])
```

parallel filter

using pvreduce

```
(defn pvfilter [pred v]
  (letfn [(filt [v x] (if (pred x) (conj v x) v))]
    (pvreduce filt [] v))

(pvfilter even? [1 2 3 4 5 6 7 8 9 .. 128])

(reduce filt (reduce filt [] [1 2 3 4 .. 32])
             (reduce filt [] [33 34 35 36 .. 64])
             (reduce filt [] [65 66 67 68 .. 96])
             (reduce filt [] [97 98 99 100 .. 128]))

(reduce filt [[2 4 6 8 .. 32]
              [34 36 38 40 .. 64]
              [66 68 70 72 .. 96]
              [98 100 102 104 .. 128]])

java.lang.ClassCastException: clojure.lang.PersistentVector cannot
be cast to java.lang.Number
```

parallel filter

using pvreduce

```
(defn pvfilter [pred v]
  (letfn [(par-filt [v x]
            (cond
              (vector? x) (apply reduce conj v x)
              (even? x) (conj v x)
              :else v))]
    (pvreduce par-filt [] v))

(pvfilter even? [1 2 3 4 5 6 7 8 9 .. 128])
```

parallel filter

using pvreduce

```
(defn pvfilter [pred v]
  (letfn [(par-filt [v x]
               (cond
                 (vector? x) (apply reduce conj v x)
                 (even? x) (conj v x)
                 :else v))]
    (pvreduce par-filt [] v))

(pvfilter even? [1 2 3 4 5 6 7 8 9 .. 128])

(reduce par-filt (reduce par-filt [] [1 2 3 4 .. 32])
                 (reduce par-filt [] [33 34 35 36 .. 64])
                 (reduce par-filt [] [65 66 67 68 .. 96])
                 (reduce par-filt [] [97 98 99 100 .. 128]))
```

parallel filter

using pvreduce

```
(defn pvfilter [pred v]
  (letfn [(par-filt [v x]
            (cond
              (vector? x) (apply reduce conj v x)
              (even? x) (conj v x)
              :else v))]
    (pvreduce par-filt [] v)))

(pvfilter even? [1 2 3 4 5 6 7 8 9 .. 128])

(reduce par-filt (reduce par-filt [] [1 2 3 4 .. 32])
                 (reduce par-filt [] [33 34 35 36 .. 64])
                 (reduce par-filt [] [65 66 67 68 .. 96])
                 (reduce par-filt [] [97 98 99 100 .. 128]))

(apply reduce conj [[2 4 6 8 .. 32]
                    [34 36 38 40 .. 64]
                    [66 68 70 72 .. 96]
                    [98 100 102 104 .. 128]])
```

parallel filter

using pvreduce

```
(defn pvfilter [pred v]
  (letfn [(par-filt [v x]
            (cond
              (vector? x) (apply reduce conj v x)
              (even? x) (conj v x)
              :else v))]
    (pvreduce par-filt [] v)))

(pvfilter even? [1 2 3 4 5 6 7 8 9 .. 128])
```

```
(reduce par-filt (reduce par-filt [] [1 2 3 4 .. 32])
                 (reduce par-filt [] [33 34 35 36 .. 64])
                 (reduce par-filt [] [65 66 67 68 .. 96])
                 (reduce par-filt [] [97 98 99 100 .. 128]))
```

```
(apply reduce conj [[2 4 6 8 .. 32]
                    [34 36 38 40 .. 64]
                    [66 68 70 72 .. 96]
                    [98 100 102 104 .. 128]])
```

```
[2 4 6 8 .. 128]
```

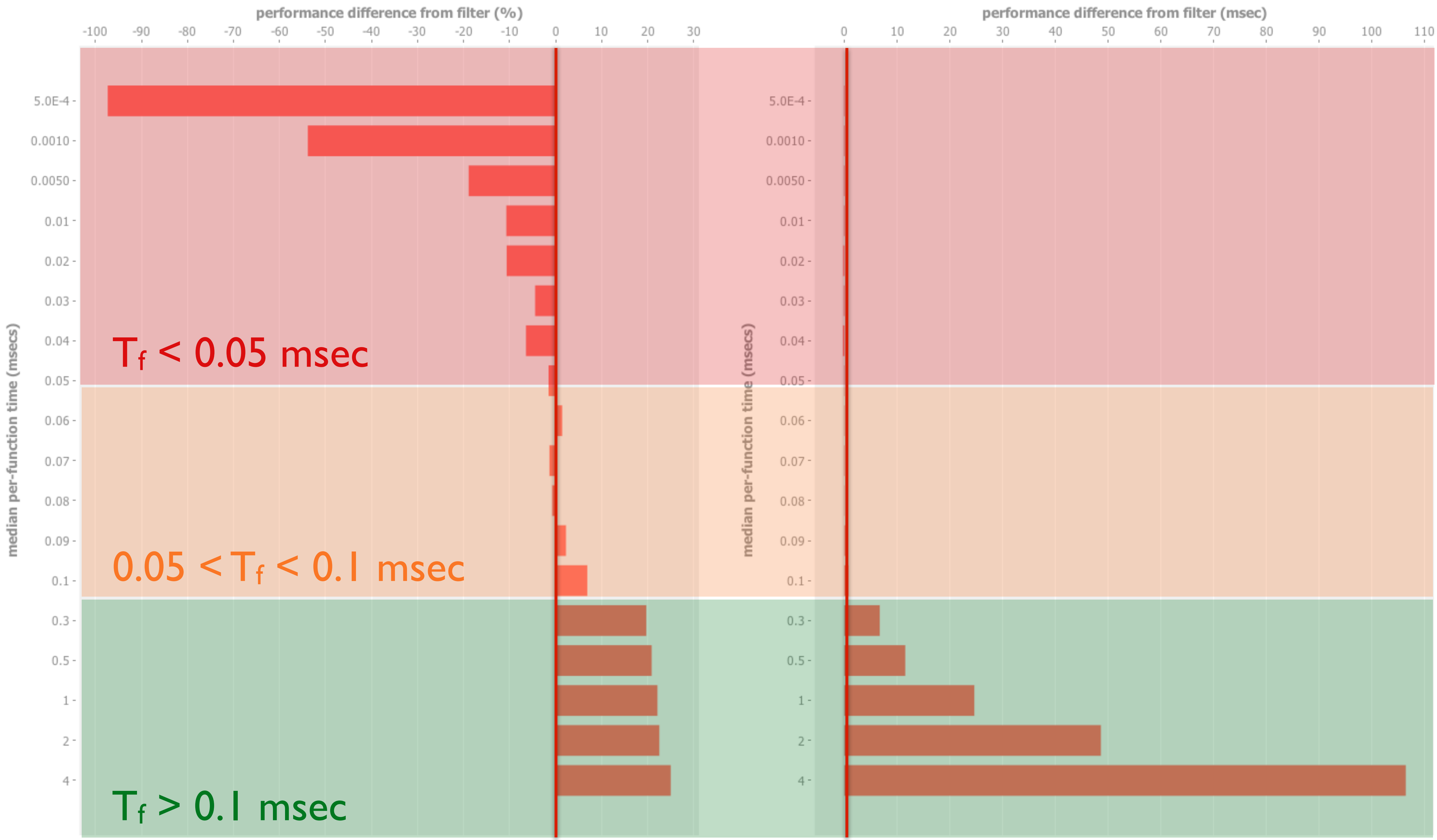
pvfilter

performance characteristics

pvfilter

compared to filter with 2 cores

* results are the median of 50 samples, where a test predicate was pvfiltered over a vector of 100 values



references

1. Brian Goetz Fork/Join talk: http://www.infoq.com/presentations/brian-goetz-concurrent-parallel
2. Fork/Join API: http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/
3. Developer's Works article: http://www.ibm.com/developerworks/java/library/j-jtp11137.html
4. Java Concurrency Wiki: http://artisans-serverintellect-com.si-eioswww6.com/default.asp?W1
5. JVM summit slides: http://wiki.jvmlangsummit.com/images/f/f0/Lea-fj-jul10.pdf
6. Fork/Join Paper: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.42.1918&rep=rep1&type=pdf

questions?

thank you

parallel frequency

using pvreduce

```
(defn pvfreq [x]
  (letfn [(par-freq [m x]
             (if (map? x)
               (merge-with #(+ %1 %2) m x)
               (update-in m [x] #(if % 1 (inc %)))))]
    (pvreduce par-freq {} x)

(pvfreq [:foo :bar :baz :bar .. :foo])
```

parallel frequency

using pvreduce

```
(defn pvfreq [x]
  (letfn [(par-freq [m x]
            (if (map? x)
              (merge-with #(+ %1 %2) m x)
              (update-in m [x] #(if % 1 (inc %)))))]
    (pvreduce par-freq {} x)))

(pvfreq [:foo :bar :baz :bar .. :foo])
```

```
(reduce par-freq (reduce par-freq {} [:foo :foo .. :bar])
                 (reduce par-freq {} [:bar :foo .. :foo])
                 (reduce par-freq {} [:baz :bar .. :foo])
                 (reduce par-freq {} [:bar :baz .. :baz]))
```

parallel frequency

using pvreduce

```
(defn pvfreq [x]
  (letfn [(par-freq [m x]
            (if (map? x)
              (merge-with #(+ %1 %2) m x)
              (update-in m [x] #(if % 1 (inc %)))))]
    (pvreduce par-freq {} x)))

(pvfreq [:foo :bar :baz :bar .. :foo])

(reduce par-freq (reduce par-freq {} [:foo :foo .. :bar])
                 (reduce par-freq {} [:bar :foo .. :foo])
                 (reduce par-freq {} [:baz :bar .. :foo])
                 (reduce par-freq {} [:bar :baz .. :baz]))

(reduce par-freq [{:foo 15, :bar 10, :baz 7}
                  {:foo 10, :bar 12, :baz 5}
                  {:foo 7, :bar 14, :baz 11}
                  {:foo 12, :bar 10, :baz 10}])
```

parallel frequency

using pvreduce

```
(defn pvfreq [x]
  (letfn [(par-freq [m x]
            (if (map? x)
              (merge-with #(+ %1 %2) m x)
              (update-in m [x] #(if % 1 (inc %)))))]
    (pvreduce par-freq {} x)))

(pvfreq [:foo :bar :baz :bar .. :foo])
```

```
(reduce par-freq (reduce par-freq {} [:foo :foo .. :bar])
                 (reduce par-freq {} [:bar :foo .. :foo])
                 (reduce par-freq {} [:baz :bar .. :foo])
                 (reduce par-freq {} [:bar :baz .. :baz]))
```

```
(reduce par-freq [{:foo 15, :bar 10, :baz 7}
                  {:foo 10, :bar 12, :baz 5}
                  {:foo 7, :bar 14, :baz 11}
                  {:foo 12, :bar 10, :baz 10}])
```

```
{:foo 44, :bar 46, :baz 33}
```