From Concurrency to Parallelism

an illustrated guide to multi-core parallelism in Clojure

David Edgar Liebke

clojureconj

Durham, NC
23 October 2010
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
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.
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.
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.
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.
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.
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.
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.
The \textit{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 \textit{futures} to invoke the function being mapped on just enough input values, but no more.
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.
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.
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.
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.
pmap

when lazy isn’t parallel
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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
pmap
uneven loads
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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
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

$T_f < 0.05 \text{ msec}$

$0.05 < T_f < 0.1 \text{ msec}$

$T_f > 0.1 \text{ msec}$
pmap

chunky style
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 \textit{pmap} instead of individual values.
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 (fn [chunk] (map f chunk)) (partition 32 coll))
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.
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.
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.
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.
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 \textit{pmap} instead of individual values.
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.
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

chunk < 10

10 < chunk < 50

T_f > 50
fork-join parallelism

divide and conquer
deque

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.
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.
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.
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.
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.
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.
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.
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.
fork-join

basic algorithm
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.
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.
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.
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.
As one worker is processing its tasks, another worker can steal a task from the back of its dequeue.
As one worker is processing its tasks, another worker can steal a task from the back of its dequeue.
As one worker is processing its tasks, another worker can steal a task from the back of its dequeue.
As one worker is processing its tasks, another worker can steal a task from the back of its dequeue.
As one worker is processing its tasks, another worker can steal a task from the back of its dequeue.
Once each worker’s current task is smaller than the configured threshold, it will begin the intended computation.
Once the computation is completed for the current task, the worker will pop another task off of its dequeue...
... and perform the computation on it.
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...

worker deques  current tasks  completed tasks

compute  push

256  512  push  push
fork-join

workers: 4  branching factor: 2  sequential threshold: 256

And so on...

worker deques  current tasks  completed tasks

compute  compute  compute  compute
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
If one of the workers falls behind, another worker can take tasks from its dequeue.
If one of the workers falls behind, another worker can take tasks from its dequeue.
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.
If one of the workers falls behind, another worker can take tasks from its dequeue.
If one of the workers falls behind, another worker can *take* tasks from its dequeue.
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 □□□□
... until the job is complete.
persistent vector

parallelism with existing data structures
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.
persistent-vector

count: 32
shift: 5

Values are added to the tail...
**persistent-vector**

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

count: 4097
shift: 10

And so on...
fork-join on persistent vectors

fjvtree, pvmap, and pvreduce
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.
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

\[
\begin{align*}
(pvmap \ f \ v) & \quad (pvreduce \ f \ v)
\end{align*}
\]

implemented with

\[
(fjvtree \ v \ combine-fn \ leaf-fn)
\]
The core Fork-Join algorithm in Clojure is implemented in the \texttt{fjvtree} function, which uses the underlying tree structure of \texttt{PersistentVector} to break jobs into tasks.
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.
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.
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.
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.

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

`leaf-fn: pvmap`
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
The core Fork-Join algorithm in Clojure is implemented in the \texttt{fjvtree} function, which uses the underlying tree structure of \texttt{PersistentVector} to break jobs into tasks.

\texttt{worker deques} \quad \texttt{current tasks} \quad \texttt{completed tasks}
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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
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 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.

```clojure
(fnj [coll]
  (new-node (to-array coll)))
```
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 factor: 32
- sequential threshold: 32

**combine-fn: pvreduce**

```
(fraft  [coll]
  (reduce f coll))
```
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.
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.
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.
pvmap

performance characteristics
compared to map (2 cores)

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

- $T_f < 0.05$ msec
- $0.05 < T_f < 0.1$ msec
- $T_f > 0.1$ msec
pvmap vs pmap

compared to map (2 cores)

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

\[ T_f < 0.05 \text{ msec} \]

\[ 0.05 < T_f < 0.1 \text{ msec} \]

\[ T_f > 0.1 \text{ msec} \]
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

\[ T_f < 0.05 \text{ msec} \]

\[ 0.05 < T_f < 0.1 \text{ msec} \]

\[ T_f > 0.1 \text{ msec} \]
pvreduce examples

implementing pvfilter
parallel sum

\[(\text{pvreduce} + [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ .. \ 128])\]
parallel sum

\[(\text{pvreduce} + [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ .. \ 128])\]

\[(\text{reduce} + (\text{reduce} + [1 \ 2 \ 3 \ 4 \ 5 \ .. \ 32]))\]
\[(\text{reduce} + [33 \ 33 \ 34 \ 35 \ .. \ 64])\]
\[(\text{reduce} + [65 \ 66 \ 67 \ 68 \ .. \ 96])\]
\[(\text{reduce} + [97 \ 98 \ 99 \ 100 \ .. \ 128]))\]
parallel sum using pvreduce

\[(\text{pvreduce} + [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ \ldots \ 128])\]

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

\[(\text{reduce} + [528 \ 1552 \ 2576 \ 3600])\]
parallel sum

$$(\text{pvreduce} + [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ .. \ 128])$$

$$(\text{reduce} + (\text{reduce} + [1 \ 2 \ 3 \ 4 \ 5 \ .. \ 32]))$$

$$(\text{reduce} + [33 \ 33 \ 34 \ 35 \ .. \ 64])$$

$$(\text{reduce} + [65 \ 66 \ 67 \ 68 \ .. \ 96])$$

$$(\text{reduce} + [97 \ 98 \ 99 \ 100 \ .. \ 128]))$$

$$(\text{reduce} + [528 \ 1552 \ 2576 \ 3600])$$

8256
parallel filter

\[
(\text{defn pvfilter} \ [\text{pred v}]
\begin{align*}
&\begin{align*}
&\text{(letfn } ((\text{filt} \ [v \ x] (\text{if} (\text{pred} \ x) (\text{conj} \ v \ x) \ v)))
\end{align*}
\end{align*}
\begin{align*}
&\text{pvreduce filt } [] \ v)
\end{align*}
\end{align*}
\]
\[
(\text{pvfilter even? } [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ .. \ 128])
\]
parallel filter

(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])))
(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

```(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
```
(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

(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))

  (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])))

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]])
(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

$T_f < 0.05$ msec

$0.05 < T_f < 0.1$ msec

$T_f > 0.1$ msec
references

2. Fork/Join API: http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/
questions?
thank you
(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  

\[
\begin{align*}
(\text{defn pvfreq} \ [x] & \quad \text{(letfn [(par-freq} \ [m \ x] \\
& \quad \quad (if (map? \ x) \\
& \quad \quad \quad (merge-with \ #((1 \ %1 \ %2) \ m \ x) \\
& \quad \quad \quad \quad (update-in \ m \ [x] \ #(if \ %1 \ (inc \ %))))))] \\
& \quad (pvreduce \ par-freq \ \{\} \ x) \\
(pvfreq \ [\text{foo} : \text{bar} : \text{baz} : \text{bar} \ .. \ : \text{foo}]) \end{align*}
\]

\[
(\text{reduce} \ par-freq \ (\text{reduce} \ par-freq \ \{\} \ [\text{foo} : \text{foo} \ .. \ : \text{bar}]) \quad \text{(reduce} \ par-freq \ \{\} \ [\text{bar} : \text{foo} \ .. \ : \text{foo}]) \\
(\text{reduce} \ par-freq \ \{\} \ [\text{baz} : \text{bar} \ .. \ : \text{foo}]) \quad \text{(reduce} \ par-freq \ \{\} \ [\text{bar} : \text{baz} \ .. \ : \text{baz}])))
parallel frequency

```
(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}])
```

using pvreduce using pvreduce
parallel frequency

(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}