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

Mike Rainey

FHPNC’23

Current draft:

*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

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.)
Background

Heartbeat Scheduling as a refactoring via CPS+defunctionalization?

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

\[
\text{sum}(\text{node* } n) \rightarrow \text{int} \{
\begin{align*}
&\text{if } (n == \text{null}) \text{ return } 0 \\
&\text{return } \text{sum}(n.bs[0]) + \text{sum}(n.bs[1]) + n.v 
\end{align*}
\]

Our reference program

Enables statements \(s_1\) and \(s_2\) to execute in parallel.

\[
\text{fork2join} \{ \ s_1 \ \} \ \{ \ s_2 \ \}
\]

\(s_3\)

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

We introduce a temporary array to hold intermediate results.

Syntactic sugar for reducing clutter.

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 }
```

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., $s_3$).

```
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 }
```

We introduce a temporary array to hold intermediate results.

Syntactic sugar for reducing clutter.

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

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., $s_3$).

We introduce a temporary array to hold intermediate results.

Recursive calls can execute in parallel.

Syntactic sugar for reducing clutter
Introducing our parallelism primitive

Binary fork join

\[
\text{sum}\left(\text{node}^*\ n\right) \rightarrow \text{int} \{ \\
\quad \text{if } (n == \text{null}) \text{ return } 0 \\
\quad \text{return } \text{sum}\left(\text{n.bs}[0]\right) \\
\quad \quad + \text{sum}\left(\text{n.bs}[1]\right) \\
\quad \quad + n.v \} \\
\]

Enables statements \(s_1\) and \(s_2\) to execute in parallel.

\[
\text{fork2join} \{ \ s_1 \ \} \ \{ \ s_2 \ \} \\
\]

\(s_3\)

...\[...

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

\[
\text{sum}\left(\text{node}^*\ n\right) \rightarrow \text{int} \{ \\
\quad \text{if } (n == \text{null}) \text{ return } 0 \\
\quad s = \text{new int}[2] \\
\quad \{ \ s[i] = \text{sum}\left(\text{n.bs}[i]\right) \} \ i \in \{0, 1\} \\
\quad \text{return } s[0] \\
\quad \quad + s[1] \\
\quad \quad + n.v \} \\
\]

We introduce a temporary array to hold intermediate results.

Syntactic sugar for reducing clutter.

Recursive calls can execute in parallel.
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

V1

V2

V3

...
Our multicore-parallelization refactoring

Roadmap

Parallel refactoring

Reference program → Fork join

- CPS convert

Parallelize w/ fork2join

CPS’d → Defunct’d CPS

Serial refactoring

- Optimize

Merge

Heartbeat

Defunct’d CPS
Our multicore-parallelization refactoring

Roadmap

Parallel refactoring

Reference program

Parallelize w/ fork2join

Fork join → CPS’d → Defunct’d CPS

Serial refactoring

CPS’d → Defunct’d CPS

Optimize

Heartbeat

Merge
Our multicore-parallelization refactoring

Roadmap

Parallel refactoring
- Fork join
- CPS’d
- Defunct’d CPS

Serial refactoring
- CPS convert
- CPS’d
- Defunct’d CPS

Reference program

Parallelize w/ fork2join

Merge

Heartbeat

Optimize
Our multicore-parallelization refactoring

Roadmap

Parallel refactoring

Reference program

Serial refactoring

Heartbeat

Parallelize w/ fork2join
Fork-join primitives

Library interface

\[ t = \text{new\_task}(f) \]  
Creates a new task \( t \) that, when run will invoke its thunk \( f \).

\[ \text{fork}(tc, tj) \]  
Registers one dependency edge from the current task to join task \( tj \) and one from child task \( tc \) to \( tj \), and schedules \( tc \).

\[ \text{join}(tj) \]  
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

\[
t = \text{new}_\text{task}(f)
\]
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 \( tj \)

tj = \text{new}_\text{task}(fj)
tc = \text{new}_\text{task}(fc)
fork(tc, tj)
**Fork-join primitives**

**Library interface**

\[
t = \text{new\_task}(f)
\]

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 \( tj \)

\[
tj = \text{new\_task}(fj)
\]

\[
tc = \text{new\_task}(fc)
\]

\[
fork(tc, tj)
\]
Fork-join primitives

Library interface

\[ t = \text{new_task}(f) \]  
Creates a new task \( t \) that, when run will invoke its thunk \( f \).

\[ \text{fork}(tc, tj) \]  
Registers one dependency edge from the current task to join task \( tj \) and one from child task \( tc \) to \( tj \), and schedules \( tc \).

\[ \text{join}(tj) \]  
Resolves one dependency edge on join task \( tj \).

Example: task \( t \) spawns a child task \( tc \) with join point \( tj \)

Current task \( t \)

\[ tj = \text{new_task}(fj) \]

\[ tc = \text{new_task}(fc) \]

\[ \text{fork}(tc, tj) \]
Fork-join primitives

Library interface

\[ t = \text{new_task}(f) \]  
Creates a new task \( t \) that, when run will invoke its thunk \( f \).

\[ \text{fork}(tc, tj) \]  
Registers one dependency edge from the current task to join task \( tj \) and one from child task \( tc \) to \( tj \), and schedules \( tc \).

\[ \text{join}(tj) \]  
Resolves one dependency edge on join task \( tj \).

---

**Example:** task \( t \) spawns a child task \( tc \) with join point \( tj \)

- \( t \)
- \( t \rightarrow tj \)
  - \( tj = \text{new_task}(fj) \)
- \( t \rightarrow tc \)
  - \( tc = \text{new_task}(fc) \)
- \( \text{fork}(tc, tj) \)

- \( t \rightarrow \ldots \rightarrow tj \)
  - \( \text{join}(fj) \)
- \( t \rightarrow \ldots \rightarrow \ldots \rightarrow tj \)
  - \( \ldots \)
Fork-join primitives

Library interface

\[ t = \text{new\_task}(f) \] Creates a new task \( t \) that, when run will invoke its thunk \( f \).

\[ \text{fork}(tc, tj) \] Registers one dependency edge from the current task to join task \( tj \) and one from child task \( tc \) to \( tj \), and schedules \( tc \).

\[ \text{join}(tj) \] Resolves one dependency edge on join task \( tj \).

**Example:** task \( t \) spawns a child task \( tc \) with join point \( tj \)

\[
\begin{align*}
  & t = \text{new\_task}(f) \\
  & \text{fork}(tc, tj) \\
  & \text{join}(tj)
\end{align*}
\]
Fork-join primitives

Library interface

\[ t = \text{new\_task}(f) \]

- Creates a new task \( t \) that, when run will invoke its thunk \( f \).

\[ \text{fork}(tc, tj) \]

- Registers one dependency edge from the current task to join task \( tj \) and one from child task \( tc \) to \( tj \), and schedules \( tc \).

\[ \text{join}(tj) \]

- Resolves one dependency edge on join task \( tj \).

**Example:** task \( t \) spawns a child task \( tc \) with join point \( tj \)

```
\begin{tikzpicture}
  \node (t) at (0,0) {\( t \)};
  \node (tj) at (1,-1) {\( tj \)};
  \draw[->] (t) -- (tj);
  \node (tc) at (2,0) {\( tc \)};
  \node (fj) at (3,-1) {\( fj \)};
  \draw[->] (t) -- (fj);
  \node (tj) at (4,-1) {\( tj \)};
  \draw[->] (fj) -- (tj);
  \node (tc) at (5,0) {\( tc \)};
  \draw[->] (tc) -- (tj);
\end{tikzpicture}
```
Refactoring for parallel traversal
CPS convert the parallel algorithm

sum(node* n) \rightarrow \text{int} \{
    \text{if (n == null) return 0}
    \text{s = new int[2]}
    \text{fork2join} \{
        \text{s[i] = sum(n.bs[i])}
    \} \text{i} \in \{0, 1\}
    \text{return s[0] + s[1] + n.v}
\}

sum(node* n, k : \text{int} \rightarrow \text{void}) \rightarrow \text{void} \{
    \text{if (n == null) } \{ \text{k(0); return} \}
    \text{s = new int[2]}
    \text{tj = new_task(\lambda () \Rightarrow}
        \text{k(s[0] + s[1] + n.v))}
    \text{\{ t_i = new_task(\lambda () \Rightarrow}
        \text{sum(n.bs[i], \lambda s_i \Rightarrow}
            \text{s[i] = s_i; join(tj))}
        \text{fork(t_i, tj)} \} \text{i} \in \{0, 1\} \}

Our multicore-parallelization refactoring

Roadmap

Parallel refactoring
1. Fork join
   - Parallelize w/ fork2join
2. CPS’d
   - CPS convert
3. Defunct’d CPS
   - Defunct CPS

Serial refactoring
1. CPS’d
   - CPS convert
2. Defunct’d CPS
   - Defunct CPS
   - Optimize

Merge

Heartbeat
Our multicore-parallelization refactoring

Roadmap

Parallel refactoring

Fork join → CPS’d → Defunct’d CPS

Serial refactoring

CPS convert → CPS’d → Defunct’d CPS

Parallelize w/ fork2join

Reference program

Merge

Heartbeat

Optimize
Refactoring for parallel traversal
Defunctionalization of CPS

```plaintext
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} }

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*}
```

Top-level call with input n0; final result pointed to by ans
Refactoring for parallel traversal

Defunctionalization of CPS

```plaintext
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} }

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

Roadmap

Parallel refactoring

Reference program

Parallelize w/ fork2join

Fork join → CPS’d → Defunct’d CPS

CPS convert → Defunc. CPS

Serial refactoring

CPS convert → CPS’d → Defunct’d CPS

Optimize

Heartbeat
Our multicore-parallelization refactoring

Roadmap

Parallel refactoring

Reference program

Parallelize w/ fork2join

Fork join

CPS’d

CPS convert

Defunc. CPS

Defunct’d CPS

Heartbeat

Serial refactoring

CPS convert

CPS’d

Defunc. CPS

Optimize

Merge

Optimize
Our multicore-parallelization refactoring Roadmap

Parallel refactoring

Reference program

Serial refactoring

Parallelize w/ fork2join
Refactoring for serial traversal
CPS convert the serial algorithm

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

```c
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

Parallel refactoring
- Fork join
- CPS’d
- Defunct’d CPS
- Heartbeat
- Merge

Serial refactoring
- CPS’d
- Defunct’d CPS
- Optimize

Reference program

Parallelize w/ fork2join
Our multicore-parallelization refactoring

Roadmap

Parallel refactoring:
- Fork join
- CPS’d
- Defunct’d CPS

Serial refactoring:
- CPS
- CPS’d
- Defunct’d CPS

Optimize

Parallelize w/ fork2join

Reference program

Merge

Heartbeat
Refactoring for serial traversal
Defunctionalization of CPS

```c
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

\[
\text{sum}(\text{node}* \ n, \ k : \ kont*) \rightarrow \text{void} \ 
\begin{align*}
\text{if} \ (n == \text{null}) \ {\text{\{} \ apply(k,\ 0); \ \text{return} \ \}\text{}} \\
\text{sum}(n.\text{bs}[0], \ K\text{SBranch0}\{n=n, \ k=k\})
\end{align*}
\]

\[
\text{apply}(\text{kont}* \ k, \ sa : \text{int}) \rightarrow \text{void} \ 
\begin{align*}
\text{match} \ *k \ \text{with} \\
| \ K\text{SBranch0}\{n, \ k=k1\} \ \text{⇒} \ {\text{\{} \\
\quad \text{sum}(n.\text{bs}[1], \ K\text{SBranch1}\{s_0=sa, \ n=n, \ k=k1\}) \ \}\text{}} \\
| \ K\text{SBranch1}\{s_0, \ n, \ k=k1\} \ \text{⇒} \ {\text{\{} \\
\quad \text{apply}(k1, \ s_0 + sa + n.v) \ \}\text{}} \\
| \ K\text{Term} \ ans \ \text{⇒} \ \{ \text{*ans = sa} \ \}\text{}}
\end{align*}
\]

\[
\text{type} \ kont = \ldots \ 
\begin{align*}
| \ K\text{SBranch0} \ \text{of} \ \{n : \ \text{node*}, \ k : \ kont*\} \\
| \ K\text{SBranch1} \ \text{of} \ \{s_0 : \text{int}, \ n : \ \text{node*}, \ k : \ kont*\}
\end{align*}
\]
Our multicore-parallelization refactoring

Roadmap

Parallel refactoring:
- Fork join
- CPS’d
- Defunct’d CPS

Serial refactoring:
- CPS’d
- Defunct’d CPS

Parallelize w/ fork2join

Optimize

Heartbeat

Reference program
Refactoring for serial traversal

Tail-call elimination of \texttt{apply}

\begin{verbatim}
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 }

}\end{verbatim}
Refactoring for serial traversal

Inline `apply`

```c
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 \texttt{sum}

\begin{verbatim}
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] }
\end{verbatim}
Our multicore-parallelization refactoring

Roadmap

Parallel refactoring

- Fork join
- CPS’d
- Defunct’d

Serial refactoring

- CPS
- CPS’d
- Defunct’d

Parallelize w/ fork2join

Reference program

Optimize

Merge

Heartbeat
Our multicore-parallelization refactoring

Roadmap

Parallel refactoring

Reference program

Fork join

CPS'd

CPS convert

Defunct'd CPS

Optimize

Serial refactoring

CPS

CPS convert

Defunct'd CPS

Heartbeat

Merge
Heartbeat Scheduling

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

![Diagram showing the concept of amortizing overheads](image-url)
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

Latent parallelism

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

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

Promoted

Sequentialized

Heartbeat rate $h = 4$
Merging parallel & serial algorithms
Implementing the heartbeat

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


Merging parallel & serial algorithms
Implementing promotion

We handle the heartbeat here
We promote this

Heartbeat!

try_promote(k)

\( t \rightarrow \)

\( v_1 \)
\( v_2 \)
\( v_3 \)
\( v_4 \)
\( v_5 \)
\( v_6 \)
\( v_7 \)

\( tc \)
\( KPBranch \)
\( \text{sum}(...) \)
Merging parallel & serial algorithms
Implementing promotion

```
try_promote(k)
```

We handle the heartbeat here

We promote this

```
sum(...)  
```

```
tj
```

```
tc  
KPBranch
sum(…)
```
Merging parallel & serial algorithms
Implementing promotion

```
try_promote(k)
```

Heartbeat!

We handle the heartbeat here
We promote this
Merging parallel & serial algorithms
Implementing promotion

```
try_promote()
t
 KPBranch
 KSBranch0

⇒
 tj
 KTerm

⇒
 tc
 KPBranch
 sum(…)

⇒
 t
 KPBranch
 KSBranch0

⇒
 tj
 KTerm

⇒
 tc
 KPBranch
 sum(…)
```

```
⇒ ...
⇒ ...
⇒ ...
```
Merging parallel & serial algorithms
Implementing promotion

```
try_promote()
```

```
t
KPBranch
KSBranch0
try_promote()
tj
KTerm
```

```
t
KPBranch
KSBranch0
try_promote()
tc
KPBranch
sum(...)
tj
KTerm
```

```
tc
KPBranch
sum(...)```

```
t
KPBranch
join(tj)
tj
KTerm
```

```
tc
KPBranch
sum(...)```

```
⇒ ...
⇒ ...
⇒ ...
⇒ ...
⇒ ...
⇒ ...
⇒ ...
```
Merging parallel & serial algorithms
Implementing promotion

\[ t \xrightarrow{\text{KPBranch}} \xrightarrow{\text{KSBranch0}} \xrightarrow{\text{try_promote()}} t_j \xrightarrow{\text{KTerm}} \]

\[ t \xrightarrow{\text{KPBranch}} \xrightarrow{\text{try_promote()}} t_c \xrightarrow{\text{KPBranch}} \xrightarrow{\text{sum(…)}} t_j \xrightarrow{\text{KTerm}} \]

\[ t \xrightarrow{\text{KPBranch}} \xrightarrow{\text{sum(…)}} tc \xrightarrow{\text{KPBranch}} \]

\[ \xrightarrow{\text{join(tj)}} \xrightarrow{\text{KTerm}} \]

\[ \xrightarrow{\text{...}} \]

\[ \xrightarrow{\text{...}} \]
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

<table>
<thead>
<tr>
<th>input</th>
<th>serial (s)</th>
<th>ours</th>
<th>cilk</th>
<th>cilk+granctrl</th>
</tr>
</thead>
<tbody>
<tr>
<td>完美</td>
<td>0.7</td>
<td>28.4x</td>
<td>15.4x</td>
<td>34.5x</td>
</tr>
<tr>
<td>随机</td>
<td>0.8</td>
<td>31.8x</td>
<td>15.3x</td>
<td>33.7x</td>
</tr>
<tr>
<td>链条</td>
<td>2.5</td>
<td>11.5x</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>链</td>
<td>1.2</td>
<td>0.4x</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

- **高并行度**
- **低并行度**

- **完美**是一个完美二叉树，高度为27
- **随机**最初是一个完美二叉树，由一系列路径复制插入构建，针对随机叶
- **链条**是一个小初始树，高度为20，延伸了30个长度为1百万的路径
- **链**是一个长链。

注：可能存在Stack overflow的溢出情况。
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:
\[ E[\text{fork2join}(f_0, f_1)]_k \overset{\text{def}}{=} \{
\begin{align*}
    \text{tj} &= \text{new\_task}(k) \\
    \text{t}_0 &= \text{new\_task}(E[f_0()]_{\lambda()} \cdot \text{join(tj)}) \\
    \text{t}_1 &= \text{new\_task}(E[f_1()]_{\lambda()} \cdot \text{join(tj)}) \\
    \text{fork}(t_0, tj); \text{fork}(t_1, tj)
\end{align*}\]