The best multicore-parallelization refactoring you've never heard of*

Mike Rainey

FHPNC’23

Current draft:

http://mike-rainey.site/papers/pardefunc.pdf

*The title is a riff on “The Best Refactoring You’ve Never Heard Of”, from Koppel’s popular blog post and Compose 2019 talk.

Background

Inspiration

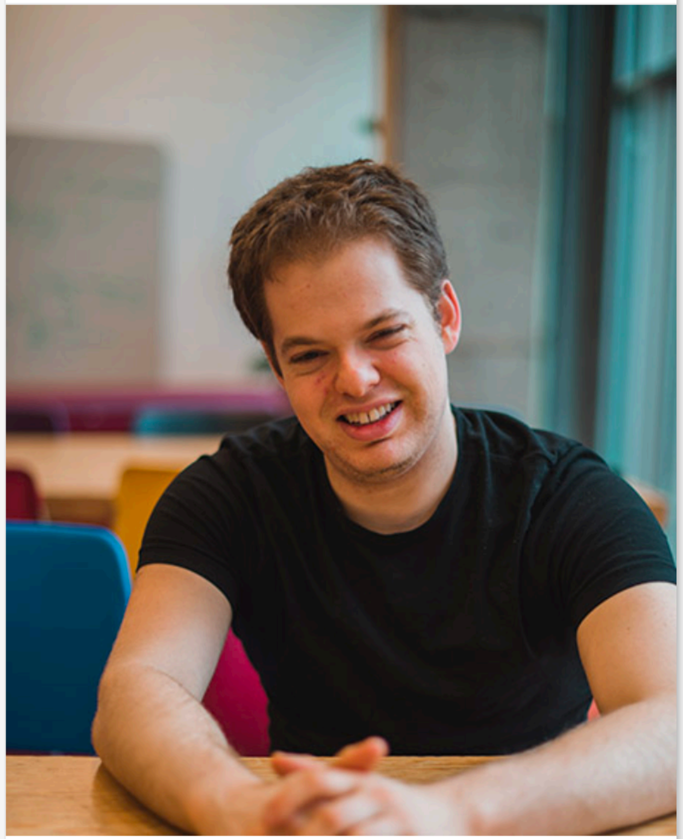
The Best Refactoring You've Never Heard Of

TALKS / JULY 15, 2019

This article/talk helped popularize the use of **CPS** and **defunctionalization** as, e.g., a way to derive efficient iterative algorithms from recursive algorithms.

(video and transcript of my Compose 2019 talk, given June 25th, 2019.)

About me



Jimmy Koppel

Hello world! I'm Jimmy, and I help software engineers **learn to write better code**. Previously, I did my Ph. D. at MIT on ways to make program transformation and synthesis tools easier to build, a.k.a. "meta-metaprogramming." I blog mainly about improving code quality, and occasionally about life quality.

Background

Heartbeat Scheduling as a refactoring via CPS+defunctionalization?

Heartbeat Scheduling: Provable Efficiency for Nested Parallelism

Umut A. Acar
Carnegie Mellon University and Inria
USA
umut@cs.cmu.edu

Arthur Charguéraud
Inria and Univ. of Strasbourg, ICube
France
arthur.chargueraud@inria.fr

Adrien Guatto
Inria
France
adrien@guatto.org

Mike Rainey
Inria and Center for Research in
Extreme Scale Technologies (CREST)
USA
me@mike-rainey.site

Filip Sieczkowski
Inria
France
filip.sieczkowski@inria.fr

Abstract

A classic problem in parallel computing is to take a high-level parallel program written, for example, in nested-parallel style with fork-join constructs and run it efficiently on a real machine. The problem could be considered solved in theory, but not in practice, because the overheads of creating and managing parallel threads can overwhelm their benefits. Developing efficient parallel codes therefore usually requires

guarantees low overheads for all nested parallel programs. We present a prototype C++ implementation and an evaluation that shows that Heartbeat competes well with manually optimized Cilk Plus codes, without requiring manual tuning.

CCS Concepts • **Software and its engineering → Parallel programming languages**;

Keywords parallel programming languages, granularity control

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06...$15.00
https://doi.org/10.1145/3192366.3192391

ACM Reference Format:
Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3192366.3192391

1 Introduction

Java [38], Habanero Java [35], TPL [41], TBB [36], X10 [16], parallel ML [24, 25, 30, 48, 51], and parallel Haskell [43].

These systems have the desirable feature that the user expresses parallelism at an abstract level, without directly specifying how to map lightweight threads (just threads, from hereon) onto processors. A *scheduler* is then responsible for the placement of threads. The scheduler does not require that the thread structure is known ahead of time, and therefore operates online as part of the runtime system. Many scheduling algorithms have been developed, taking into account a variety of asymptotic cost factors including execution time, space consumption, and locality [1–3, 5, 9–13, 15, 18, 29, 31, 45].

Most scheduling algorithms that come with a formal analysis establish asymptotic bounds in a simplified model in which spawning a thread has unit cost. Correspondingly, the job of achieving low constant factors for scheduling operations is usually treated as a purely empirical question, and approached as such. Yet, in practice, depending on the implementation, the cost of creating a thread, scheduling it,

Using the recursion-to-iteration refactoring technique, and Heartbeat Scheduling, we can solve tricky multicore-parallelization problems via a series of simple refactoring steps.

Our multicore-parallelization challenge:

Traverse a pointer-based, binary tree

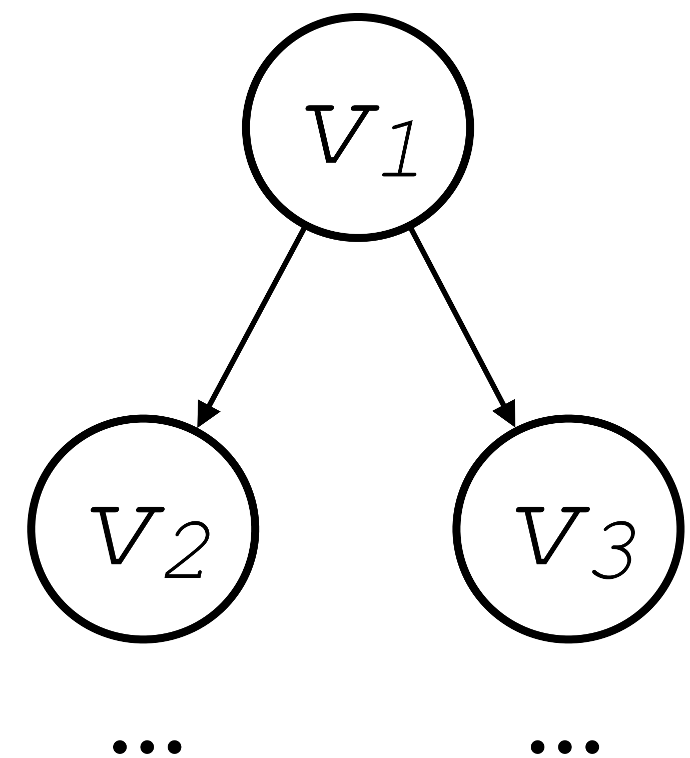
```
type node = {v : int,    bs : node*[2]}
```

Integer payload

Child pointers

```
sum(node* n) → int {
  if (n == null) return 0
  return sum(n.bs[0])
       + sum(n.bs[1])
       + n.v }
```

Our reference program
(In pseudo C++)



Introducing our parallelism primitive

Binary fork join

```
sum(node* n) → int {
  if (n == null) return 0
  return sum(n.bs[0])
       + sum(n.bs[1])
       + n.v }
```

Our reference program

```
sum(node* n) → int {
  if (n == null) return 0
  s = new int[2]
  { s[i] = sum(n.bs[i])
  } i ∈ {0, 1}
  return s[0]
       + s[1]
       + n.v }
```

We introduce a temporary array to hold intermediate results.

Syntactic sugar for reducing clutter

Enables statements s_1 and s_2 to execute in parallel.

```
fork2join { s1 } { s2 }
s3
...
```

All of their writes are visible at the **join point** (i.e., s_3).

```
sum(node* n) → int {
  if (n == null) return 0
  s = new int[2]
  fork2join {
    s[i] = sum(n.bs[i])
  } i ∈ {0, 1}
  return s[0]
       + s[1]
       + n.v }
```

Recursive calls can execute in parallel.

Introducing our parallelism primitive

Binary fork join

```
sum(node* n) → int {
  if (n == null) return 0
  return sum(n.bs[0])
       + sum(n.bs[1])
       + n.v }
```

Our reference program

```
sum(node* n) → int {
  if (n == null) return 0
  s = new int[2]
  { s[i] = sum(n.bs[i])
  } i ∈ {0, 1}
  return s[0]
       + s[1]
       + n.v }
```

We introduce a temporary array to hold intermediate results.

Syntactic sugar for reducing clutter

Enables statements s_1 and s_2 to execute in parallel.

```
fork2join { s1 } { s2 }
s3
...
```

All of their writes are visible at the **join point** (i.e., s_3).

```
sum(node* n) → int {
  if (n == null) return 0
  s = new int[2]
  fork2join {
    s[i] = sum(n.bs[i])
  } i ∈ {0, 1}
  return s[0]
       + s[1]
       + n.v }
```

Recursive calls can execute in parallel.

Introducing our parallelism primitive

Binary fork join

```
sum(node* n) → int {
  if (n == null) return 0
  return sum(n.bs[0])
       + sum(n.bs[1])
       + n.v }
```

Our reference program

```
sum(node* n) → int {
  if (n == null) return 0
  s = new int[2]
  { s[i] = sum(n.bs[i])
  } i ∈ {0, 1}
  return s[0]
       + s[1]
       + n.v }
```

We introduce a temporary array to hold intermediate results.

Syntactic sugar for reducing clutter

```
sum(node* n) → int {
  if (n == null) return 0
  s = new int[2]
  fork2join {
    s[i] = sum(n.bs[i])
  } i ∈ {0, 1}
  return s[0]
       + s[1]
       + n.v }
```

Recursive calls can execute in parallel.

Enables statements s_1 and s_2 to execute in parallel.

```
fork2join { s1 } { s2 }
s3
...
```

All of their writes are visible at the **join point** (i.e., s_3).

Introducing our parallelism primitive

Binary fork join

```
sum(node* n) → int {
  if (n == null) return 0
  return sum(n.bs[0])
       + sum(n.bs[1])
       + n.v }
```

Our reference program

```
sum(node* n) → int {
  if (n == null) return 0
  s = new int[2]
  { s[i] = sum(n.bs[i])
  } i ∈ {0, 1}
  return s[0]
       + s[1]
       + n.v }
```

We introduce a temporary array to hold intermediate results.

Syntactic sugar for reducing clutter

```
sum(node* n) → int {
  if (n == null) return 0
  s = new int[2]
  fork2join {
    s[i] = sum(n.bs[i])
  } i ∈ {0, 1}
  return s[0]
       + s[1]
       + n.v }
```

Recursive calls can execute in parallel.

Enables statements s_1 and s_2 to execute in parallel.

fork2join { s_1 } { s_2 }

s_3

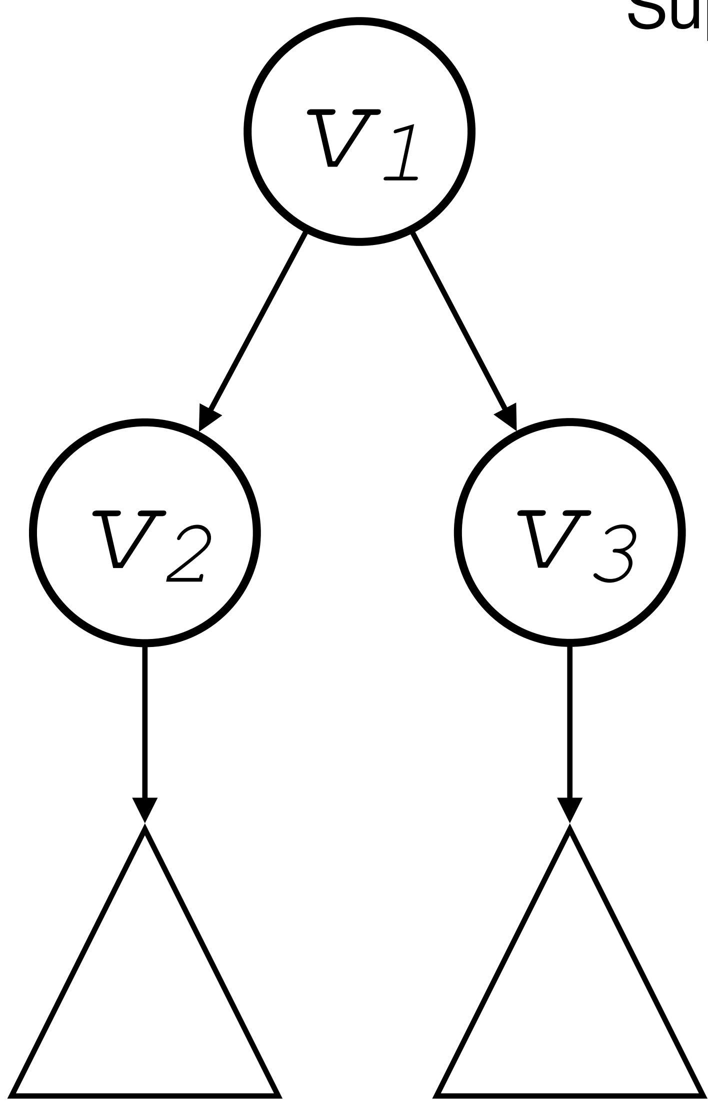
...

All of their writes are visible at the **join point** (i.e., s_3).

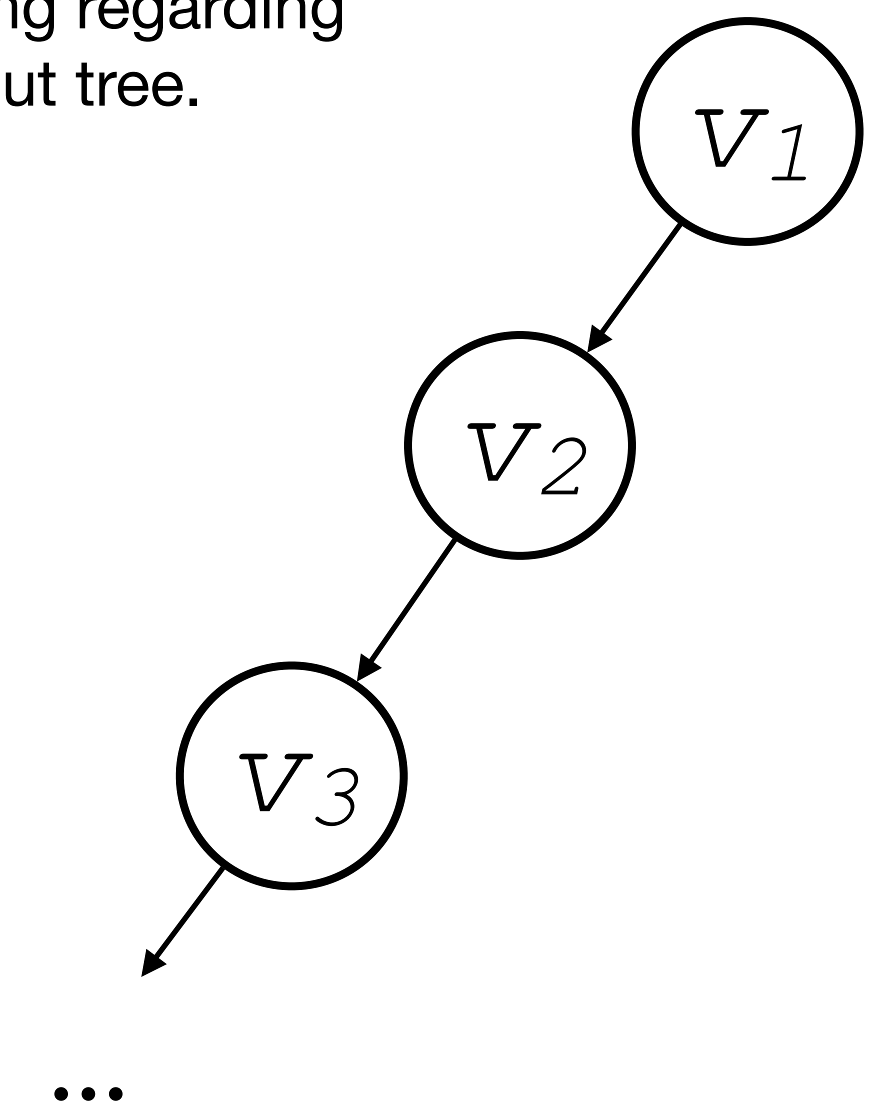
Challenge #1:

Granularity control

Suppose we can assume nothing regarding the shape of any given input tree.



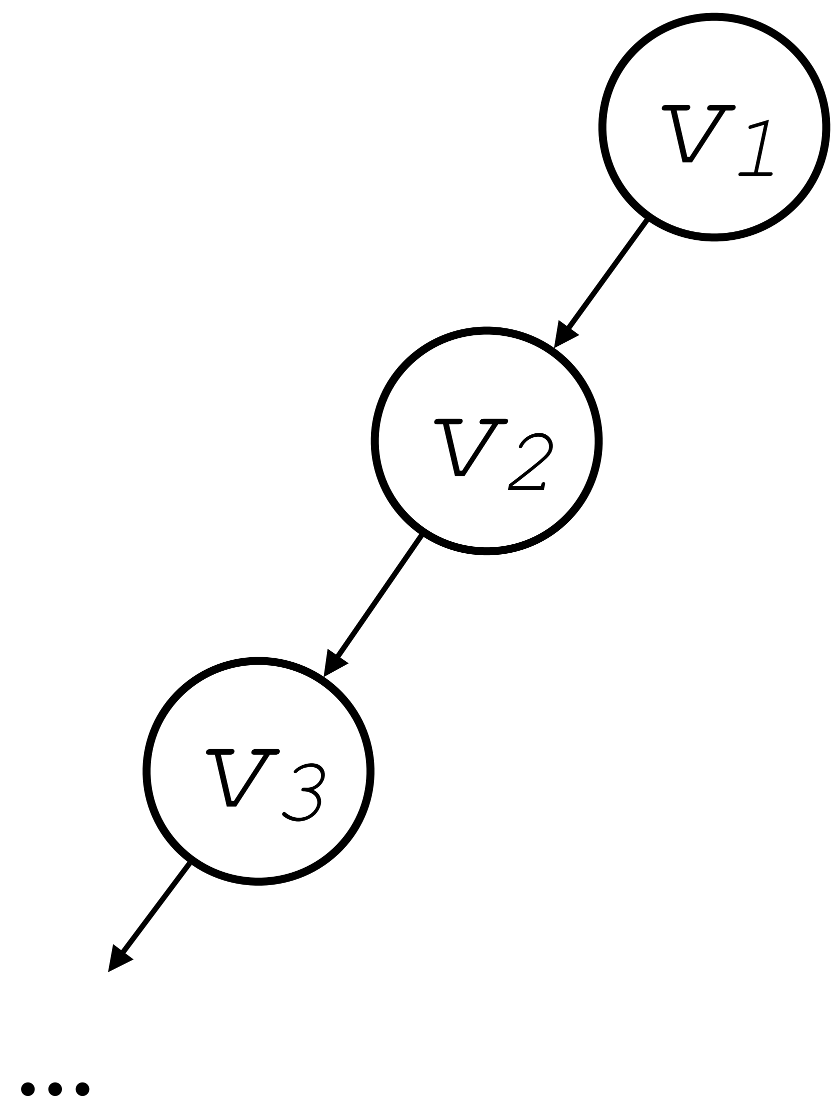
Balanced and large with abundant parallelism ⇒ Want to parallelize aggressively



Long chains with no parallelism ⇒ Want to serialize

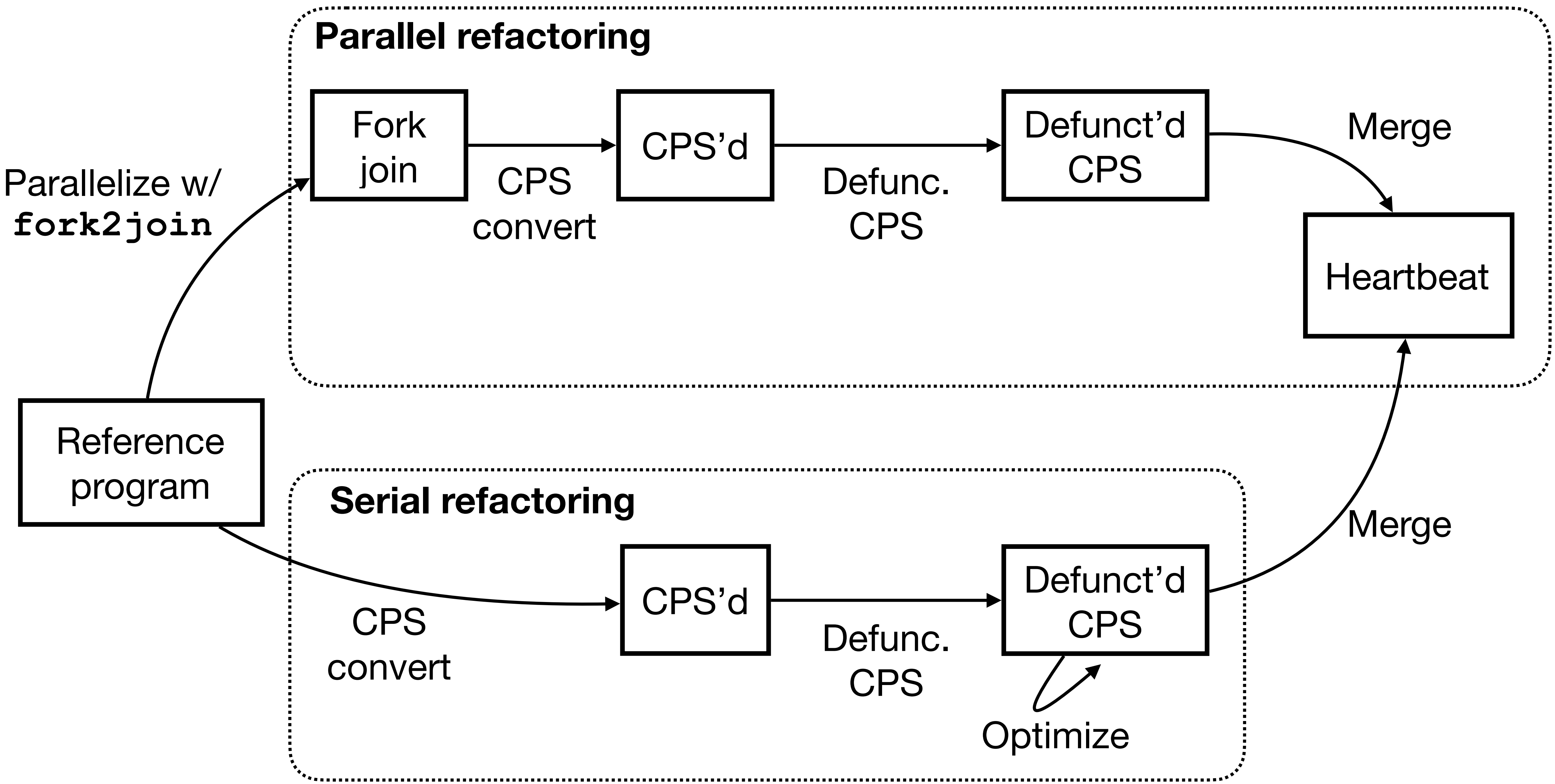
Challenge #2:

Overflow of the call stack is possible given certain inputs



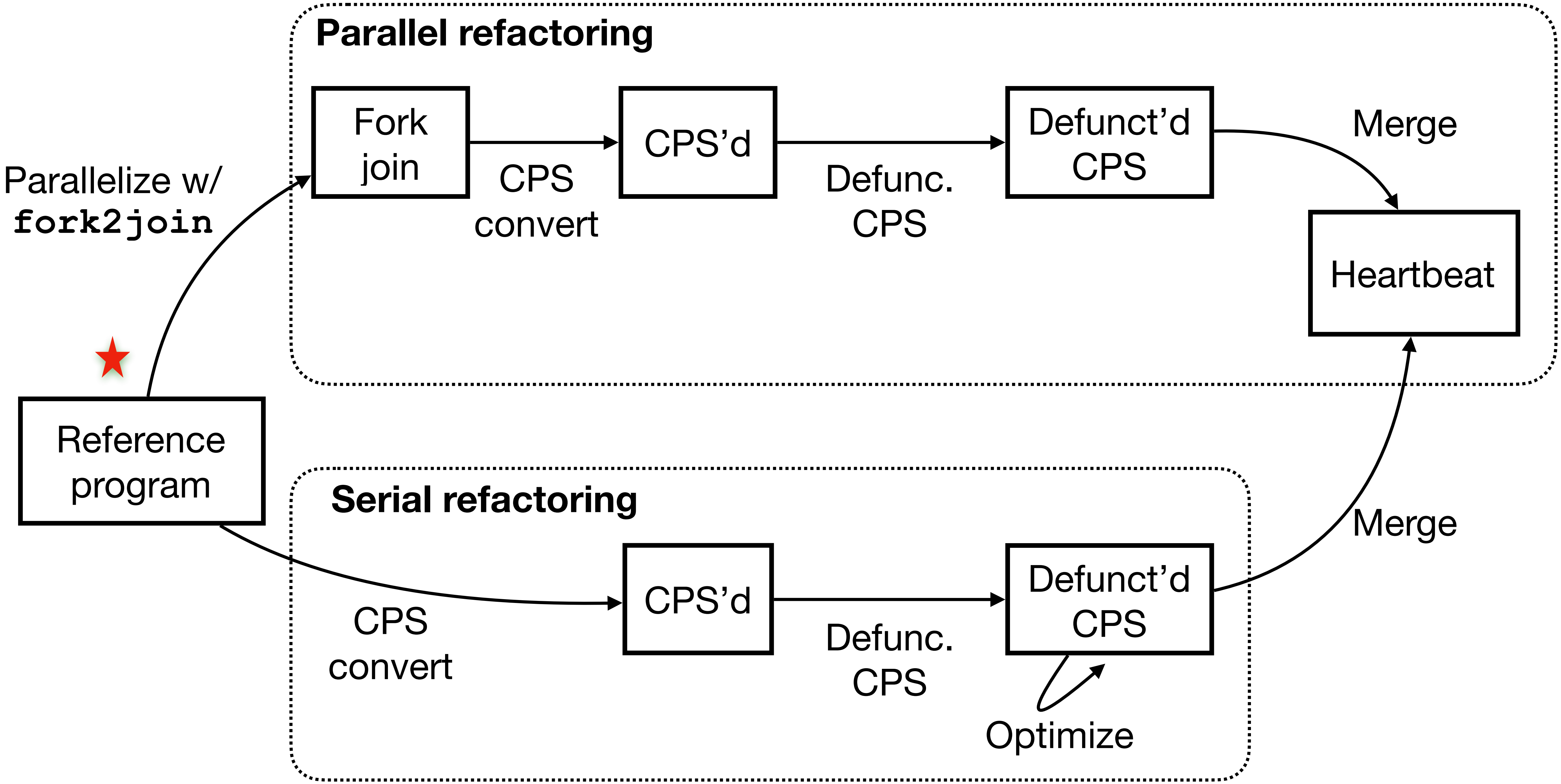
Our multicore-parallelization refactoring

Roadmap



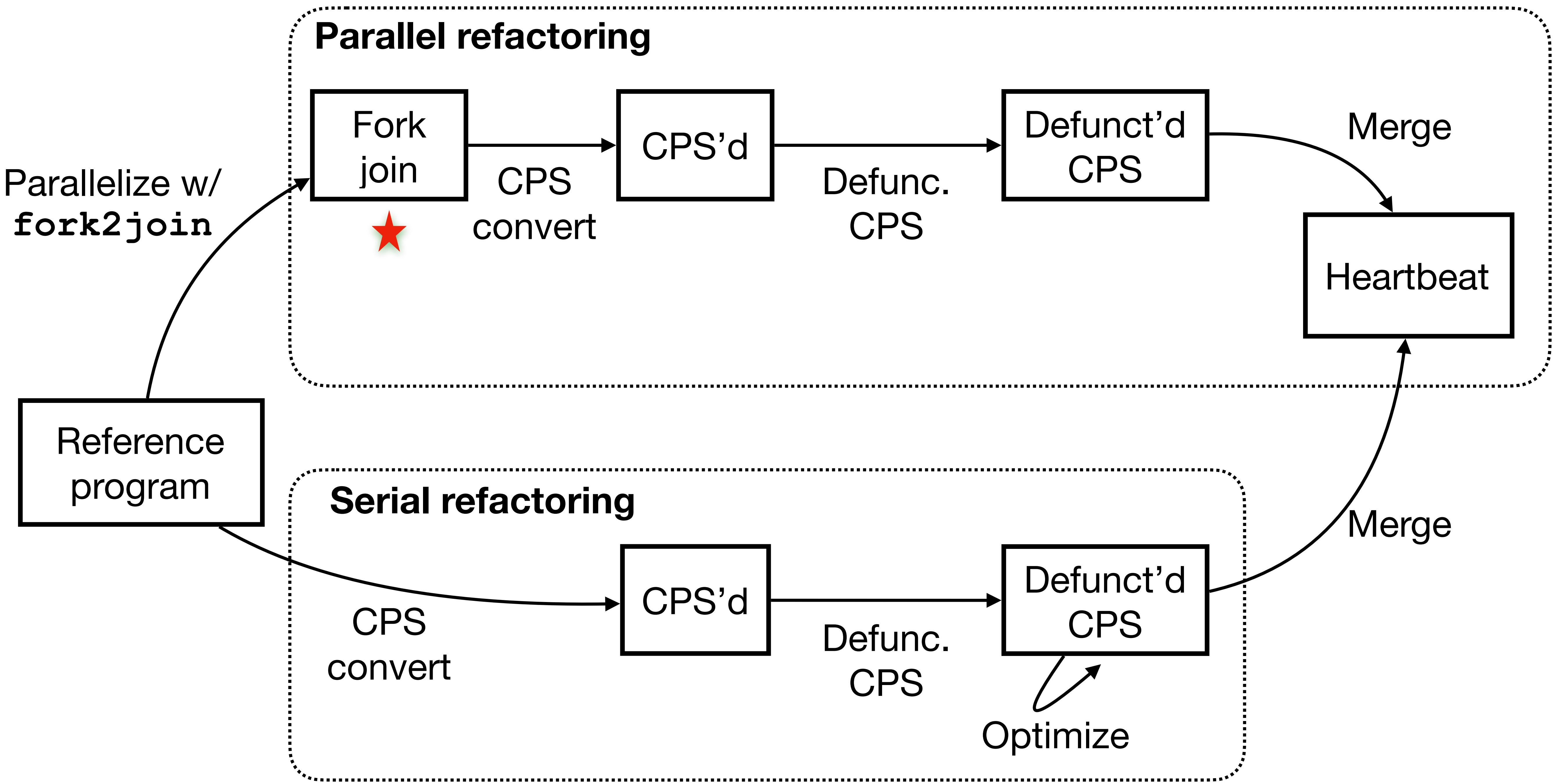
Our multicore-parallelization refactoring

Roadmap



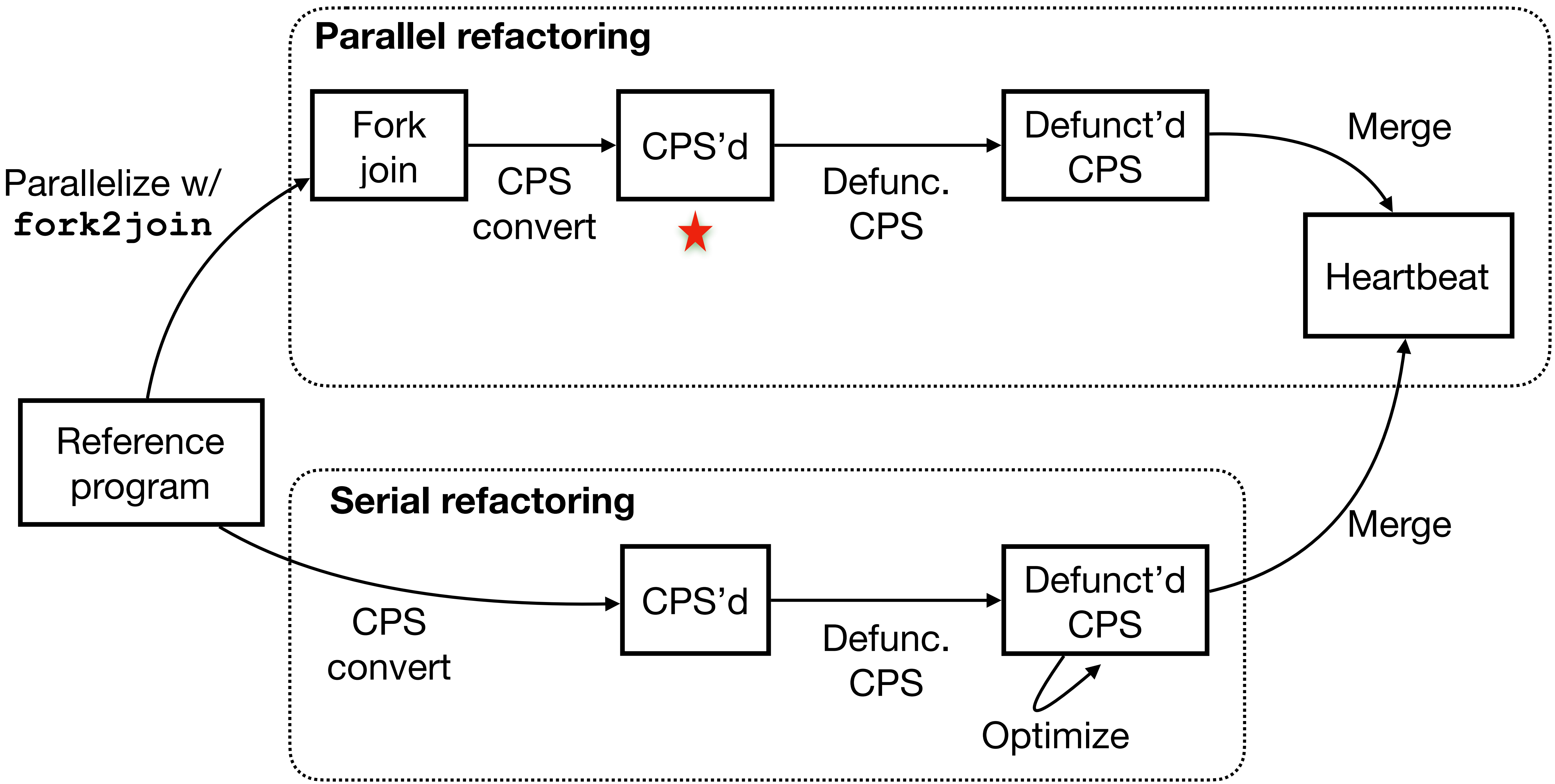
Our multicore-parallelization refactoring

Roadmap



Our multicore-parallelization refactoring

Roadmap

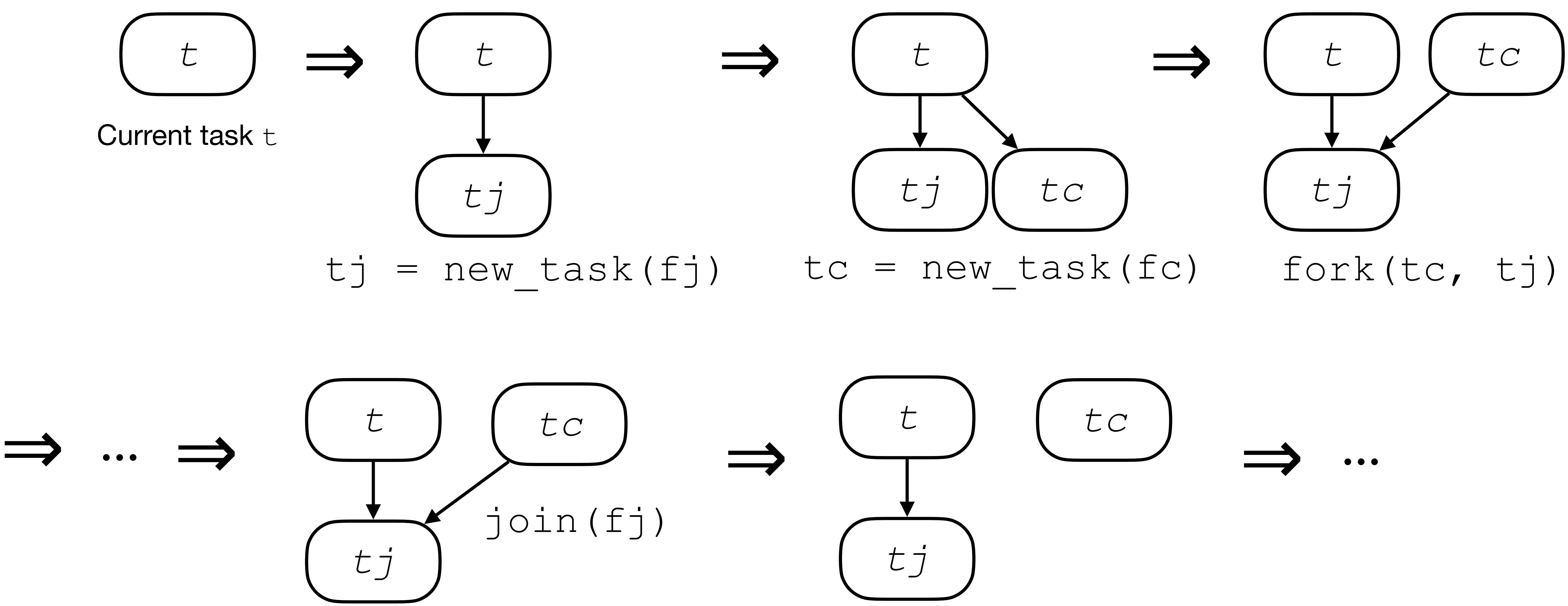


Fork-join primitives

Library interface

`t = new_task(f)`	Creates a new task *t* that, when run will invoke its thunk *f*.
`fork(tc, tj)`	Registers one dependency edge from the current task to join task *tj* and one from child task *tc* to *tj*, and schedules *tc*.
`join(tj)`	Resolves one dependency edge on join task *tj*.

Example: task `t` spawns a child task `tc` with join point `tj`

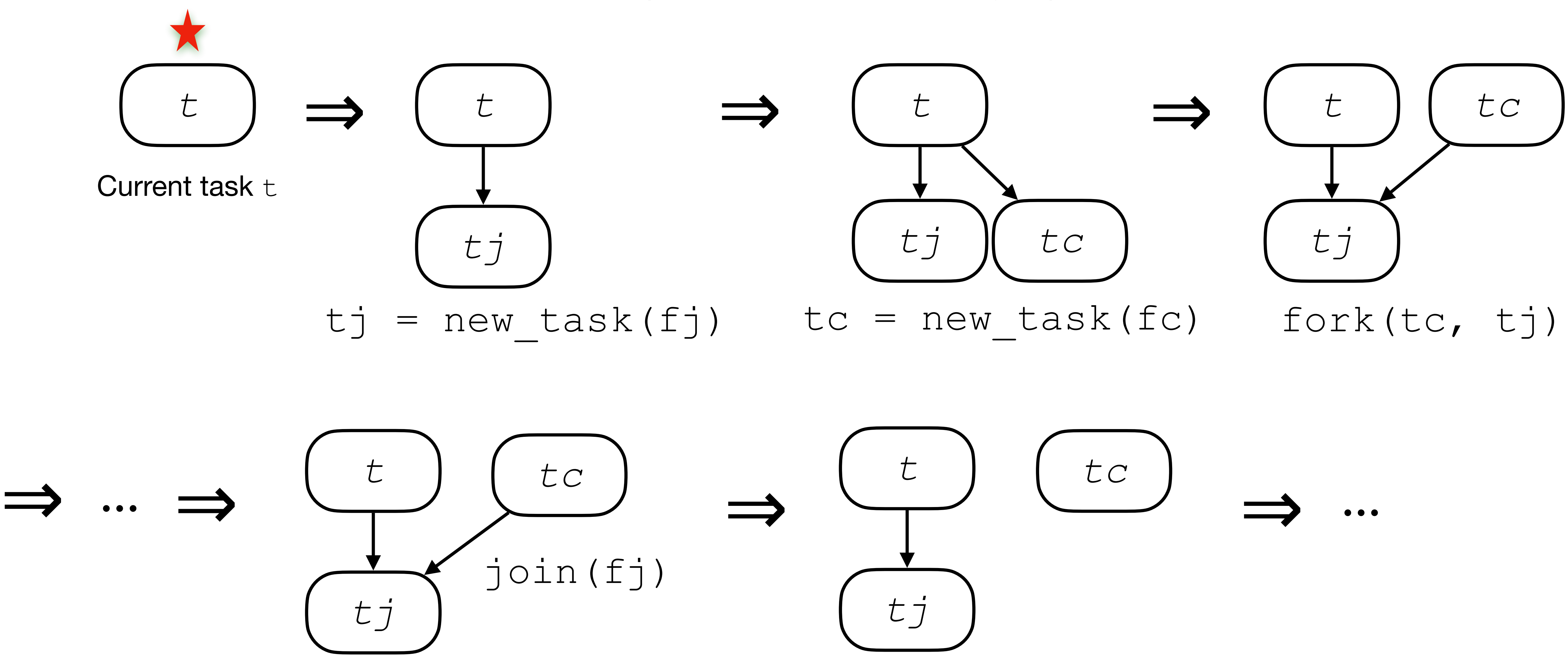


Fork-join primitives

Library interface

`t = new_task(f)`	Creates a new task *t* that, when run will invoke its thunk *f*.
`fork(tc, tj)`	Registers one dependency edge from the current task to join task *tj* and one from child task *tc* to *tj*, and schedules *tc*.
`join(tj)`	Resolves one dependency edge on join task *tj*.

Example: task `t` spawns a child task `tc` with join point `tj`

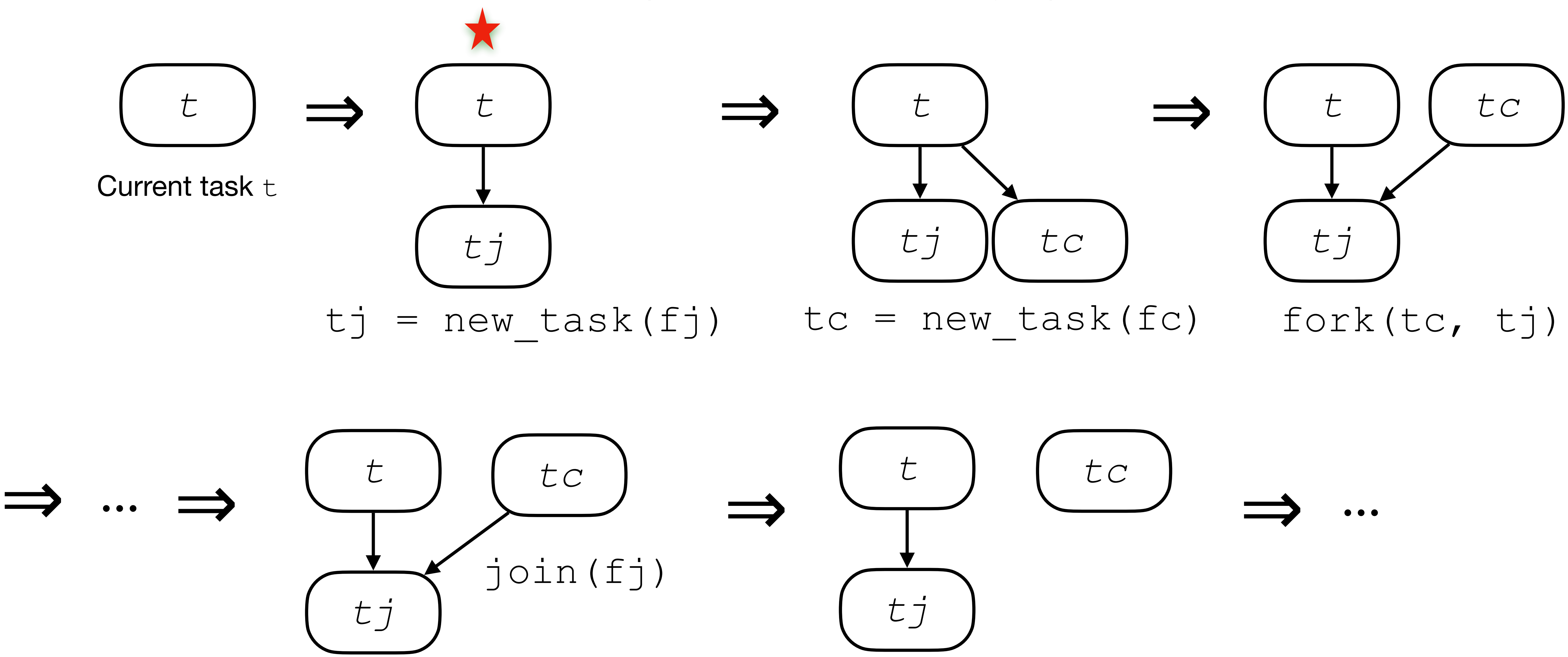


Fork-join primitives

Library interface

`t = new_task(f)`	Creates a new task *t* that, when run will invoke its thunk *f*.
`fork(tc, tj)`	Registers one dependency edge from the current task to join task *tj* and one from child task *tc* to *tj*, and schedules *tc*.
`join(tj)`	Resolves one dependency edge on join task *tj*.

Example: task `t` spawns a child task `tc` with join point `tj`

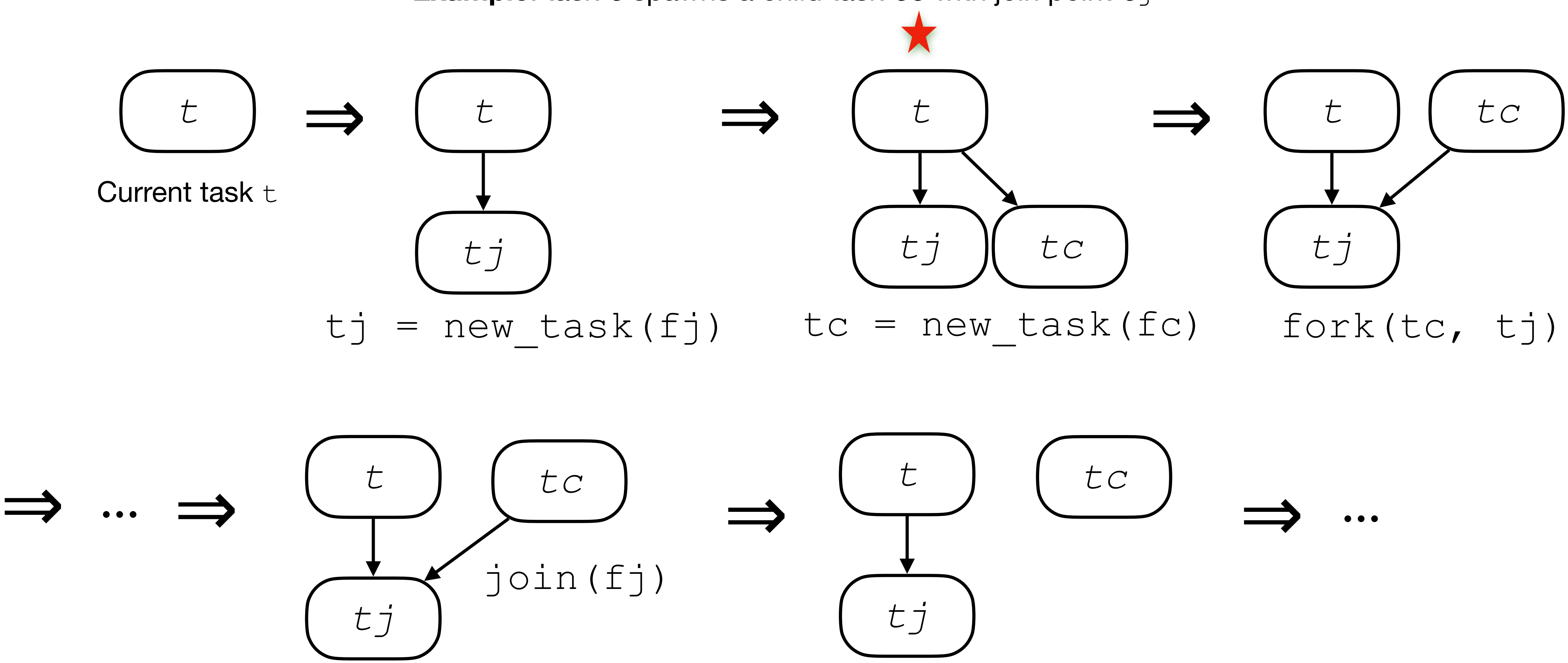


Fork-join primitives

Library interface

`t = new_task(f)`	Creates a new task `t` that, when run will invoke its thunk `f`.
`fork(tc, tj)`	Registers one dependency edge from the current task to join task `tj` and one from child task `tc` to `tj`, and schedules `tc`.
`join(tj)`	Resolves one dependency edge on join task `tj`.

Example: task `t` spawns a child task `tc` with join point `tj`

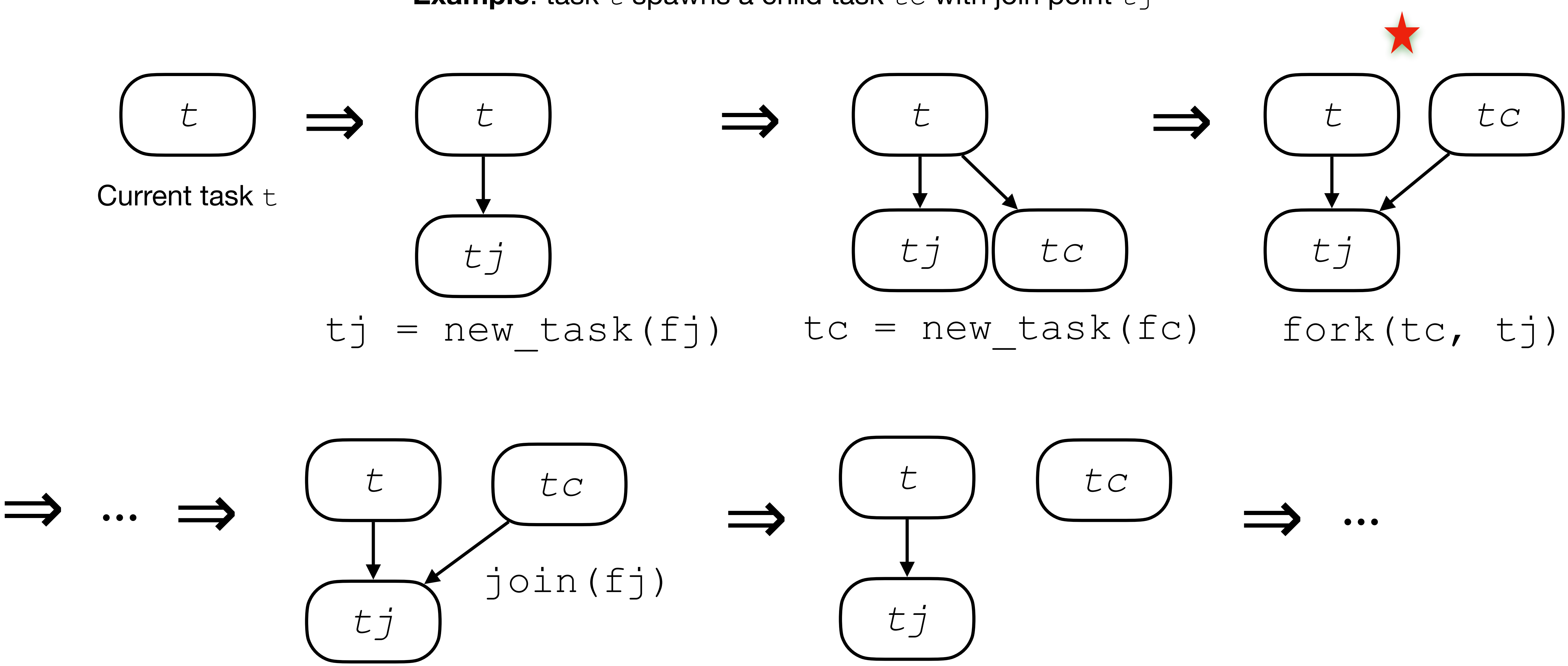


Fork-join primitives

Library interface

`t = new_task(f)`	Creates a new task `t` that, when run will invoke its thunk `f`.
`fork(tc, tj)`	Registers one dependency edge from the current task to join task `tj` and one from child task `tc` to `tj`, and schedules `tc`.
`join(tj)`	Resolves one dependency edge on join task `tj`.

Example: task `t` spawns a child task `tc` with join point `tj`

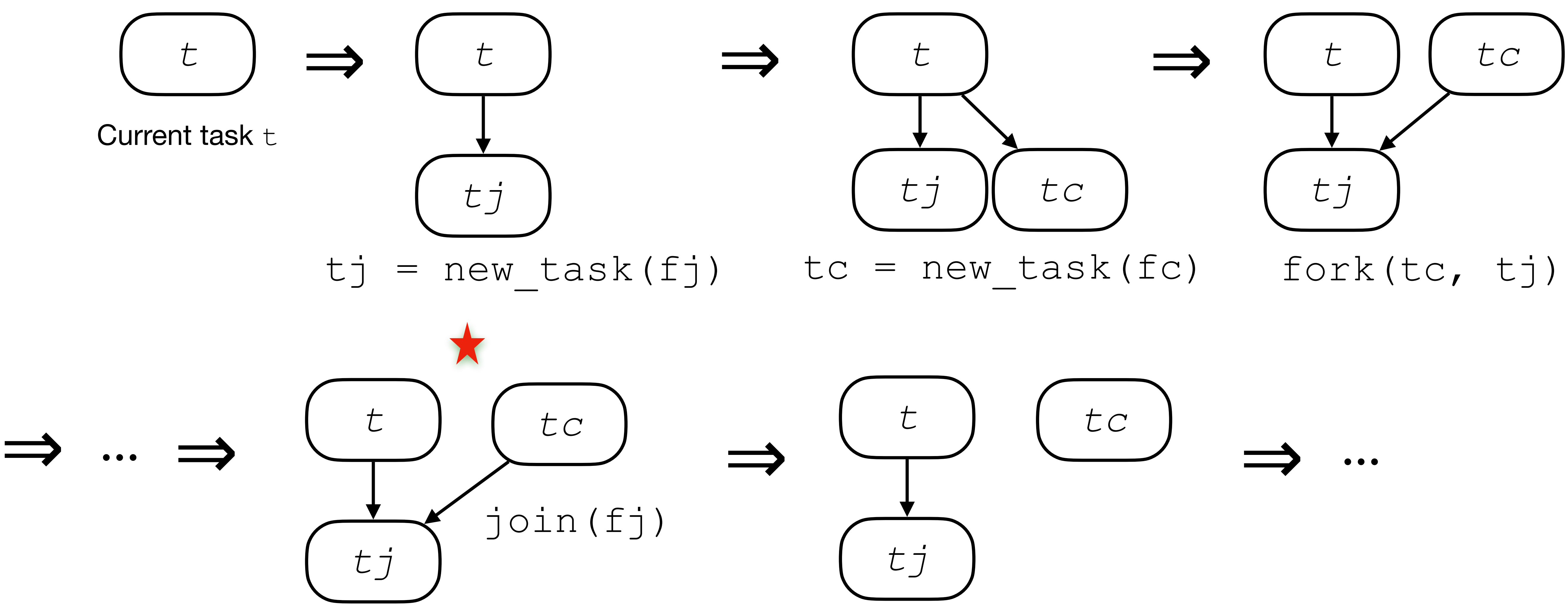


Fork-join primitives

Library interface

`t = new_task(f)`	Creates a new task `t` that, when run will invoke its thunk `f`.
`fork(tc, tj)`	Registers one dependency edge from the current task to join task `tj` and one from child task `tc` to `tj`, and schedules `tc`.
`join(tj)`	Resolves one dependency edge on join task `tj`.

Example: task `t` spawns a child task `tc` with join point `tj`

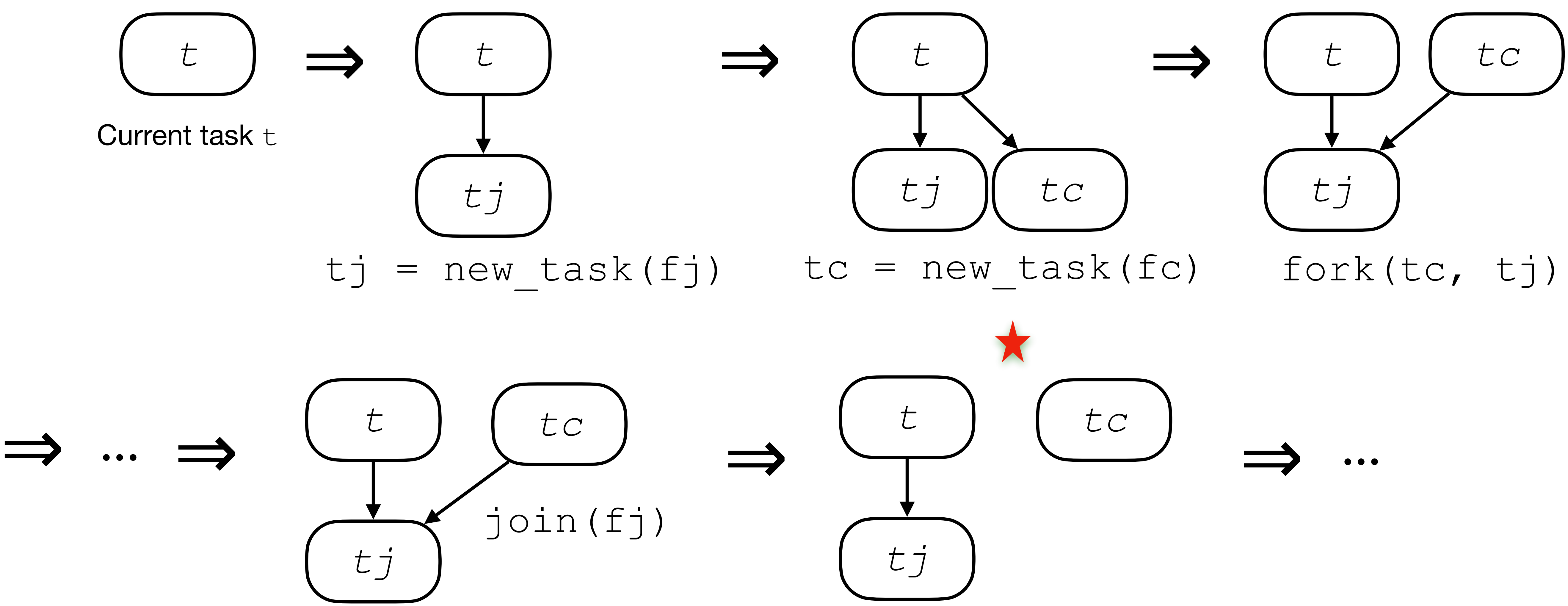


Fork-join primitives

Library interface

`t = new_task(f)`	Creates a new task t that, when run will invoke its thunk f.
`fork(tc, tj)`	Registers one dependency edge from the current task to join task tj and one from child task tc to tj, and schedules tc.
`join(tj)`	Resolves one dependency edge on join task tj.

Example: task `t` spawns a child task `tc` with join point `tj`



Refactoring for parallel traversal

CPS convert the parallel algorithm

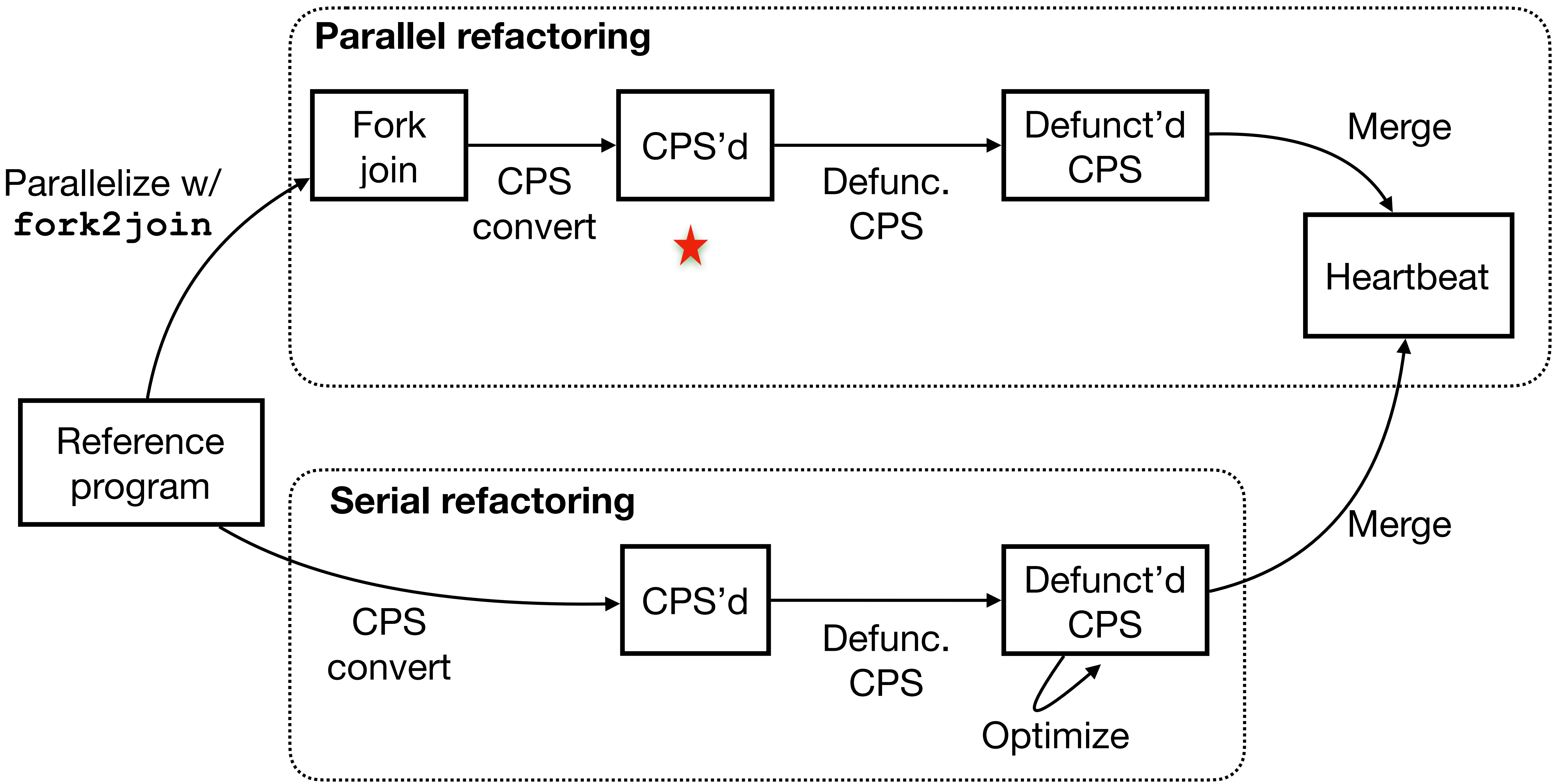
```
sum(node* n) → int {

  if (n == null) return 0
  s = new int[2]
  fork2join {
    s[i] = sum(n.bs[i])
  } i ∈ {0, 1}
  return s[0]
       + s[1]
       + n.v }
```

```
sum(node* n, k : int → void)
                              → void {
  if (n == null) { k(0); return }
  s = new int[2]
  tj = new_task(λ () ⇒
           k(s[0] + s[1] + n.v))
  { ti = new_task(λ () ⇒
         sum(n.bs[i], λ si ⇒ {
           s[i] = si; join(tj)})
    fork(ti, tj) } i ∈ {0, 1} }
```

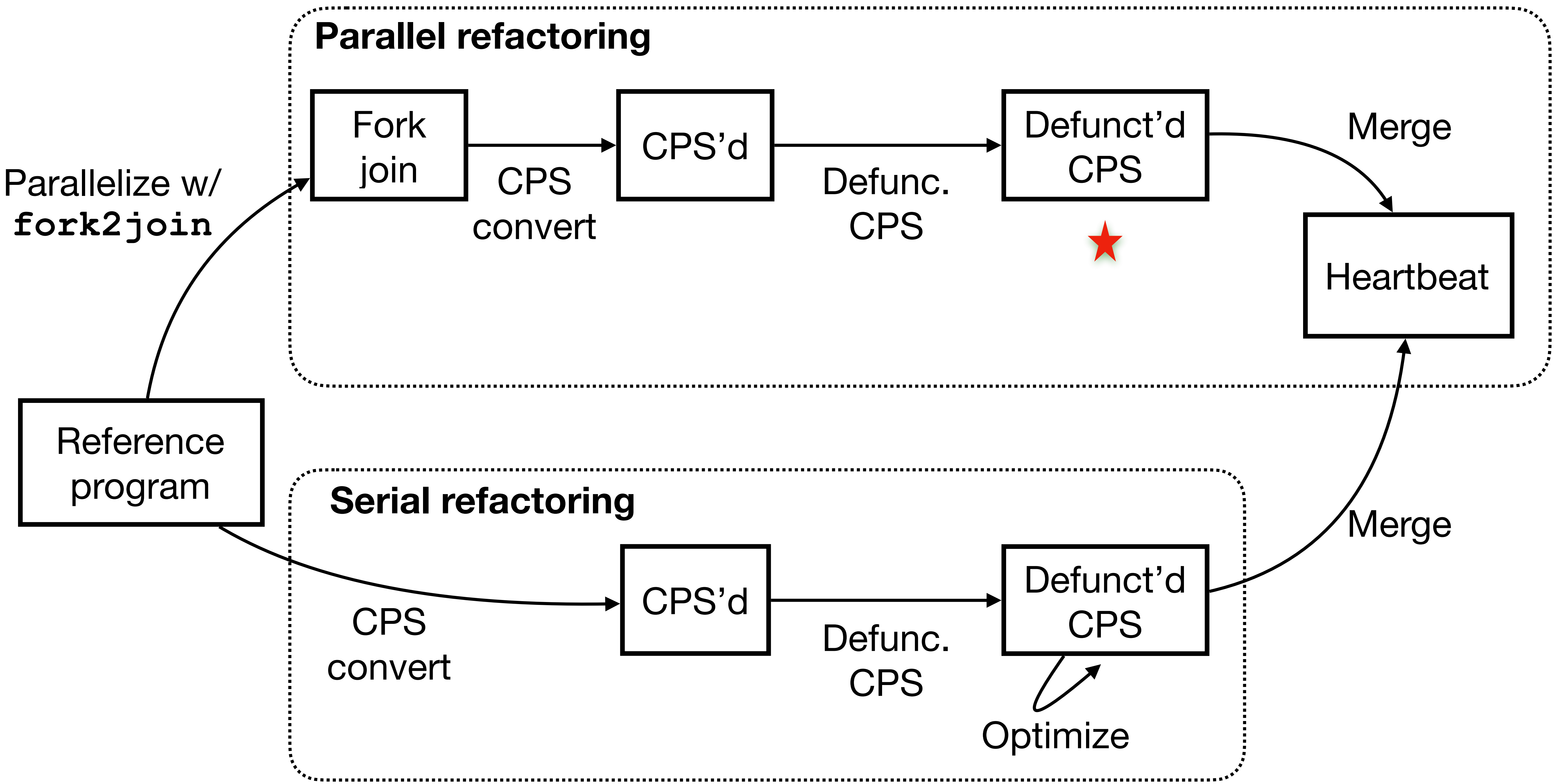
Our multicore-parallelization refactoring

Roadmap



Our multicore-parallelization refactoring

Roadmap



Refactoring for parallel traversal

Defunctionalization of CPS

```
sum(node* n, k : int → void)
                          → void {
  if (n == null) { k(0); return }
  s = new int[2]
  tj = new_task(λ () ⇒
           k(s[0] + s[1] + n.v))
  { tᵢ = new_task(λ () ⇒
        sum(n.bs[i], λ sᵢ ⇒ { // KPBranch
           s[i] = sᵢ; join(tj)})
    fork(tᵢ, tj) } i ∈ {0, 1} }
```

```
sum(n0, λ s ⇒ { // KTerm
  *ans = s })
```

Top-level call with input `n0`; final result pointed to by `ans`

There are two possible continuations (highlighted).
We introduce a data constructor to represent each:

```
type kont =
  | KTerm of int* // dest. of final result
  | KPBranch of {i : int, s : int*, tj : task*}
```

Refactoring for parallel traversal

Defunctionalization of CPS

```
sum(node* n, k : kont*) → void {
  if (n == null) {
    apply(k, 0); return }
  s = new int[2]
  tj = new_task(λ () ⇒
        apply(k, s[0] + s[1] + n.v))
  { ti = new_task(λ () ⇒
         sum(n.bs[i],
             KPBranch{i=i, s=s, tj=tj}))
    fork(ti, tj) } i ∈ {0, 1} }
```

```
sum(n0, KTerm ans)
```

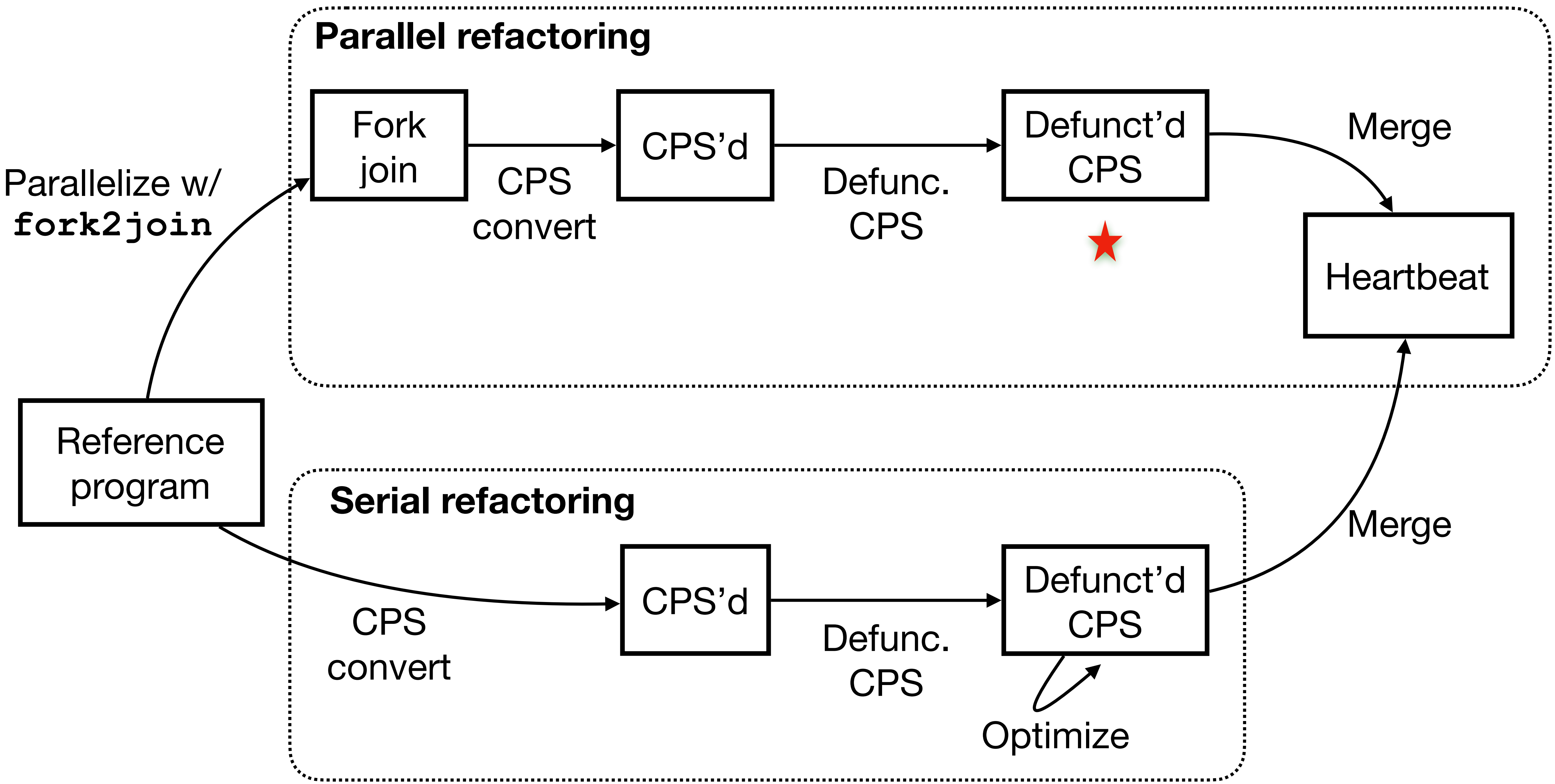
Top-level call;
final result pointed to by `ans`

```
apply(kont* k, sa : int) → void {
  match *k with
  | KPBranch{i, s, tj} ⇒
    {s[i] = sa; join(tj)}
  | KTerm ans ⇒ {*ans = sa} }
```

```
type kont =
  | KTerm of int* // final result
  | KPBranch of {i : int, s : int*, tj : task*}
```

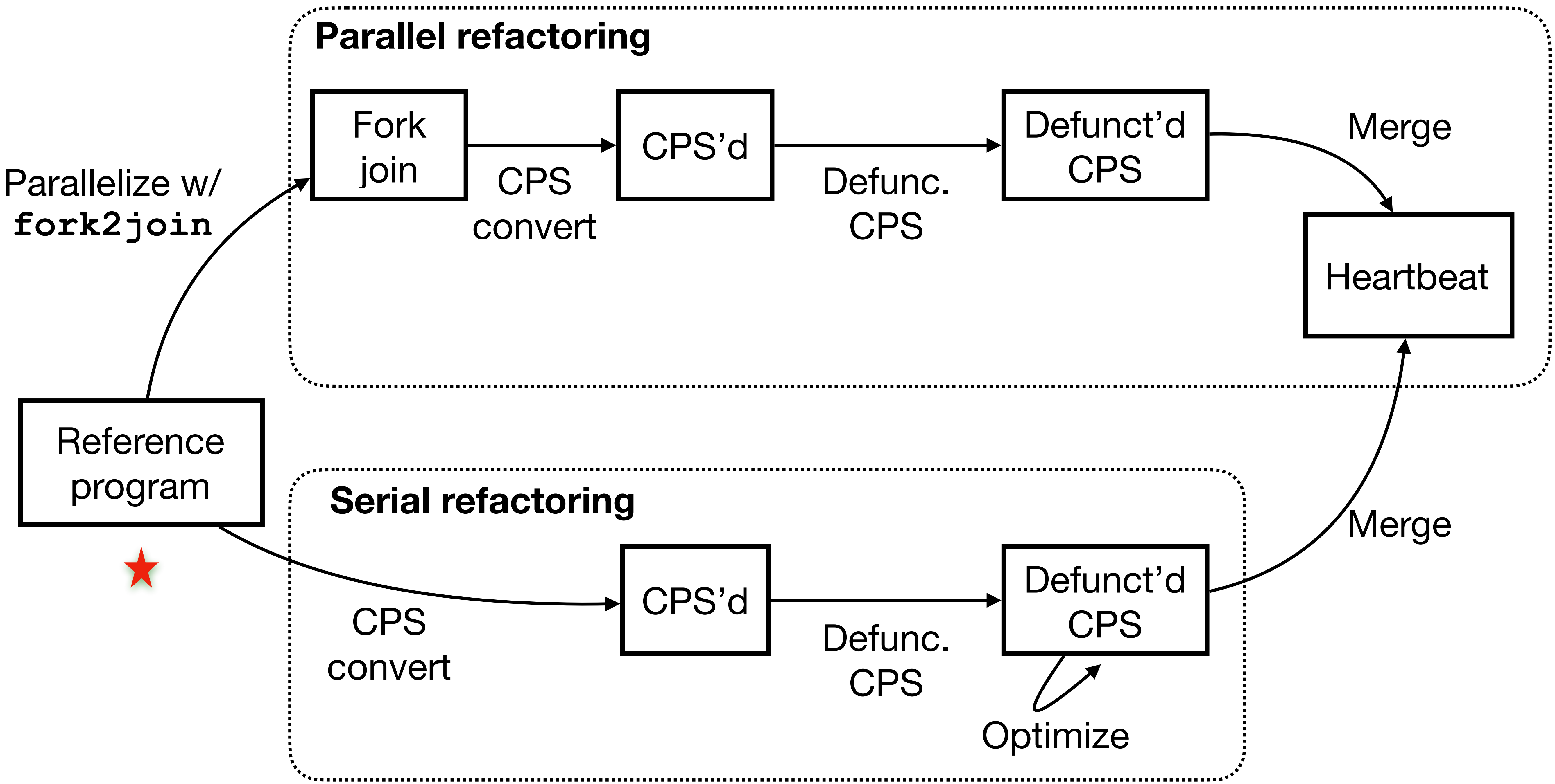
Our multicore-parallelization refactoring

Roadmap



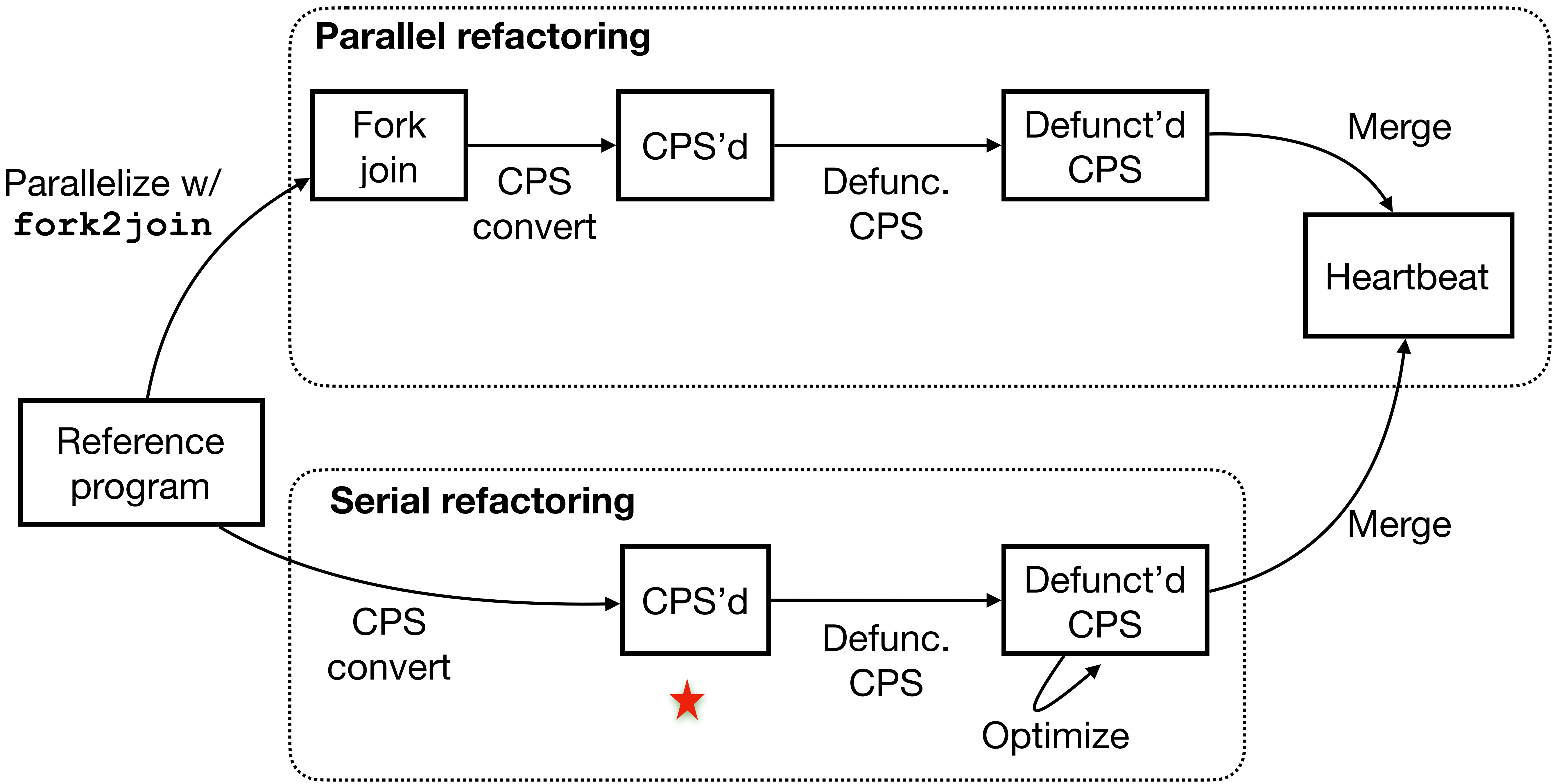
Our multicore-parallelization refactoring

Roadmap



Our multicore-parallelization refactoring

Roadmap



Refactoring for serial traversal

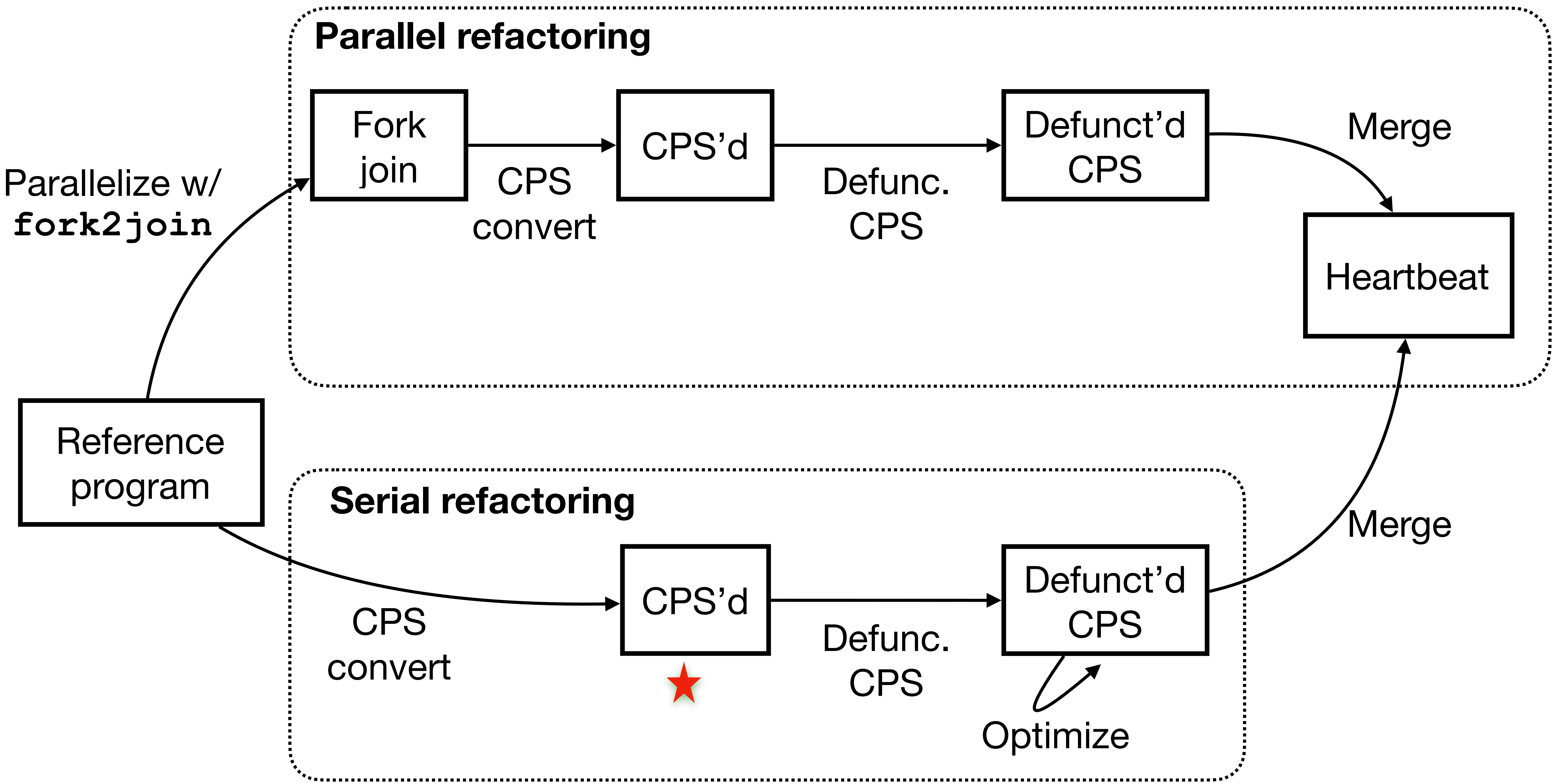
CPS convert the serial algorithm

```
sum(node* n) → int {
  if (n == null) return 0
  return sum(n.bs[0])
       + sum(n.bs[1])
       + n.v }
```

```
sum(node* n, k : int → void) → void {
  if (n == null) { k(0); return }
  sum(n.bs[0], λ s0 ⇒
    sum(n.bs[1], λ s1 ⇒
      k(s0 + s1 + n.v))) }
```

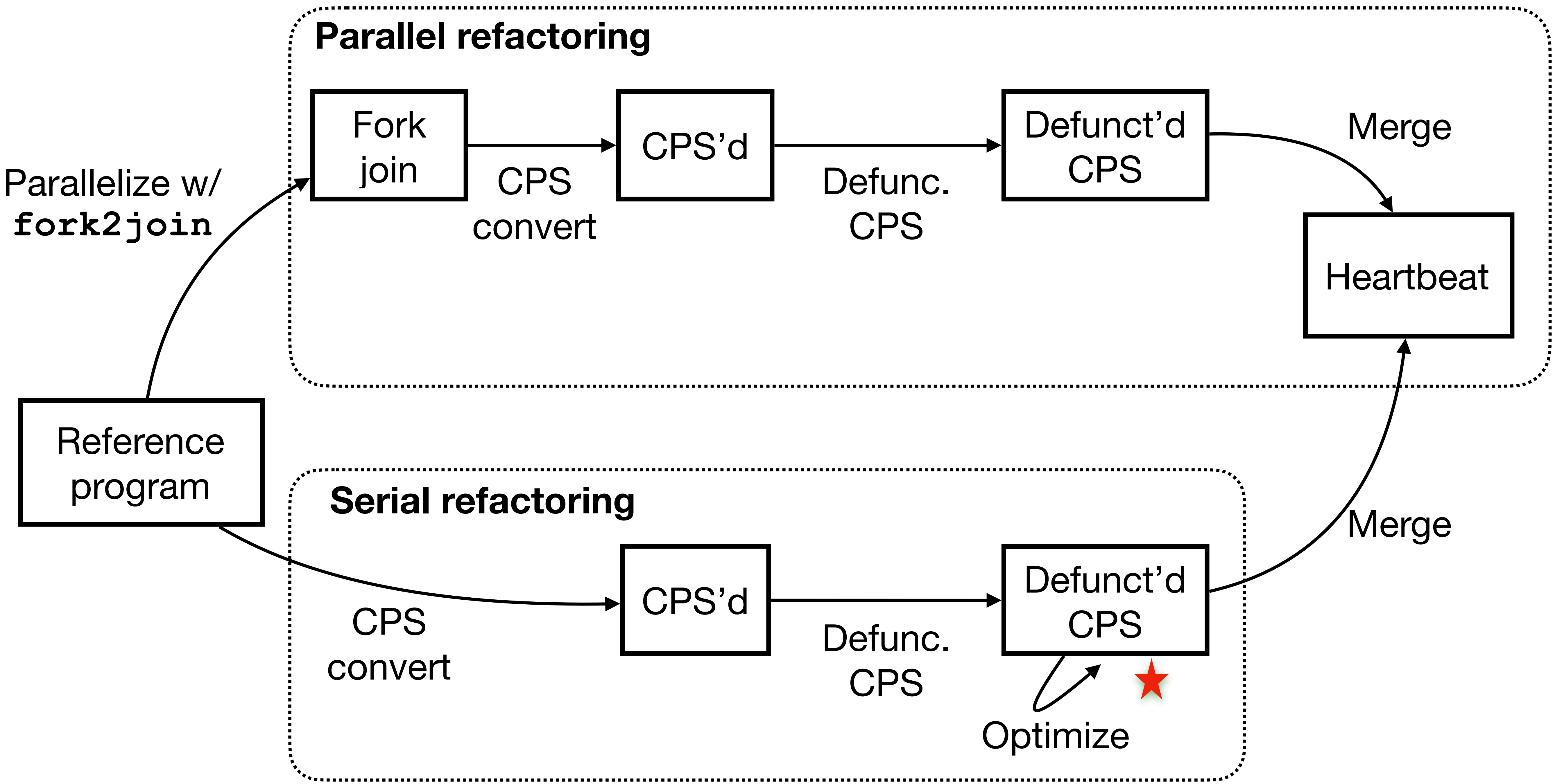
Our multicore-parallelization refactoring

Roadmap



Our multicore-parallelization refactoring

Roadmap



Refactoring for serial traversal

Defunctionalization of CPS

```
sum(node* n, k : int → void) → void {
  if (n == null) { k(0); return }
  sum(n.bs[0], λ s₀ ⇒ // KSBranch0
    sum(n.bs[1], λ s₁ ⇒ // KSBranch1
      k(s₀ + s₁ + n.v))) }
```

```
type kont = ...
  | KSBranch0 of {n : node*, k : kont*}
  | KSBranch1 of {s₀ : int, n : node*, k : kont*}
```

Refactoring for serial traversal

Defunctionalization of CPS

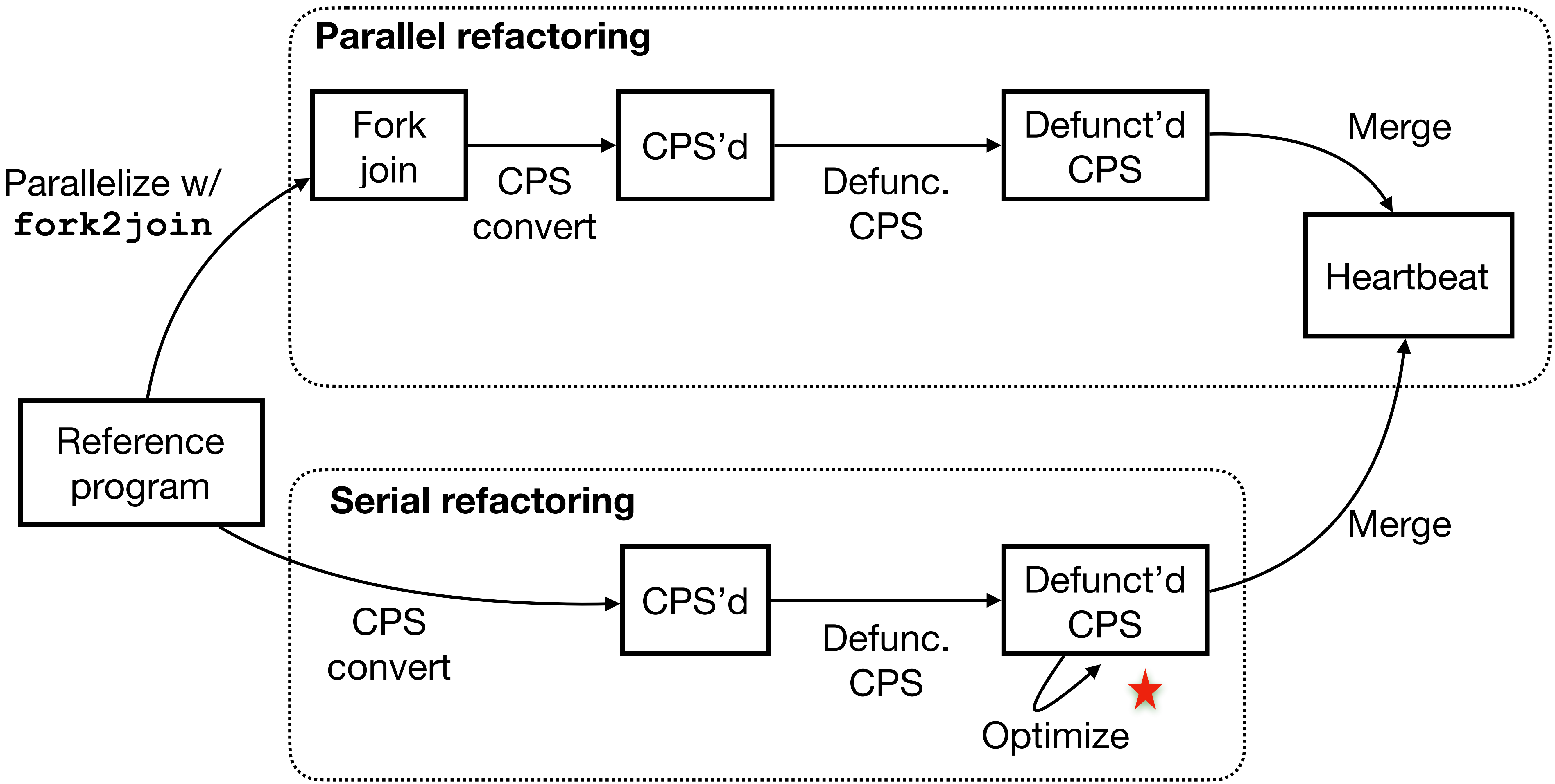
```
sum(node* n, k : kont*) → void {
  if (n == null) { apply(k, 0); return }
  sum(n.bs[0], KSBranch0{n=n, k=k})

apply(kont* k, sa : int) → void {
  match *k with
  | KSBranch0{n, k=k1} ⇒ {
    sum(n.bs[1], KSBranch1{s0=sa, n=n, k=k1}) }
  | KSBranch1{s0, n, k=k1} ⇒ {
    apply(k1, s0 + sa + n.v) }
  | KTerm ans ⇒ {*ans = sa} }
```

```
type kont = ...
  | KSBranch0 of {n : node*, k : kont*}
  | KSBranch1 of {s0 : int, n : node*, k : kont*}
```

Our multicore-parallelization refactoring

Roadmap



Refactoring for serial traversal

Tail-call elimination of `apply`

```
apply(kont* k, sa : int) → void {
  while (true)
    match *k with
    | KSBranch0{n, k=k1} ⇒ {
      sum(n.bs[1], KSBranch1{s0=sa, n=n, k=k1})
      return }
    | KSBranch1{s0, n, k=k1} ⇒ {
      sa =s0 + sa + n.v; k=k1 }
    | KTerm ans ⇒ { *ans = sa; return } }
```

Refactoring for serial traversal

Inline `apply`

```
sum(node* n, k : kont*) → void {
  if (n == null)
    while (true)
      sa = 0
      match *k with
      | KSBranch0{n, k=k1} ⇒ {
        sum(n.bs[1], KSBranch1{s0=sa, n=n, k=k1})
        return }
      | KSBranch1{s0, n, k=k1} ⇒ {
        sa =s0 + sa + n.v; k=k1 }
      | KTerm ans ⇒ {*ans = sa; return }
    return
  sum(n.bs[0], KSBranch0{n=n, k=k})
```

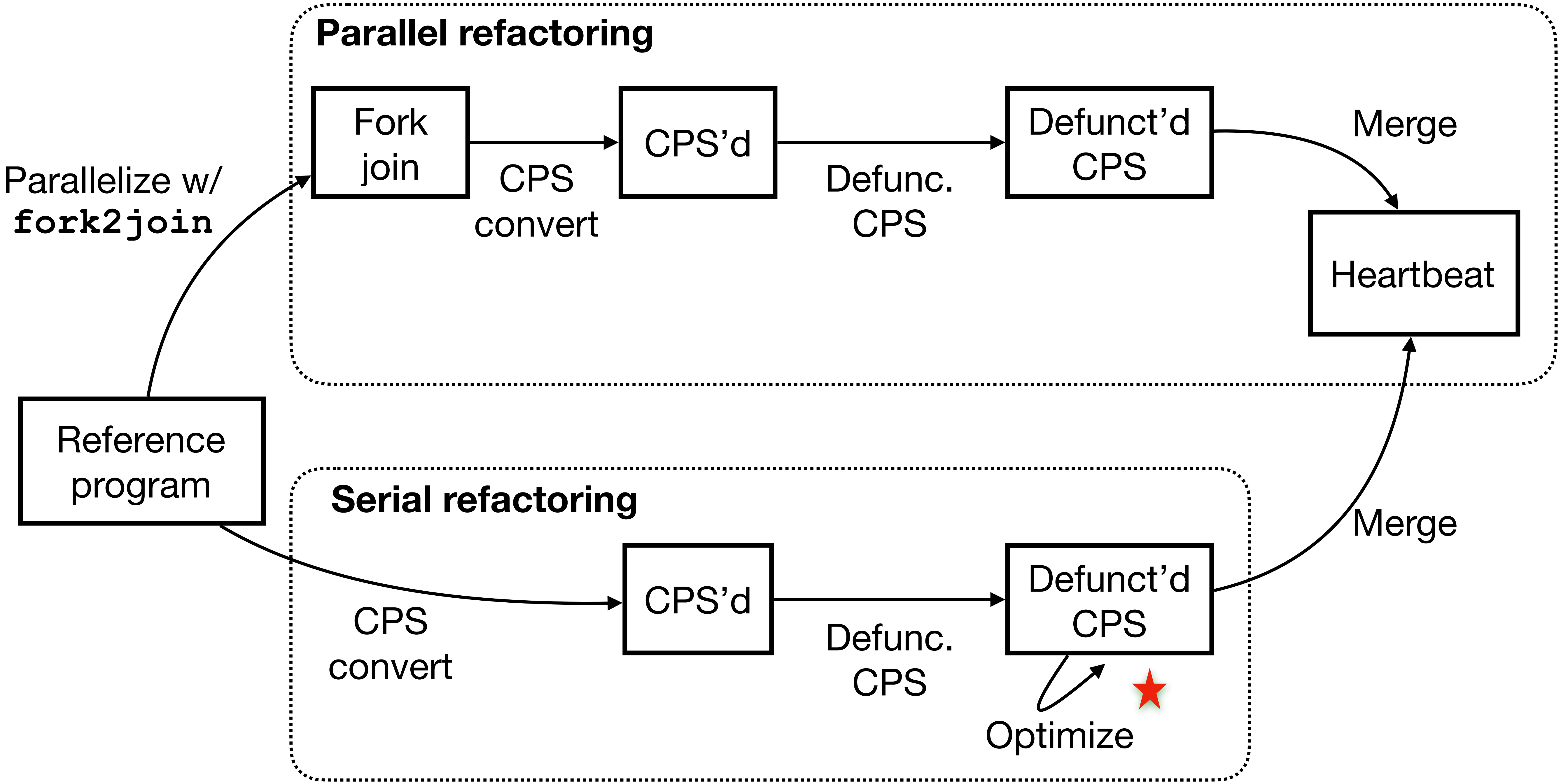
Refactoring for serial traversal

Tail-call elimination of `sum`

```
sum(node* n, k : kont*) → void {
  while (true
    if (n == null)
      while (true)
        sa=0
        match *k with
        | KSBranch0{n=n1, k=k1} ⇒ {
          n = n1.bs[1]; k = KSBranch1{s0=sa, n=n1, k=k1}
          break }
        | KSBranch1{s0, n, k=k1} ⇒ { sa =s0 + sa + n.v; k=k1 }
        | KTerm ans ⇒ { *ans = sa; return }
    else
      k = KSBranch0{n=n, k=k}
      n= n.bs[0] }
```

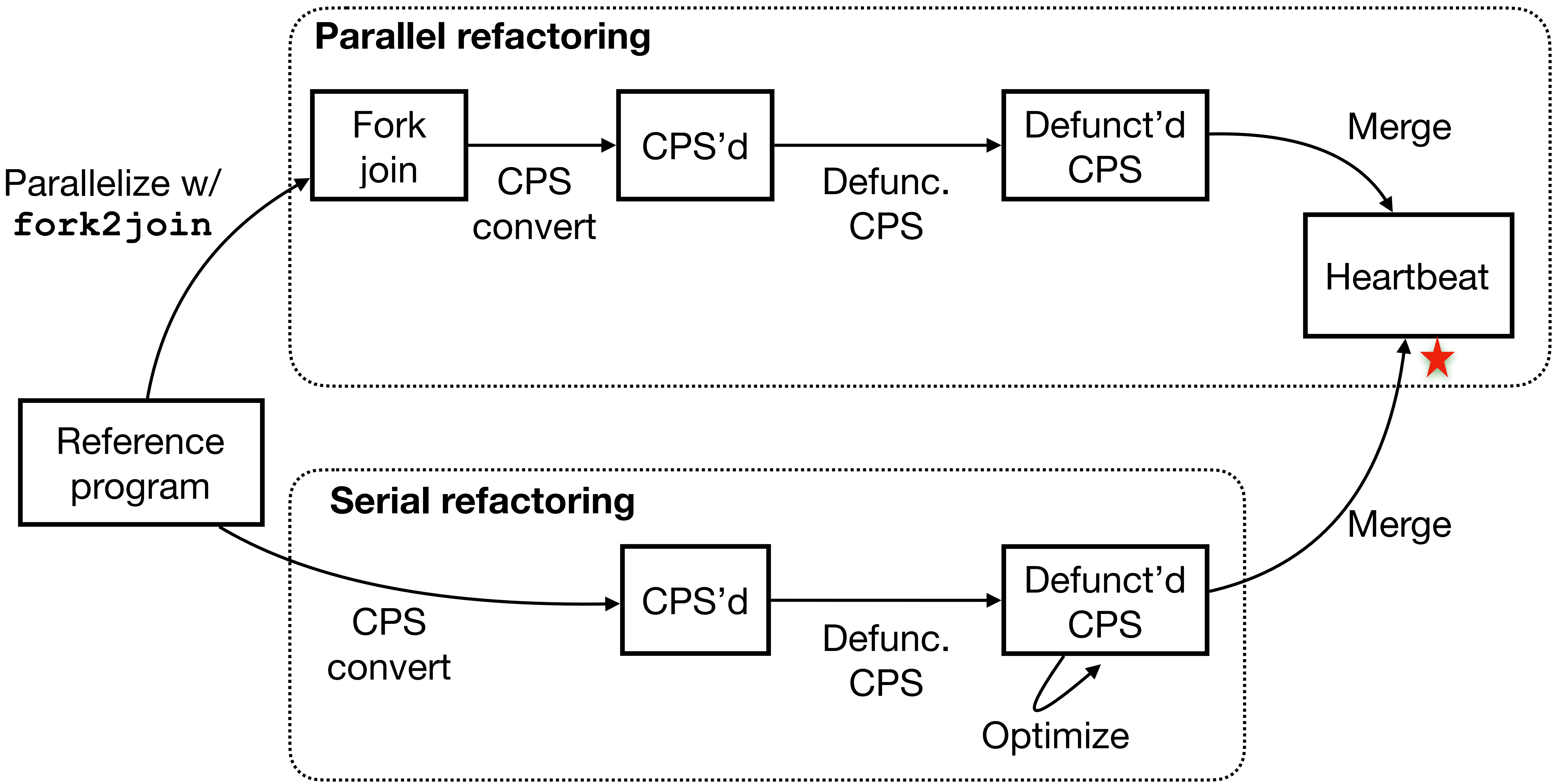
Our multicore-parallelization refactoring

Roadmap



Our multicore-parallelization refactoring

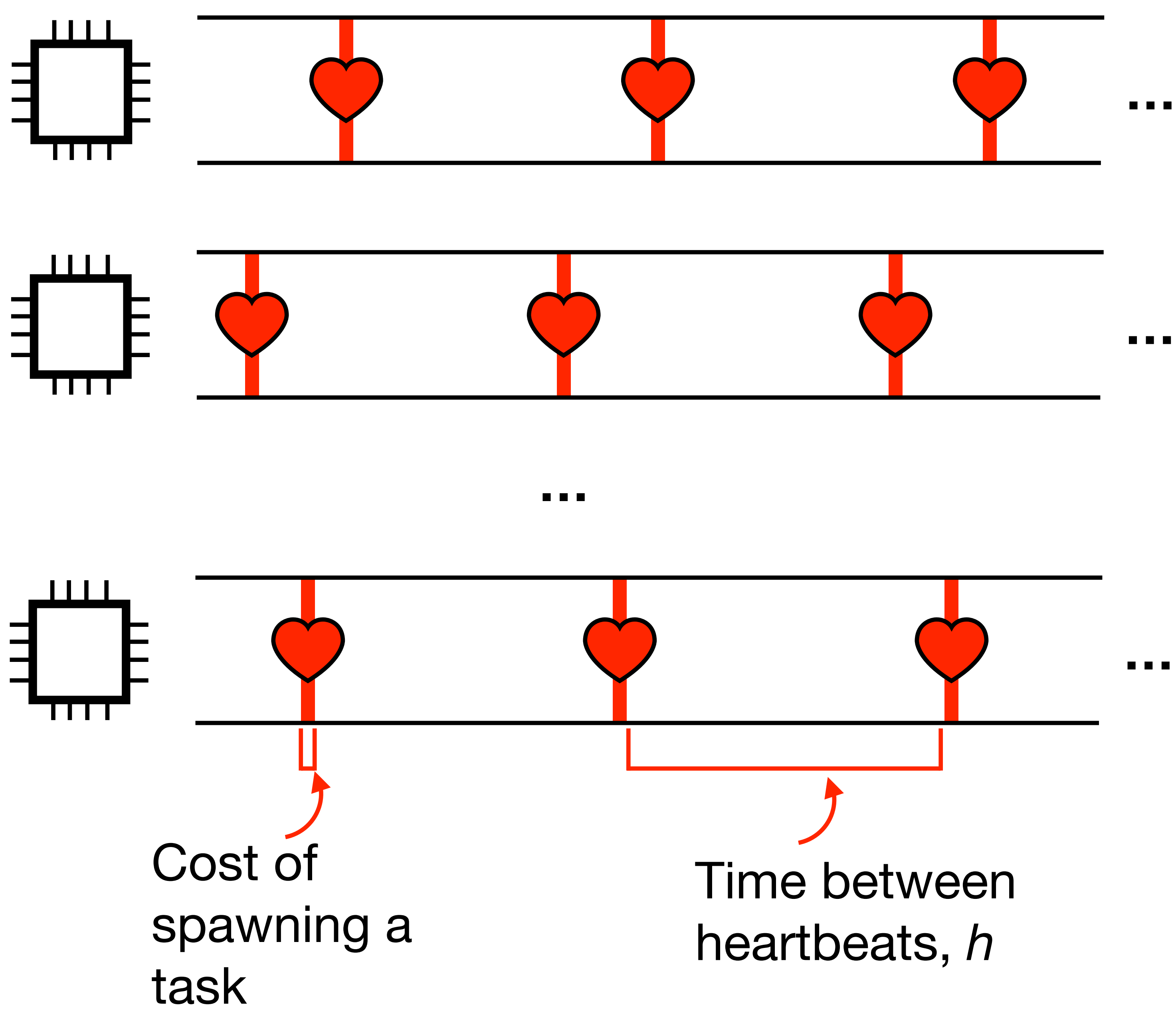
Roadmap



Heartbeat Scheduling

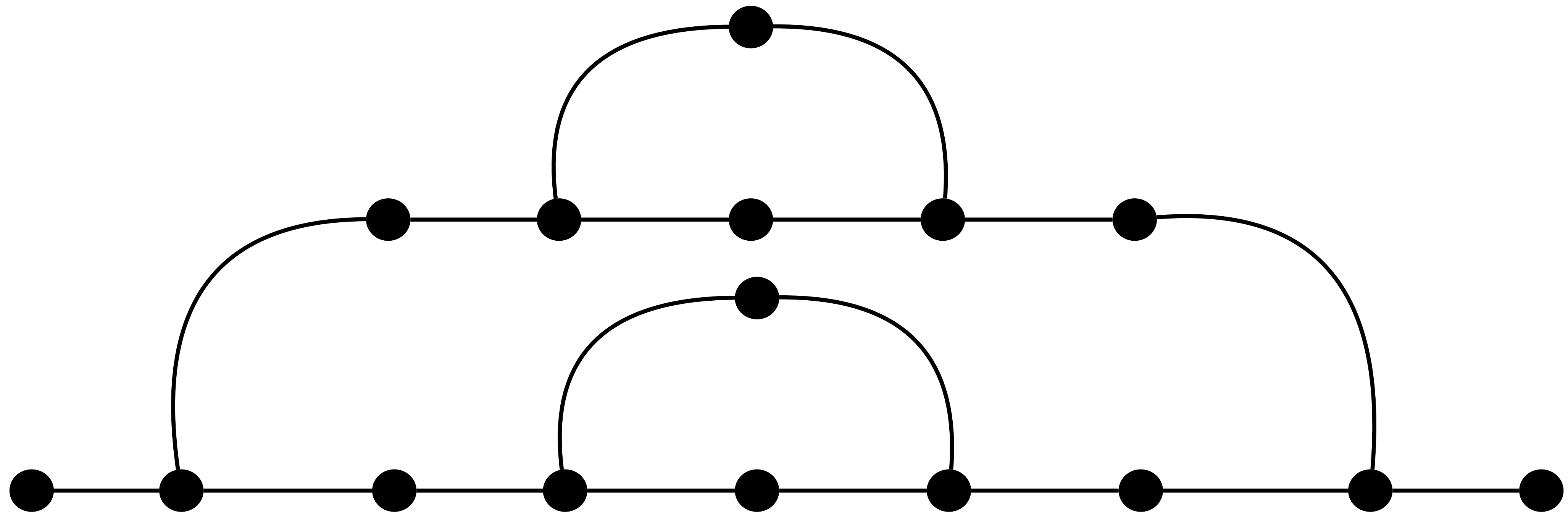
Key idea:

We **amortize** the overheads of parallelism against useful work performed between heartbeats.



Example of Heartbeat Scheduling

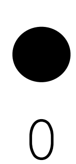
Fully parallel schedule (of, e.g., a balanced tree)



Heartbeat schedule

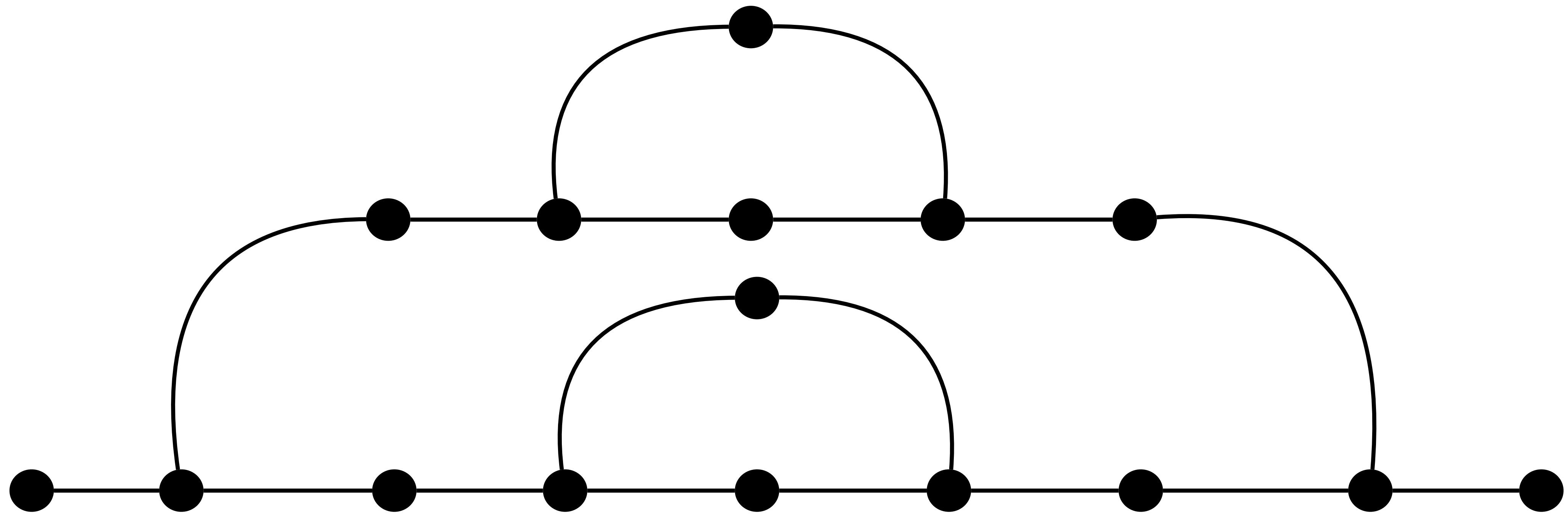
Heartbeat rate

$h = 4$



Example of Heartbeat Scheduling

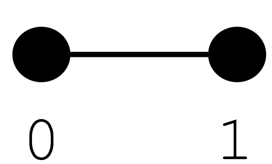
Fully parallel schedule (of, e.g., a balanced tree)



Heartbeat schedule

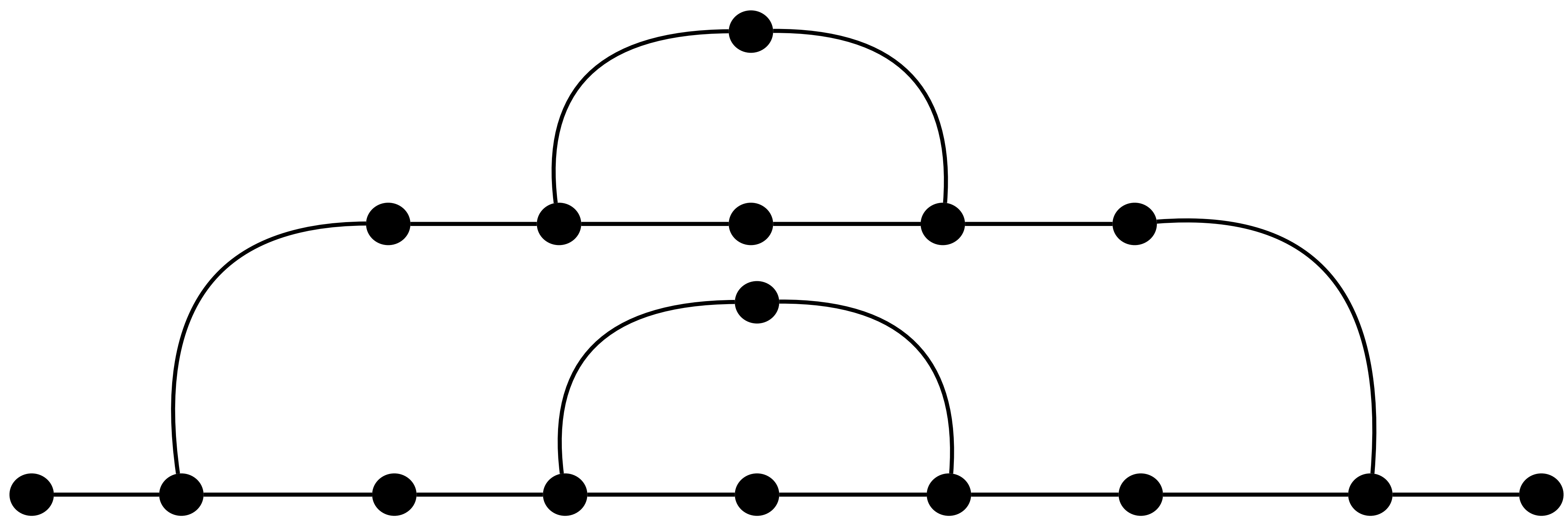
Heartbeat rate

$h = 4$

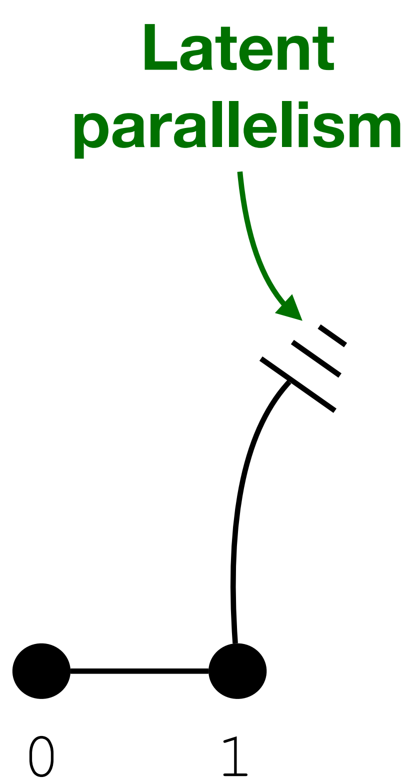


Example of Heartbeat Scheduling

Fully parallel schedule (of, e.g., a balanced tree)



Heartbeat schedule

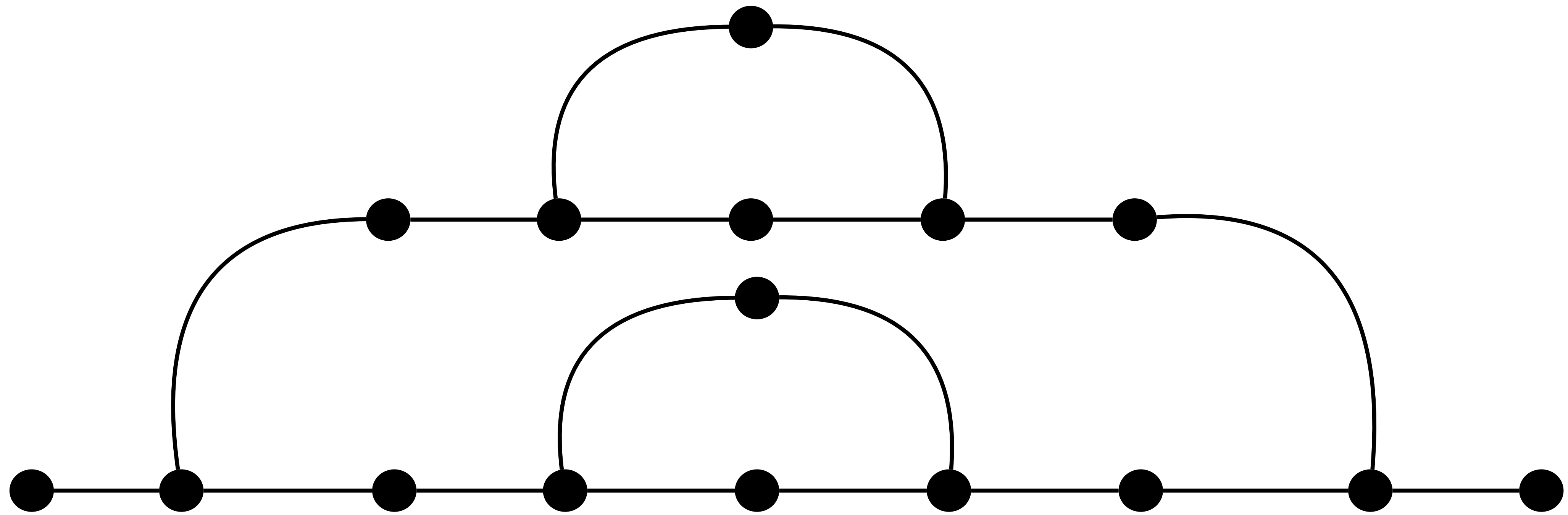


Heartbeat rate

$h = 4$

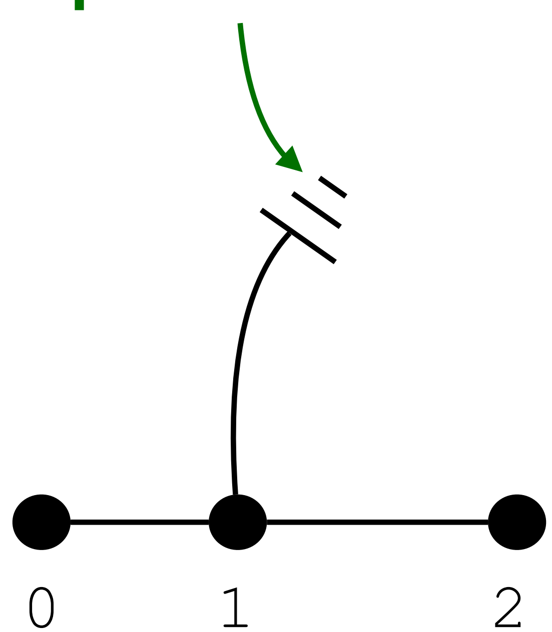
Example of Heartbeat Scheduling

Fully parallel schedule (of, e.g., a balanced tree)



Heartbeat schedule

Latent parallelism

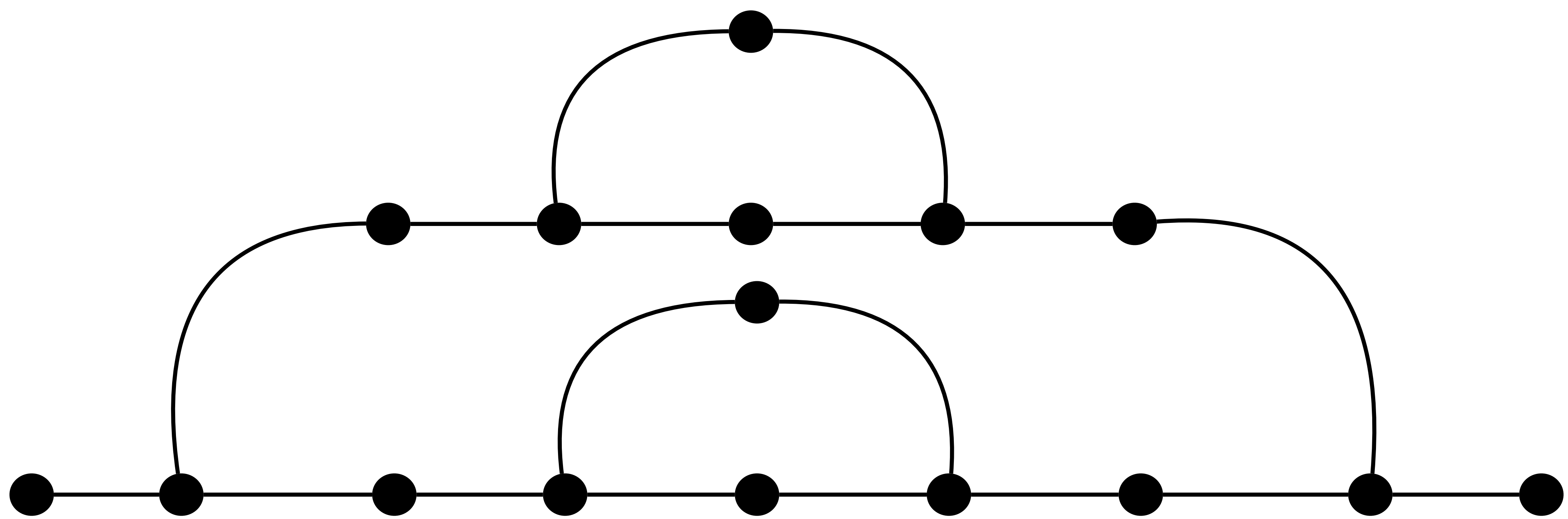


Heartbeat rate

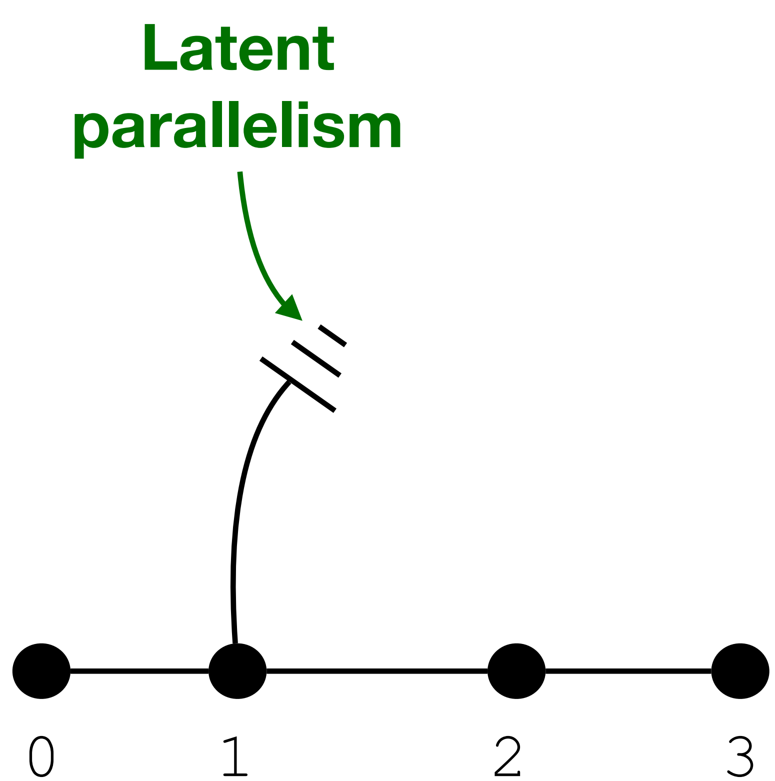
$h = 4$

Example of Heartbeat Scheduling

Fully parallel schedule (of, e.g., a balanced tree)



Heartbeat schedule

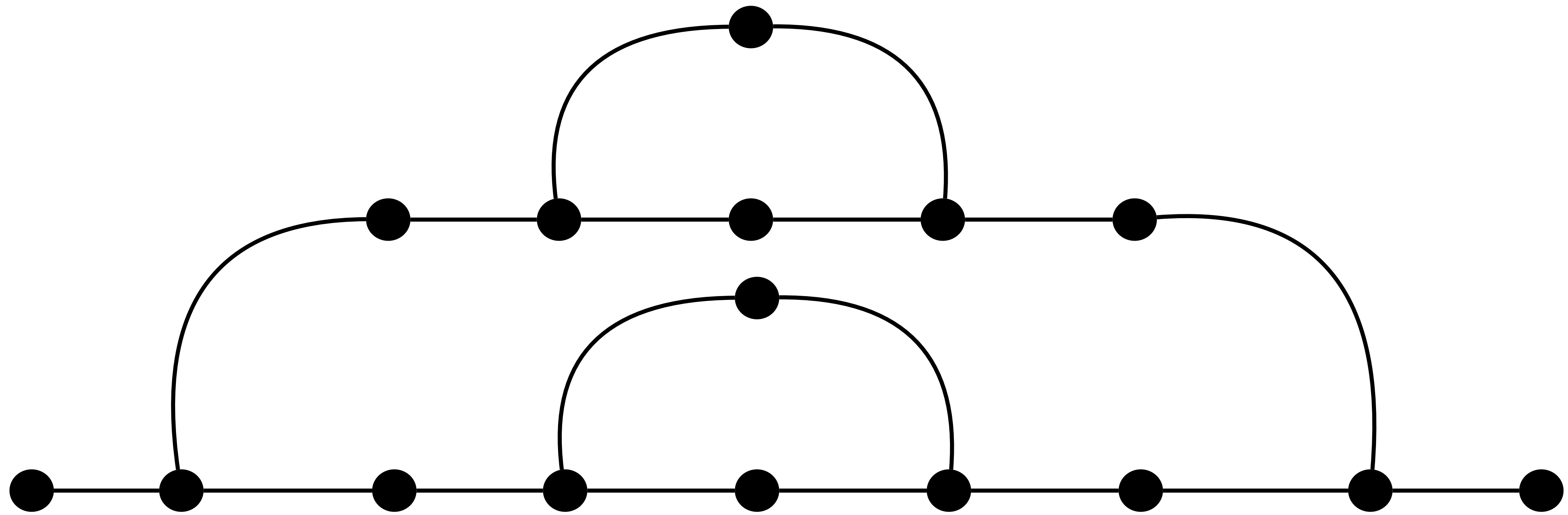


Heartbeat rate

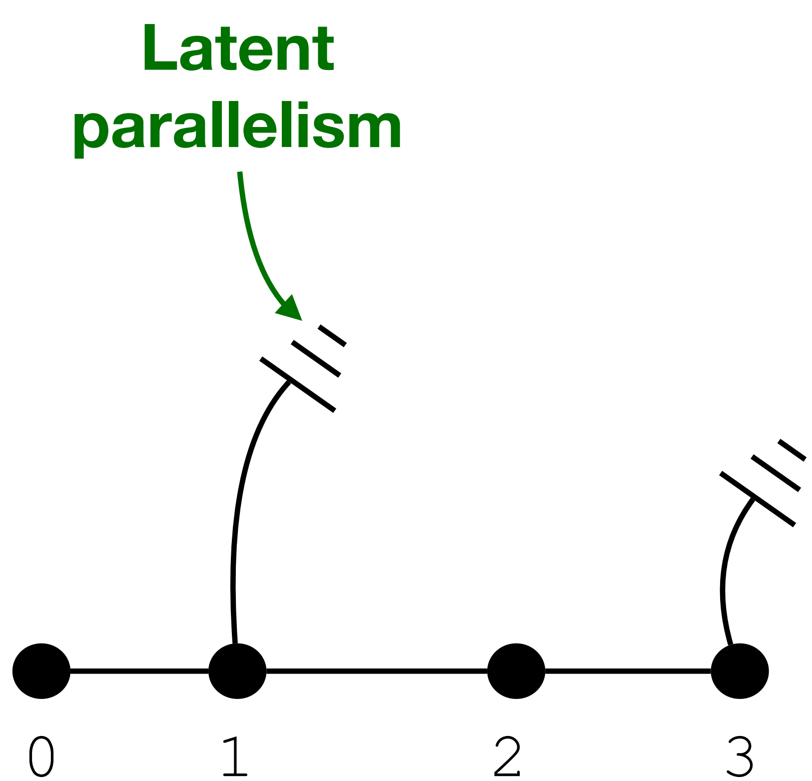
$h = 4$

Example of Heartbeat Scheduling

Fully parallel schedule (of, e.g., a balanced tree)



Heartbeat schedule

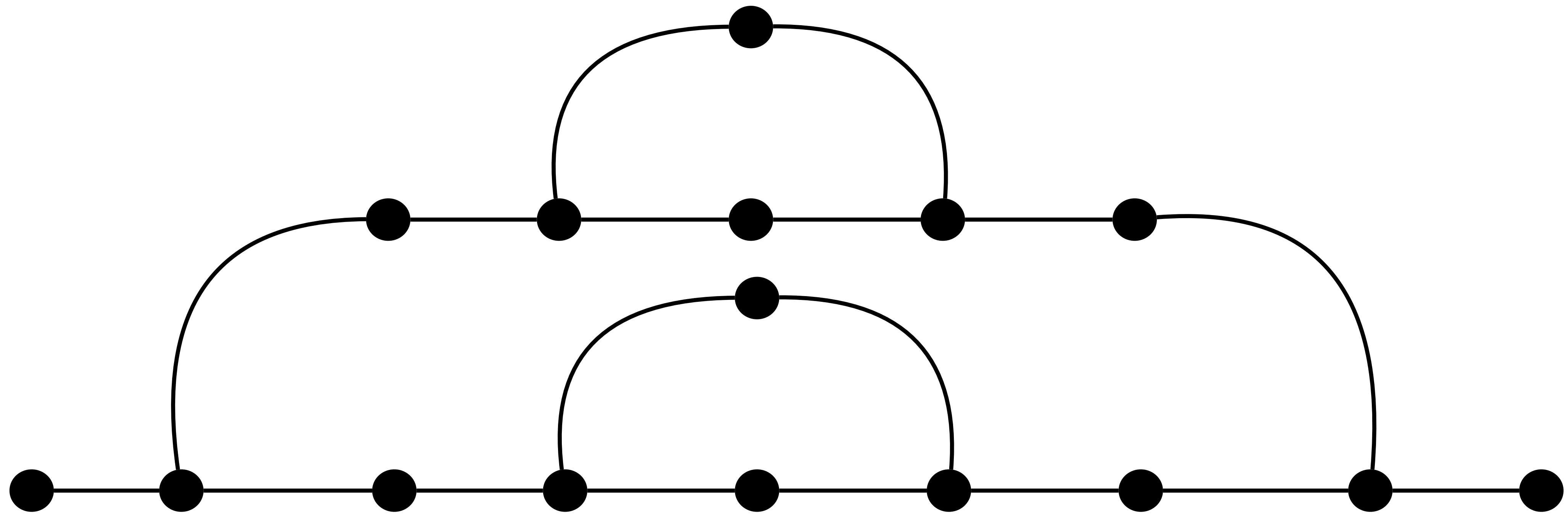


Heartbeat rate

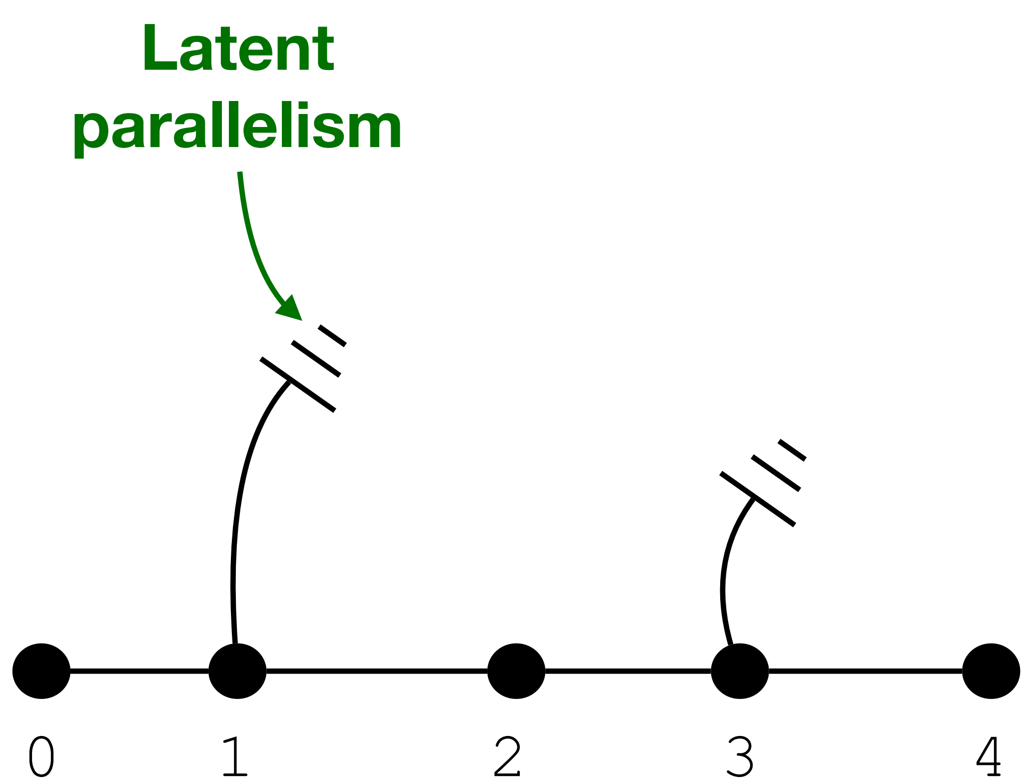
$h = 4$

Example of Heartbeat Scheduling

Fully parallel schedule (of, e.g., a balanced tree)



Heartbeat schedule

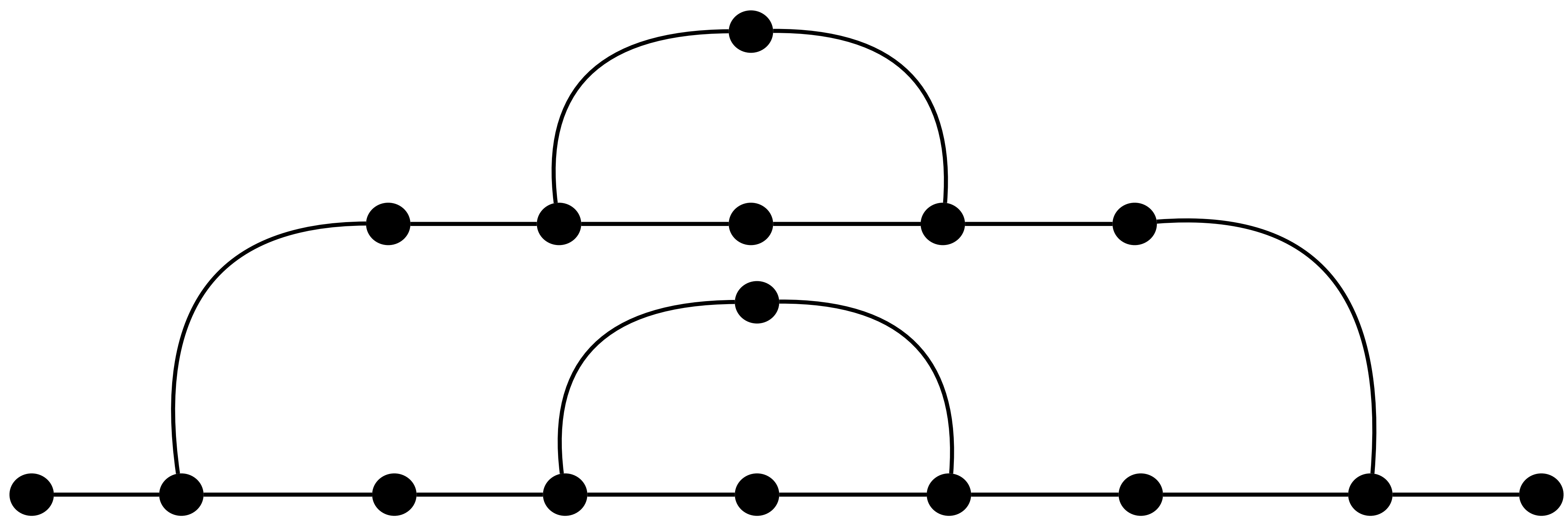


Heartbeat rate

$h = 4$

Example of Heartbeat Scheduling

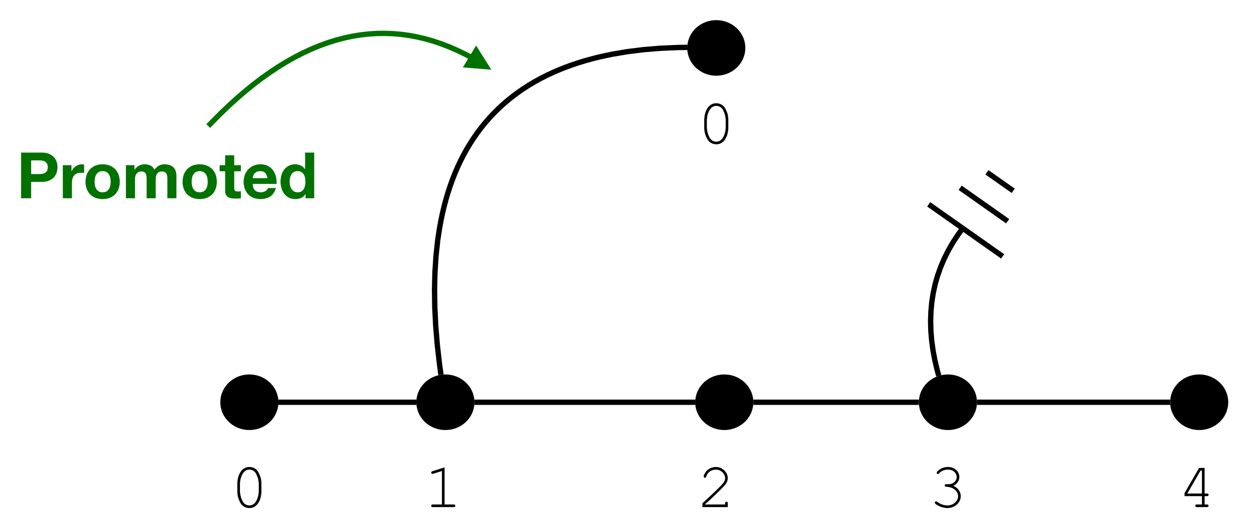
Fully parallel schedule (of, e.g., a balanced tree)



Heartbeat schedule

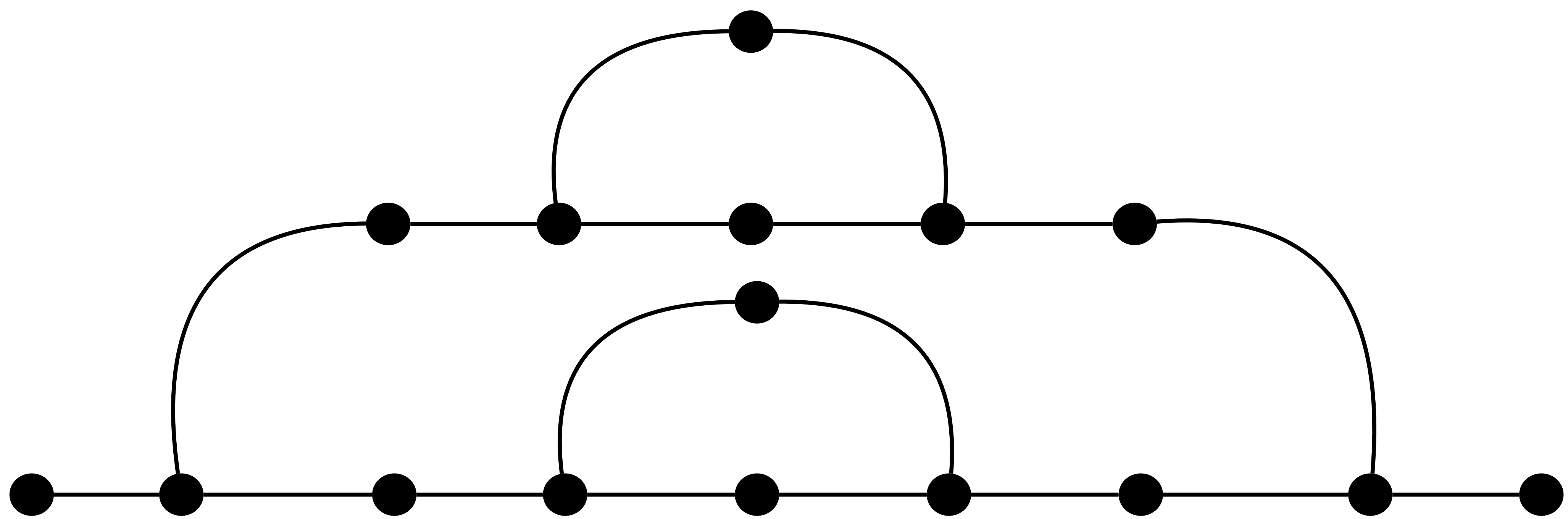
Heartbeat rate

$h = 4$



Example of Heartbeat Scheduling

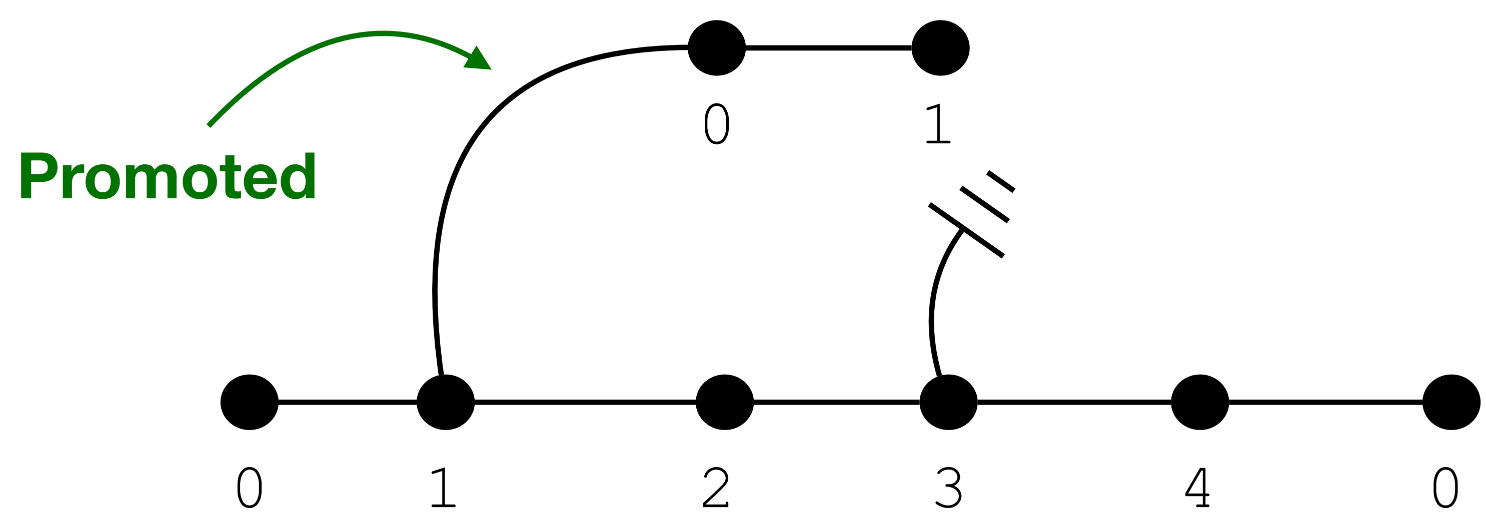
Fully parallel schedule (of, e.g., a balanced tree)



Heartbeat schedule

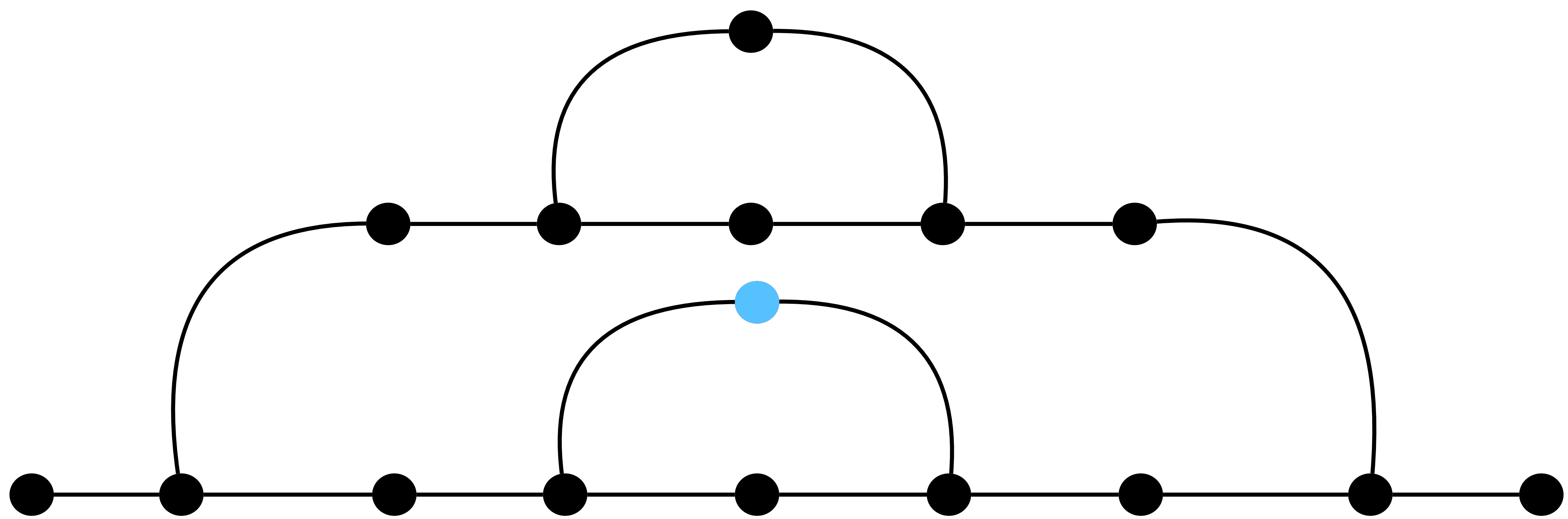
Heartbeat rate

$h = 4$

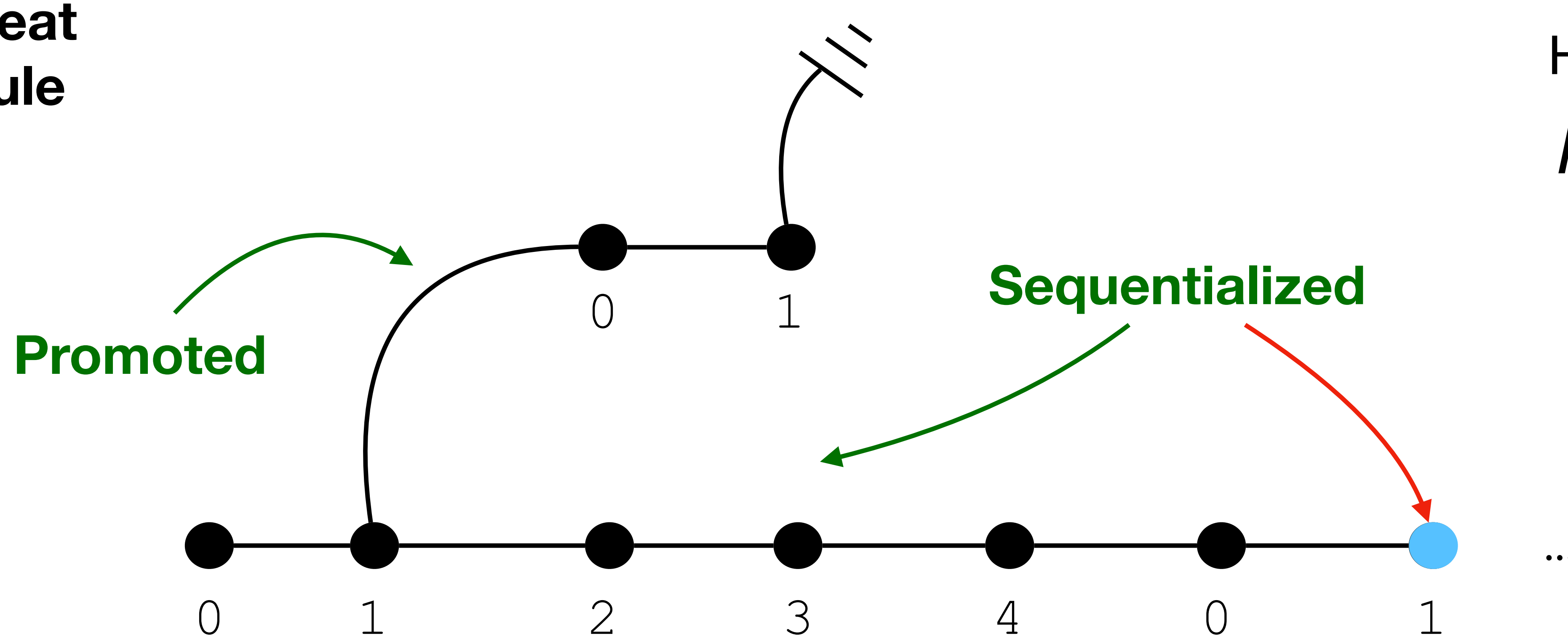


Example of Heartbeat Scheduling

Fully parallel schedule (of, e.g., a balanced tree)



Heartbeat schedule



Heartbeat rate

$h = 4$

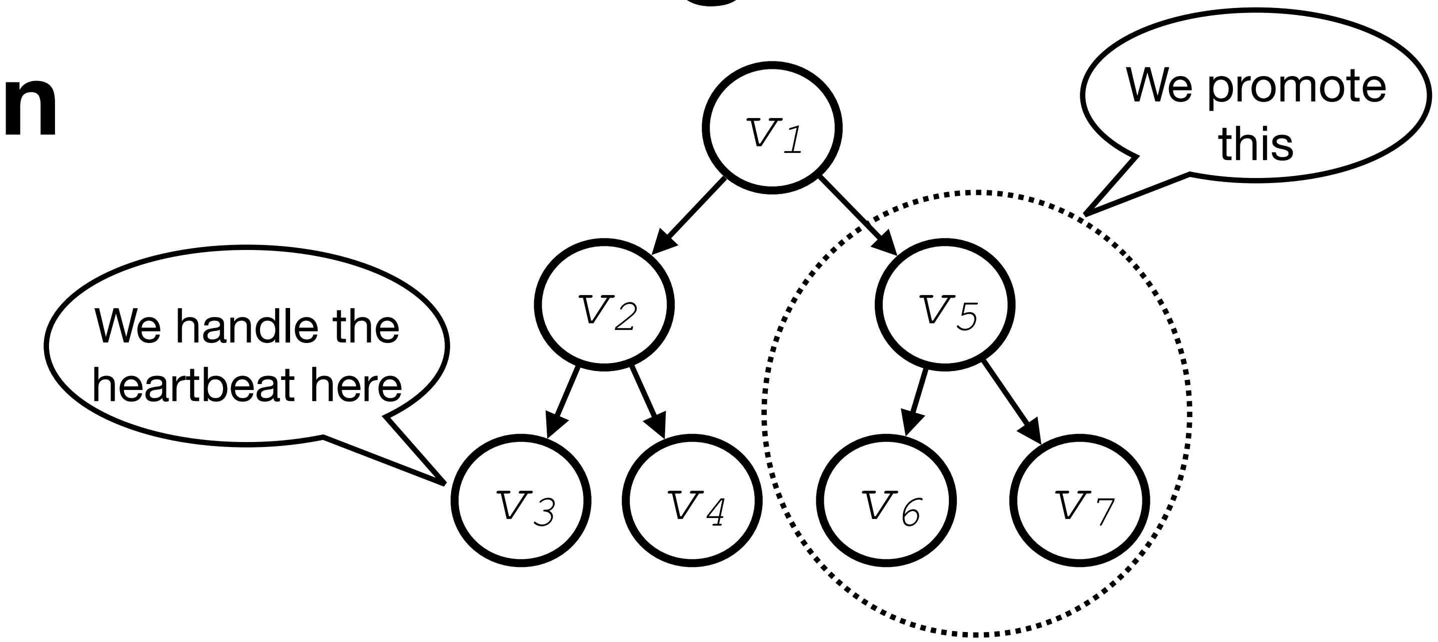
Merging parallel & serial algorithms

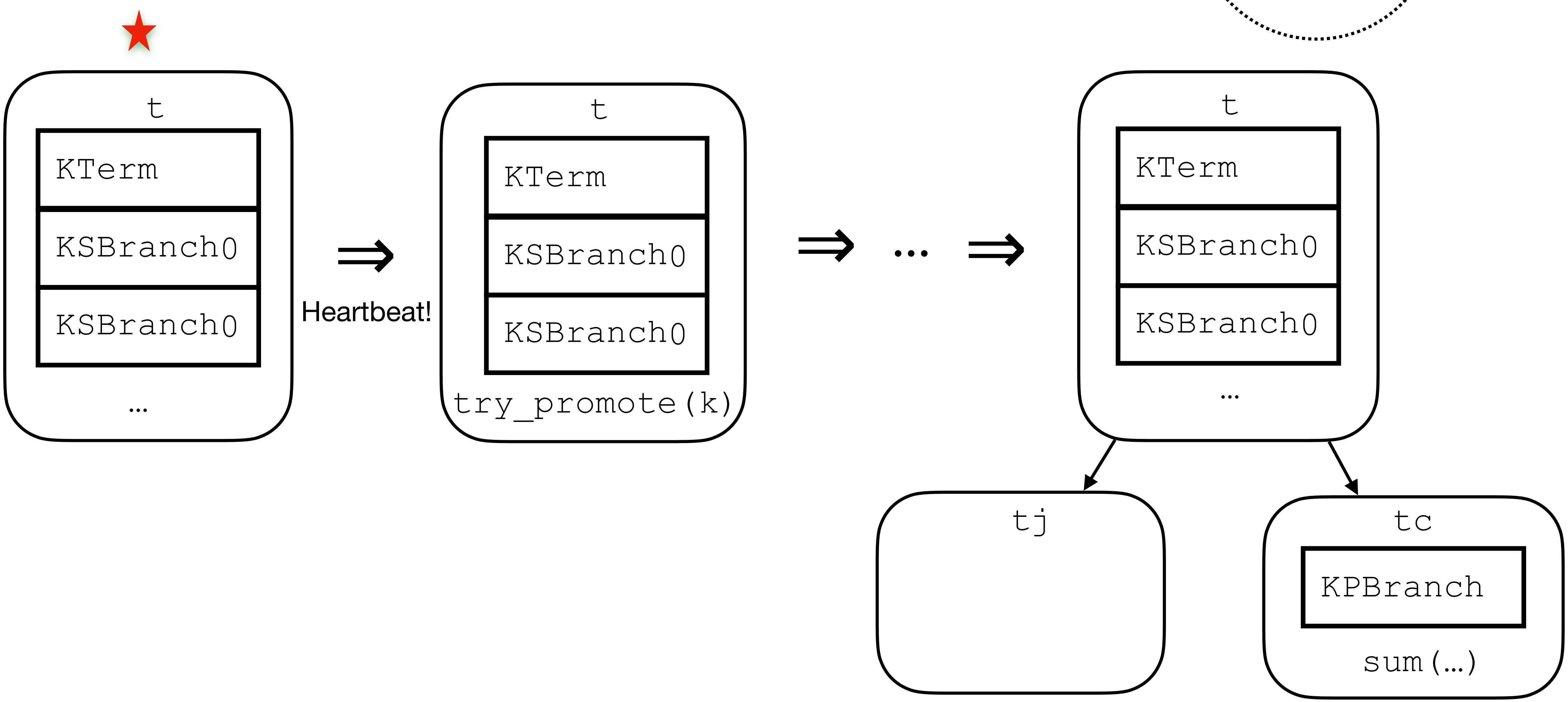
Implementing the heartbeat

```
sum(node* n, k : kont*) → void {
  while (true)
    k = try_promote(k) if heartbeat() else k
    if (n == null)
      sa = 0
      while (true)
        k = try_promote(k) if heartbeat() else k
        match *k with
        | KSBranch0 {n=n1 , k=k1} ⇒ {
          n = n1.bs[1]; k = KSBranch1 {s0 =sa, n=n1 , k=k1}; break }
        | KSBranch1 {s0 , n=n1 , k=k1} ⇒ { s+=s0 +n1.v; k=k1}
        | KPBranch {i, s, tj} ⇒ { s[i] = sa; join(tj); return }
        | KTerm ans ⇒ { *ans = sa }
    else { k = KSBranch0 {n=n, k=k}; n = n.bs[0] } }
```

Merging parallel & serial algorithms

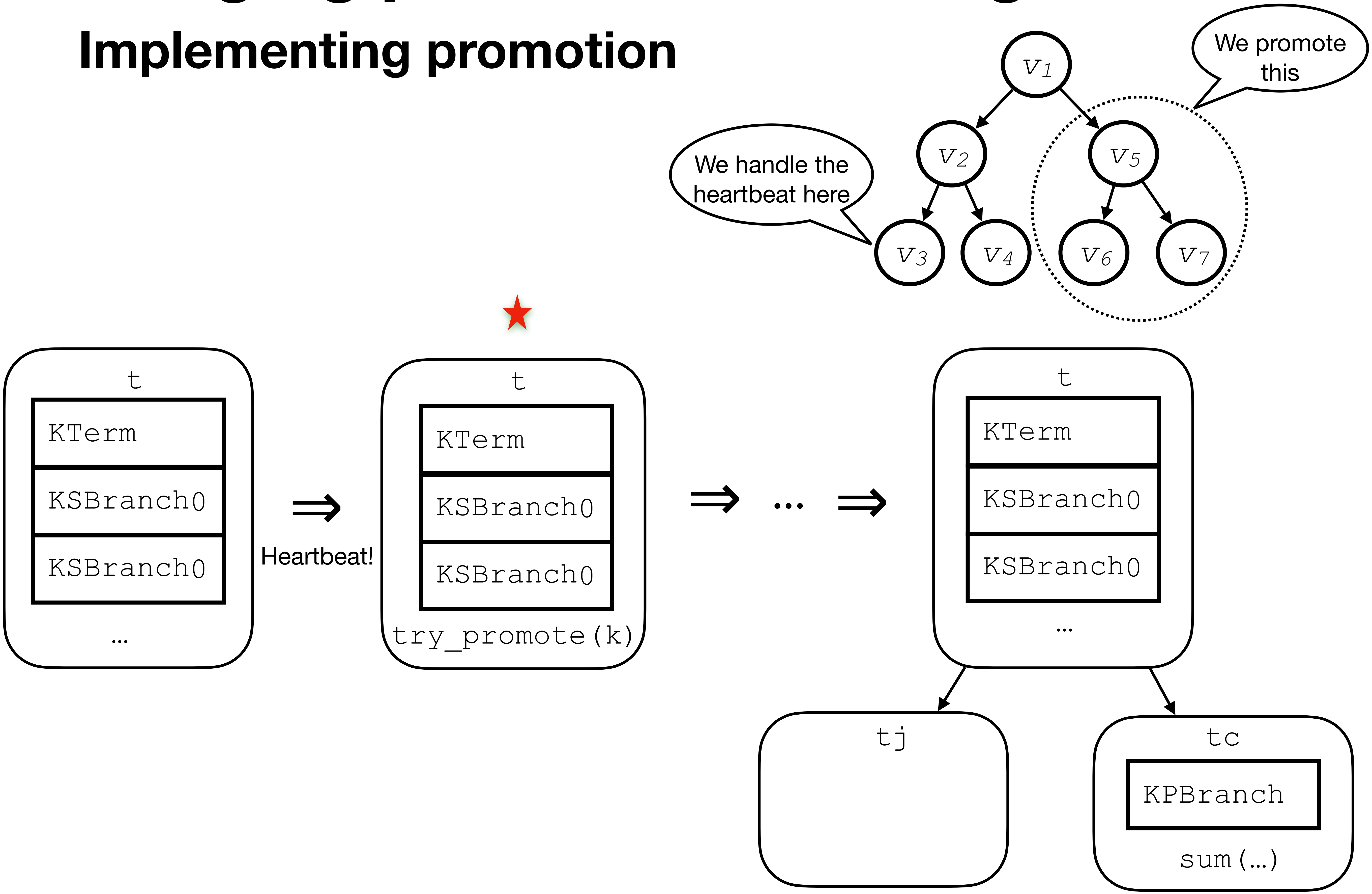
Implementing promotion





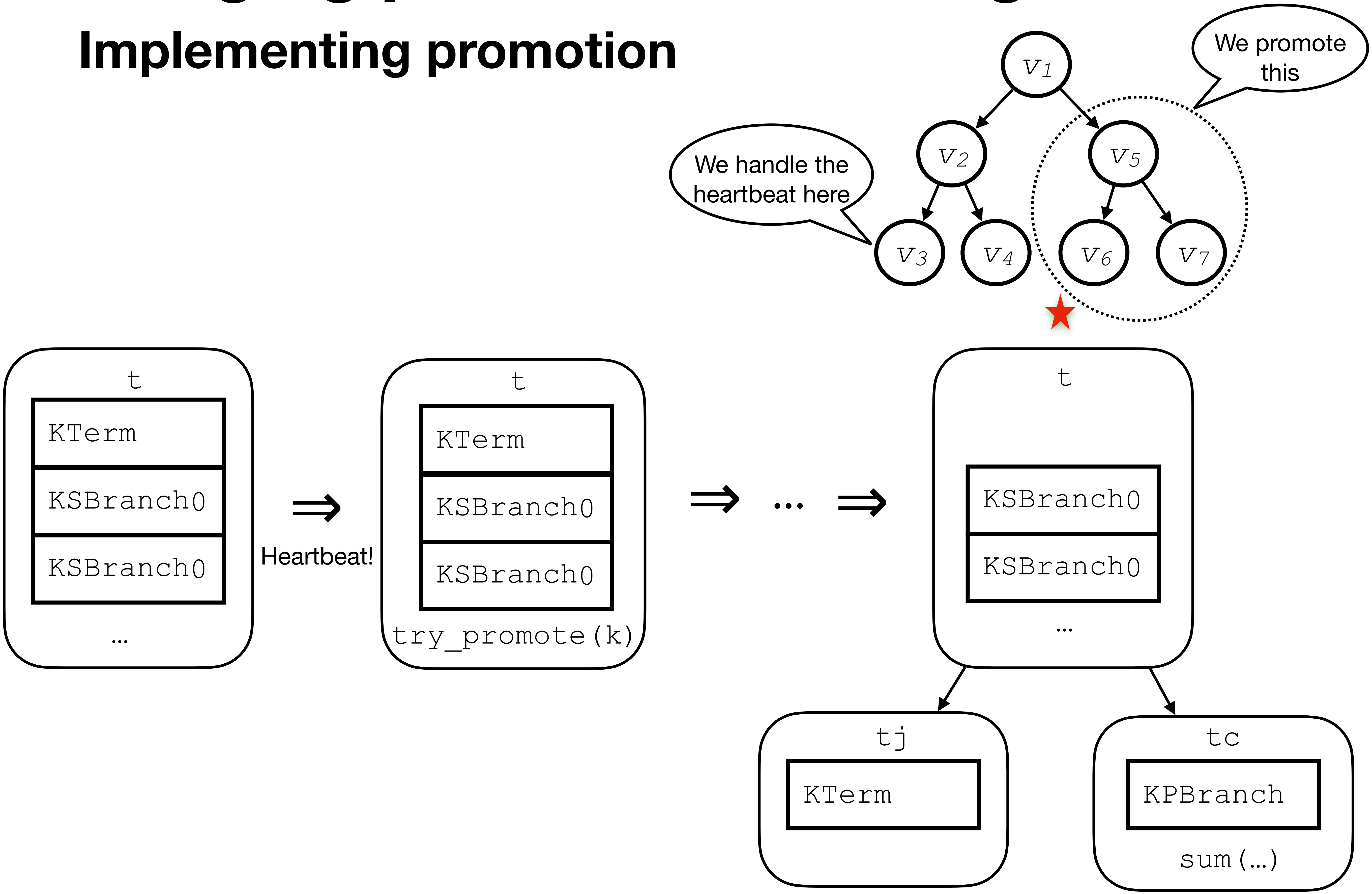
Merging parallel & serial algorithms

Implementing promotion



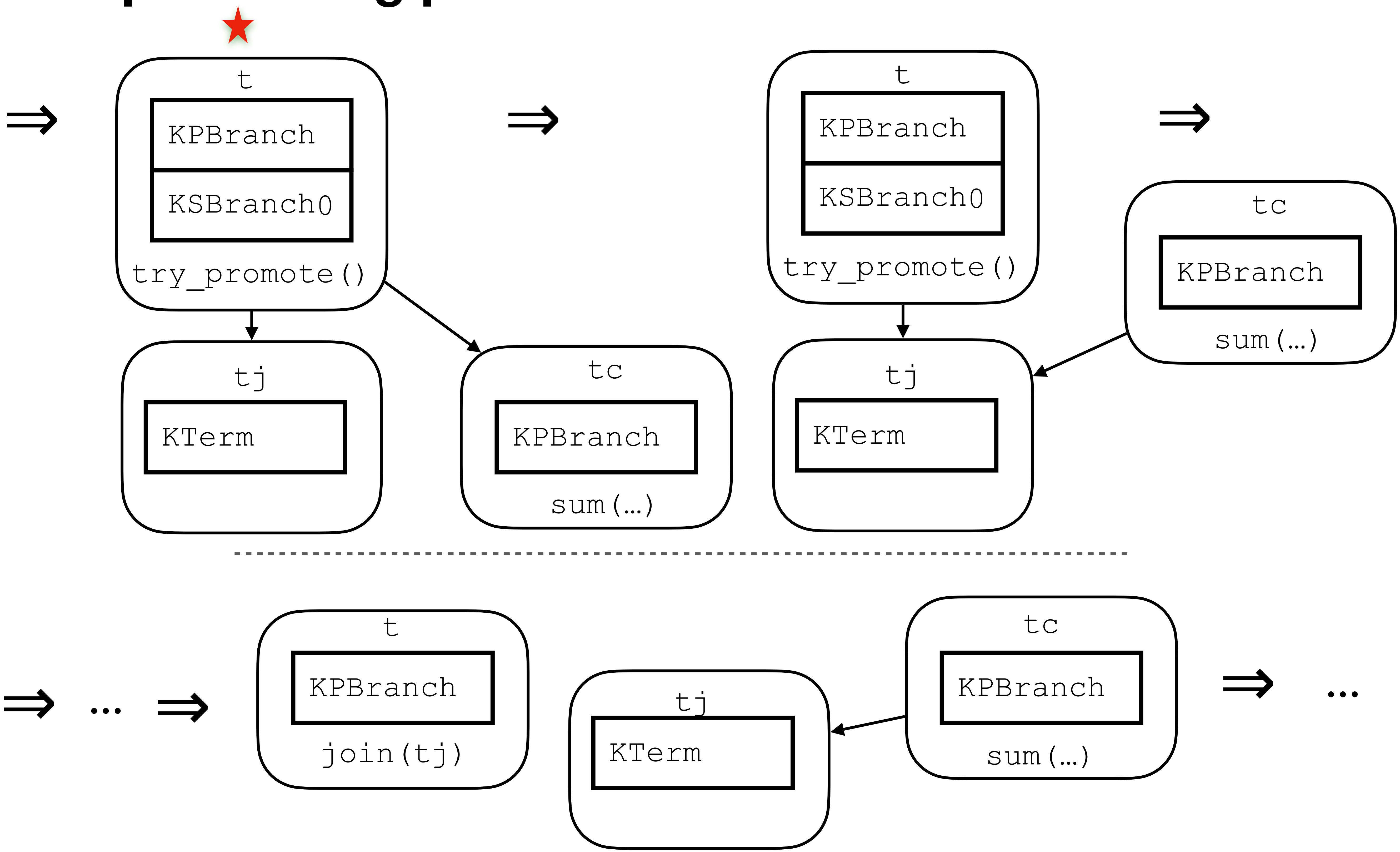
Merging parallel & serial algorithms

Implementing promotion



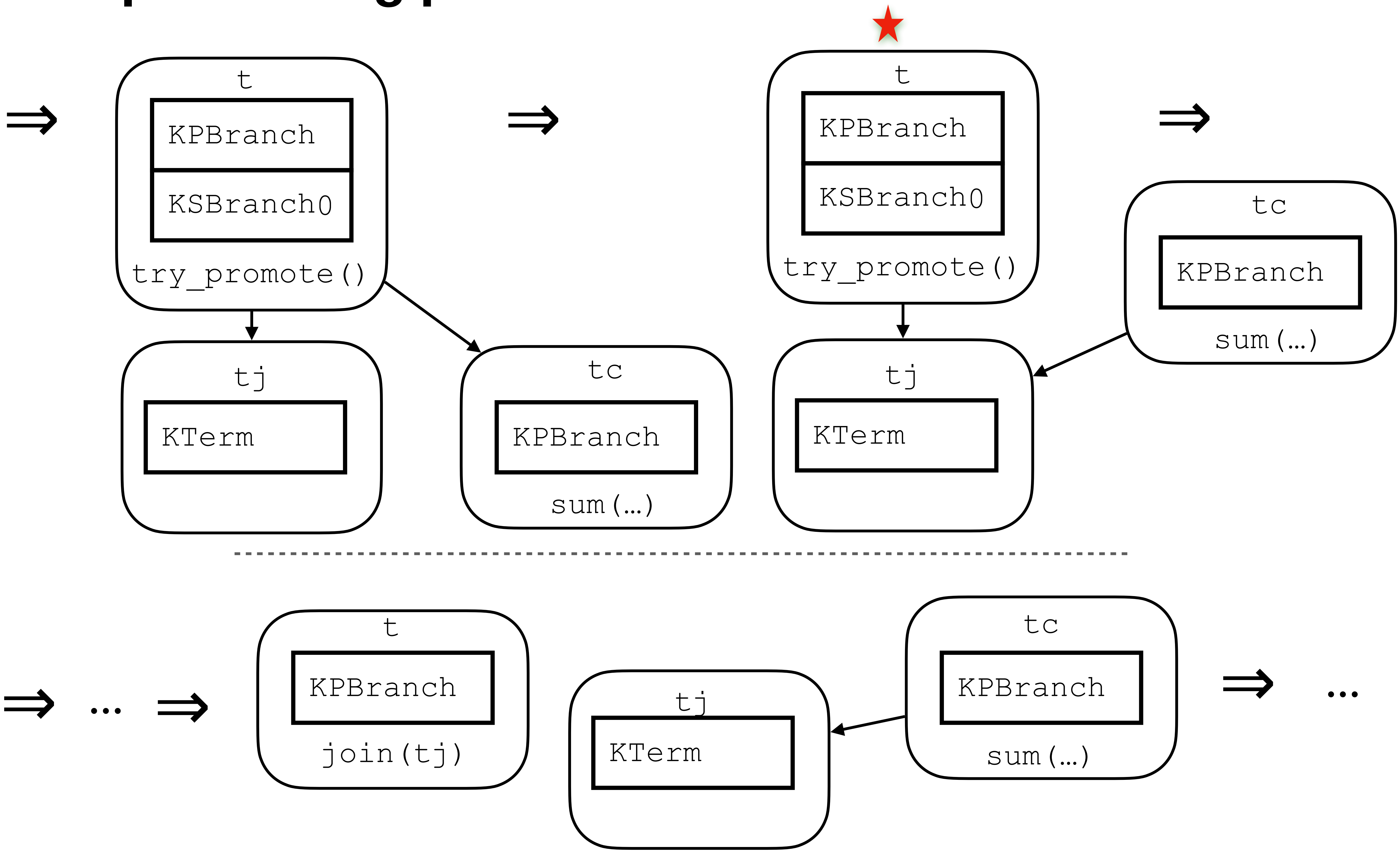
Merging parallel & serial algorithms

Implementing promotion



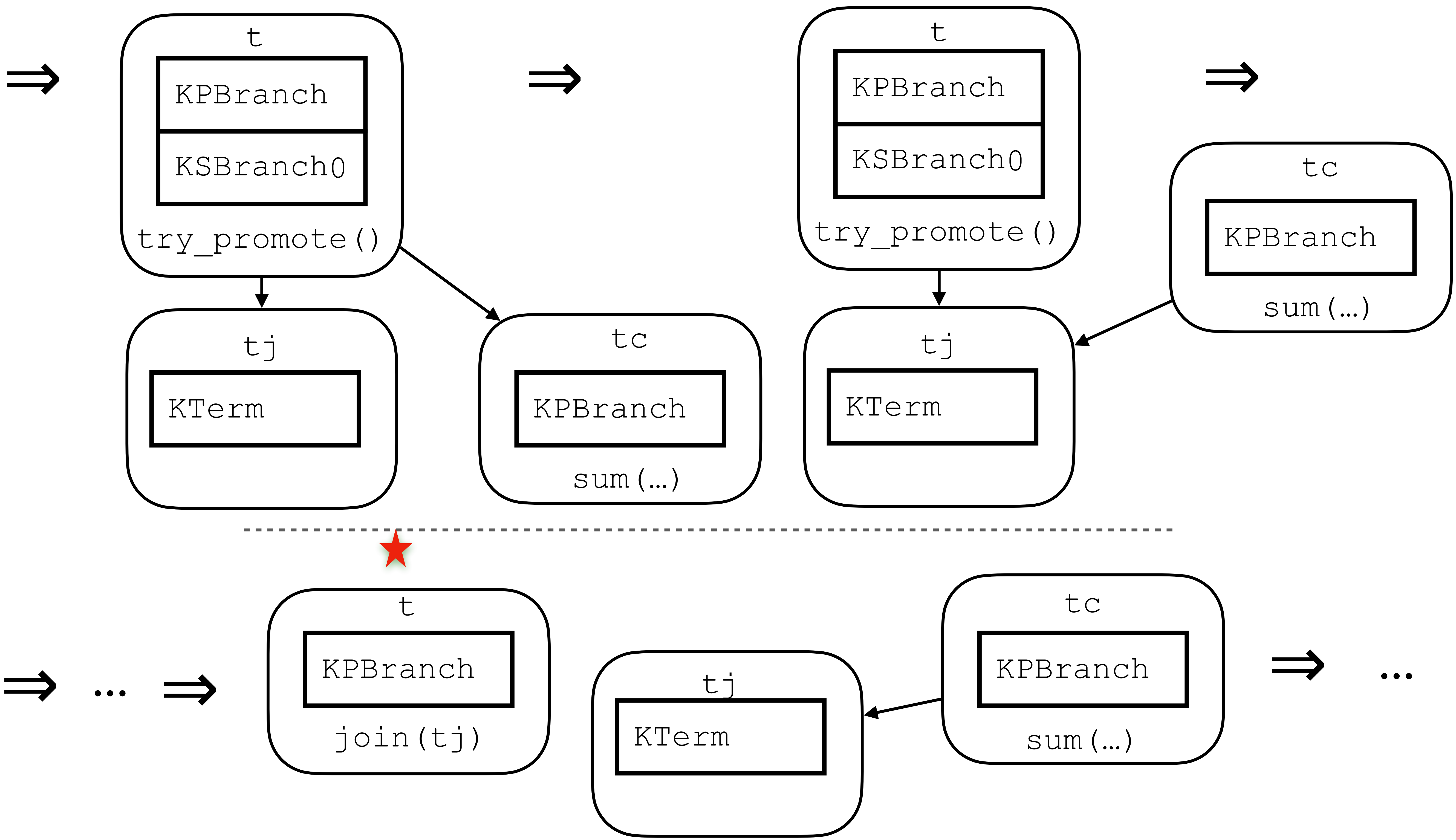
Merging parallel & serial algorithms

Implementing promotion



Merging parallel & serial algorithms

Implementing promotion



Merging parallel & serial algorithms

Handling promotions

```
try_promote(k : kont*) → kont* {
  kt = find_outermost(k, λ k ⇒ {
            match *k with
            | KSBranch0 _ ⇒ true | _ ⇒ false })
  if (kt == null) { return k }
  match *kt with
  | KSBranch0{n, k=kj} ⇒ {
    s = new int[2]
    tj = new_task(λ () ⇒ {
             k0 = KSBranch1{s0=s[0] + s[1], n=n, k=kj}
             sum(null, k0) })
    tc = new_task(λ () ⇒ {
          sum(n.bs[1], KPBranch {i=1, s=s, tj=tj})})
    fork(tc, tj)
    k1 = KPBranch {i=0, s=s, tj=tj}
    return replace(k, kt, k1) } }
```

Benchmarking results

Collected from an Intel Xeon system, using all 64 cores, showing speedup over the iterative, serial algorithm

	input	serial (s)	ours	cilk	cilk+granctrl	
High parallelism	perfect	0.7	28.4x	15.4x	34.5x	
	random	0.8	31.8x	15.3x	33.7x	
Low parallelism	chains	2.5	11.5x	n/a	n/a	Stack overflow
	chain	1.2	0.4x	n/a	n/a	

- perfect is a perfect binary tree of height 27
- random is initially a perfect binary tree built from a series of path-copying insertions targeting random leaves
- chains is a small initial tree of height 20 extended with 30 paths of length 1 million
- chain is a long chain.

Summary

- CPS and defunctionalization are powerful tools for transforming code.
- They can guide code refactoring in various applications.
- This short work identifies multicore parallelization as one.
- We started from one recursive specification, and branched into two refactoring paths: one serial and one parallel.
- We refactored each to get serial efficiency and parallel scalability.
- At the end, we merged the two algorithms using Heartbeat Scheduling, as the conceptual glue.

Current draft:

http://mike-rainey.site/papers/pardefunc.pdf

$$
\begin{aligned}
E\,[\mathtt{fork2join}\,(f_0, f_1)]_k \stackrel{\text{def}}{=} \{ \\
&\mathtt{tj} = \mathtt{new\_task}\,(k) \\
&\mathtt{t}_0 = \mathtt{new\_task}(E\,[f_0\,()]_{\lambda\,()\,.\,\mathtt{join(tj)}}) \\
&\mathtt{t}_1 = \mathtt{new\_task}(E\,[f_1\,()]_{\lambda\,()\,.\,\mathtt{join(tj)}}) \\
&\mathtt{fork}\,(\mathtt{t}_0, \mathtt{tj})\,;\,\mathtt{fork}\,(\mathtt{t}_1, \mathtt{tj})\ \}
\end{aligned}
$$