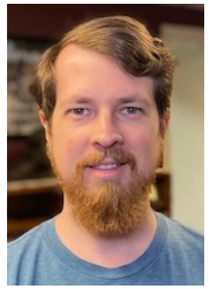
Automatic Parallelism Management

Sam Westrick
New York University

joint work with:

Matthew Fluet
RIT

Colin McDonald
CMU



Mike Rainey
CMU

Umut Acar
CMU

Managing Parallelism Workshop
Simons Institute, UC Berkeley
October 2025

(POPL 2024)

Automatic Parallelism Management

SAM WESTRICK, Carnegie Mellon University, USA
MATTHEW FLUET, Rochester Institute of Technology, USA
MIKE RAINEY, Carnegie Mellon University, USA
UMUT A. ACAR, Carnegie Mellon University, USA

On any modern computer architecture today, parallelism comes with a modest cost, born from the creation and management of threads or tasks. Today, programmers battle this cost by manually optimizing/tuning their codes to minimize the cost of parallelism without harming its benefit, performance. This is a difficult battle: programmers must reason about architectural constant factors hidden behind layers of software abstractions, including thread schedulers and memory managers, and their impact on performance, also at scale. In languages that support higher-order functions, the battle hardens: higher order functions can make it difficult, if not impossible, to reason about the cost and benefits of parallelism.

Motivated by these challenges and the numerous advantages of high-level languages, we believe that it has become essential to manage parallelism automatically so as to minimize its cost and maximize its benefit. This is a challenging problem, even when considered on a case-by-case, application-specific basis. But if a solution were possible, then it could combine the many correctness benefits of high-level languages with performance

MaPLe Compiler (mpl)

github.com/MPLLang/mpl

- **safe, “mostly functional” language with parallelism**

```
val par: (unit -> ‘a) * (unit -> ‘b) -> ‘a * ‘b
val parfor: int * int * (int -> unit) -> unit
...
```

- efficient parallel memory management and scheduling (based on “disentanglement”)
- used by 500+ students at Carnegie Mellon University and New York University in parallel algorithms courses
- competitive performance vs PBBS parallel C/C++ code

Benchmark Suite

`github.com/MPLLang/parallel-ml-bench`

graphs	betweenness centrality
	breadth-first search
	minimum spanning tree
	maximum independent set
	low-diameter decomposition
	triangle counting
geometry	delaunay triangulation
	nearest neighbors
	quickhull
	2D range query
images	seam carving
	raytracing
	tinykaboom
	GIF encode+decode
audio	reverb
	WAV encode+decode
text	tokenization
	deduplication
	grep
	word-count
	longest palindrome
	suffix array
numeric	dense+sparse matrix mult
	integration
	linear regression

includes...

- over 30 state-of-the-art parallel algorithms and applications
- ported from C/C++ ParlayLib, PBBS, GBBS, Ligra, PAM, ...

porting methodology

- preserve high-level algorithms, data structures
  - e.g., C/C++ array of structs $\iff$ MPL sequence of tuples
  - make use of common primitives: scan, reduce, filter, etc.
  - ensure same memory representation and access pattern (as much as possible)

rough summary

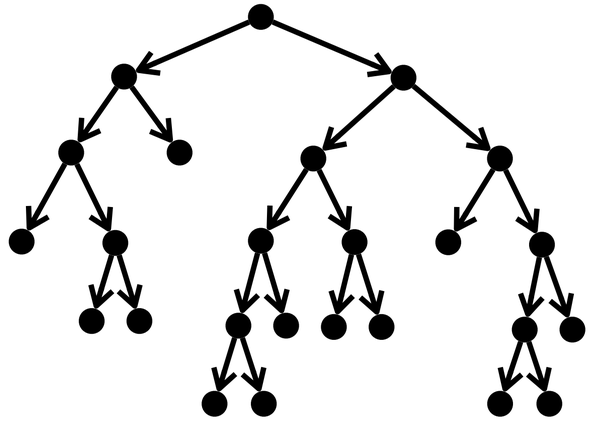
- on 72 cores: up to 40x speedup over sequential C/C++
- across all core counts: generally within 1-2x of parallel C/C++
- active research on closing the gap:
  more control over memory layouts, loop unrolling, etc.

The Granularity Control Problem

```
(* do func(...) in parallel for every
 * leaf in the tree
 *)
fun traverse(tree, func) =
  case tree of
  | Leaf{x} => func(x)
  | Node{l, r, ...} =>
     (par(fn () => traverse(l, func),
          fn () => traverse(r, func)); ())
```

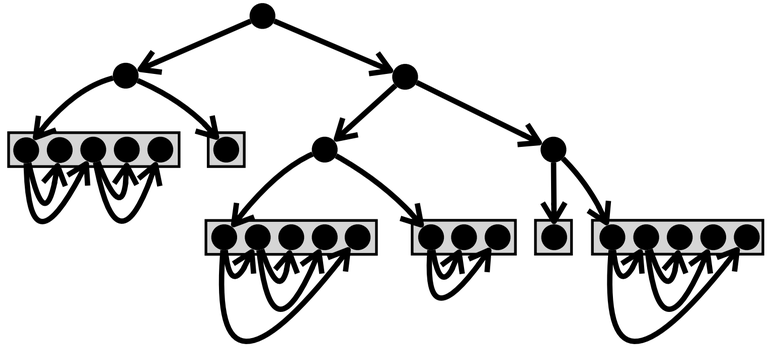
```
fun traverse(tree, func) =
  if size(tree) <= GRAIN_SIZE then
    sequential_traverse(tree, func)
  else
  case tree of
  ...
```

slow



10x (or more) performance gap in practice

fast



The Granularity Control Problem

```
(* do func(...) in parallel for every
 * leaf in the tree
 *)
fun traverse(tree, func) =
  if size(tree) <= GRAIN_SIZE then
    sequential_traverse(func, tree)
  else
  case tree of
  | Leaf{x} => func(x)
  | Node{l, r, ...} =>
     (par(fn () => traverse(l, func),
          fn () => traverse(r, func)); ())
```

what grain size should you pick?

```
traverse(tiny_tree, expensive_func)
```

```
traverse(huge_tree, cheap_func)
```

```
traverse(t, fn x =>
  let n = foo(x)
  parfor(0, n, fn i => ...)
)
```

The Granularity Control Problem

- how much parallelism should I expose? (how “fine-grained” should my tasks be?)
- **can this be automated?**

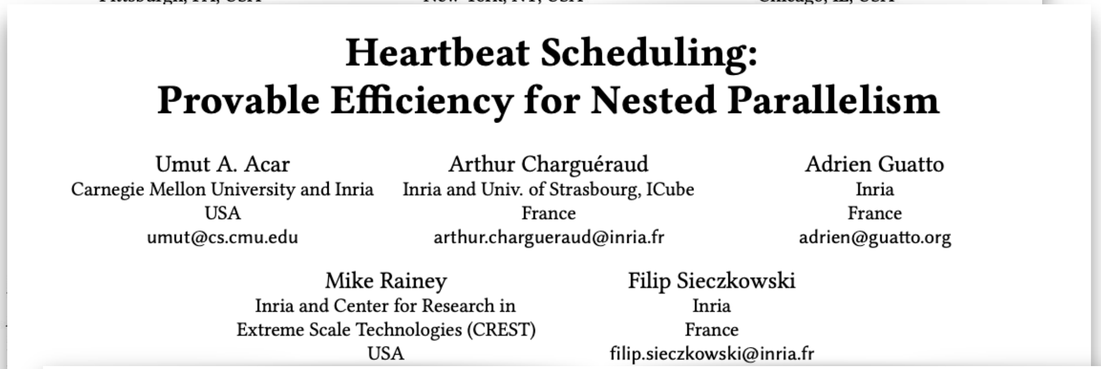
- lots of existing work *(lazy scheduling, lazy binary splitting / lazy tree splitting, heartbeat scheduling, oracle-guided control, static cut-offs, cost annotations, profiling techniques...)*
- **we want...**
  - fully general solution
  - provably efficient
  - implementable and effective in practice

Compiling Loop-Based Nested Parallelism for Irregular Workloads

Yian Su, Northwestern University, Evanston, IL, USA
Mike Rainey, Carnegie Mellon University, Pittsburgh, PA, USA
Nick Wanninger, Northwestern University, Evanston, IL, USA
Nadharm Dhiantravan, Northwestern University, Evanston, IL, USA
Jasper Liang, Northwestern University, Evanston, IL, USA
Umut A. Acar, Carnegie Mellon University, Pittsburgh, PA, USA
Peter Dinda, Northwestern University, Evanston, IL, USA
Simone Campanoni, Northwestern University, Evanston, IL, USA

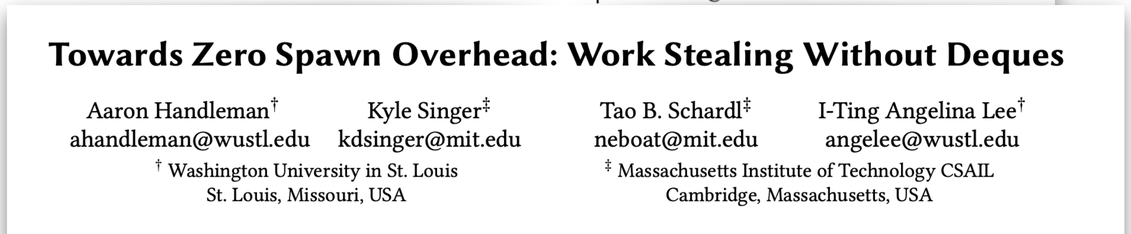
Task Parallel Assembly Language for Uncompromising Parallelism

Mike Rainey, Carnegie Mellon University, Pittsburgh, PA, USA
Ryan R. Newton, Facebook, New York, NY, USA
Kyle Hale, Illinois Institute of Technology, Chicago, IL, USA



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
Filip Sieczkowski, Inria, France, filip.sieczkowski@inria.fr



Towards Zero Spawn Overhead: Work Stealing Without Deques

Aaron Handleman[†], ahandleman@wustl.edu
Kyle Singer[‡], kdsinger@mit.edu
Tao B. Schardl[‡], neboat@mit.edu
I-Ting Angelina Lee[†], angelee@wustl.edu
[†] Washington University in St. Louis, St. Louis, Missouri, USA
[‡] Massachusetts Institute of Technology CSAIL, Cambridge, Massachusetts, USA

Automatic Parallelism Management

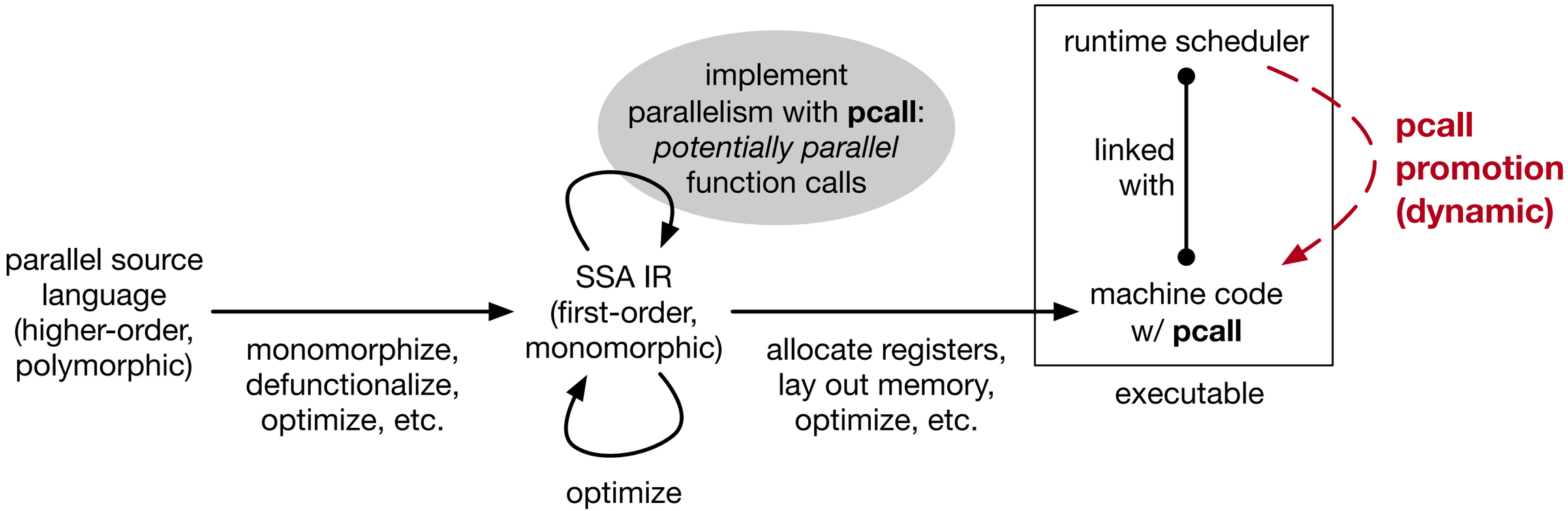
language primitive:
`par(f,g)` executes `f()` and `g()` in parallel

compiler and run-time system cooperate to guarantee efficiency

in the compiler...
nearly zero cost implementation of **`par`**
- by default, **`par`** executes sequentially
- can be ***promoted*** to create parallel tasks
  - **how?** dynamic replacement of call-stack frames

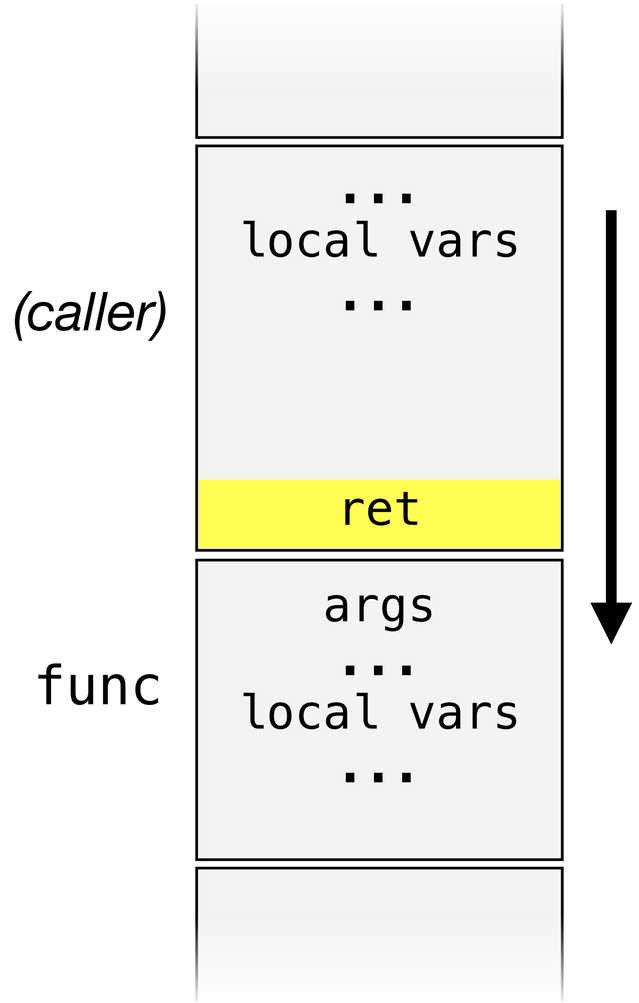
in the run-time system...
provably efficient scheduling of promotions

Compilation

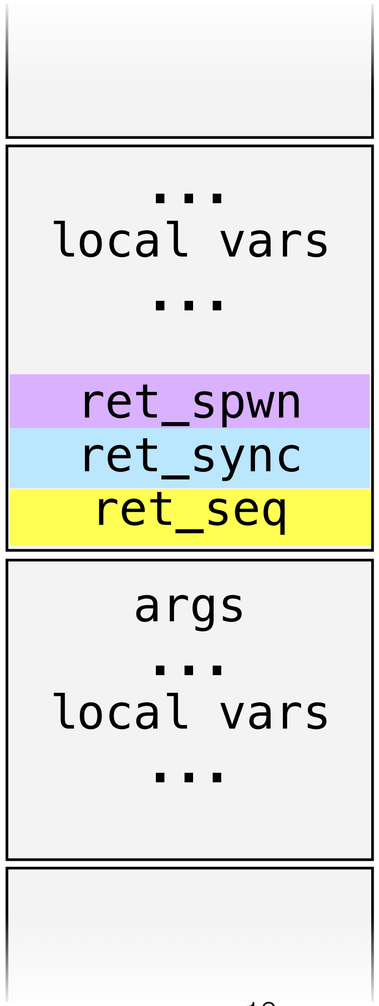


`PCall` Calling Convention

`Call``(func, args, ret)`



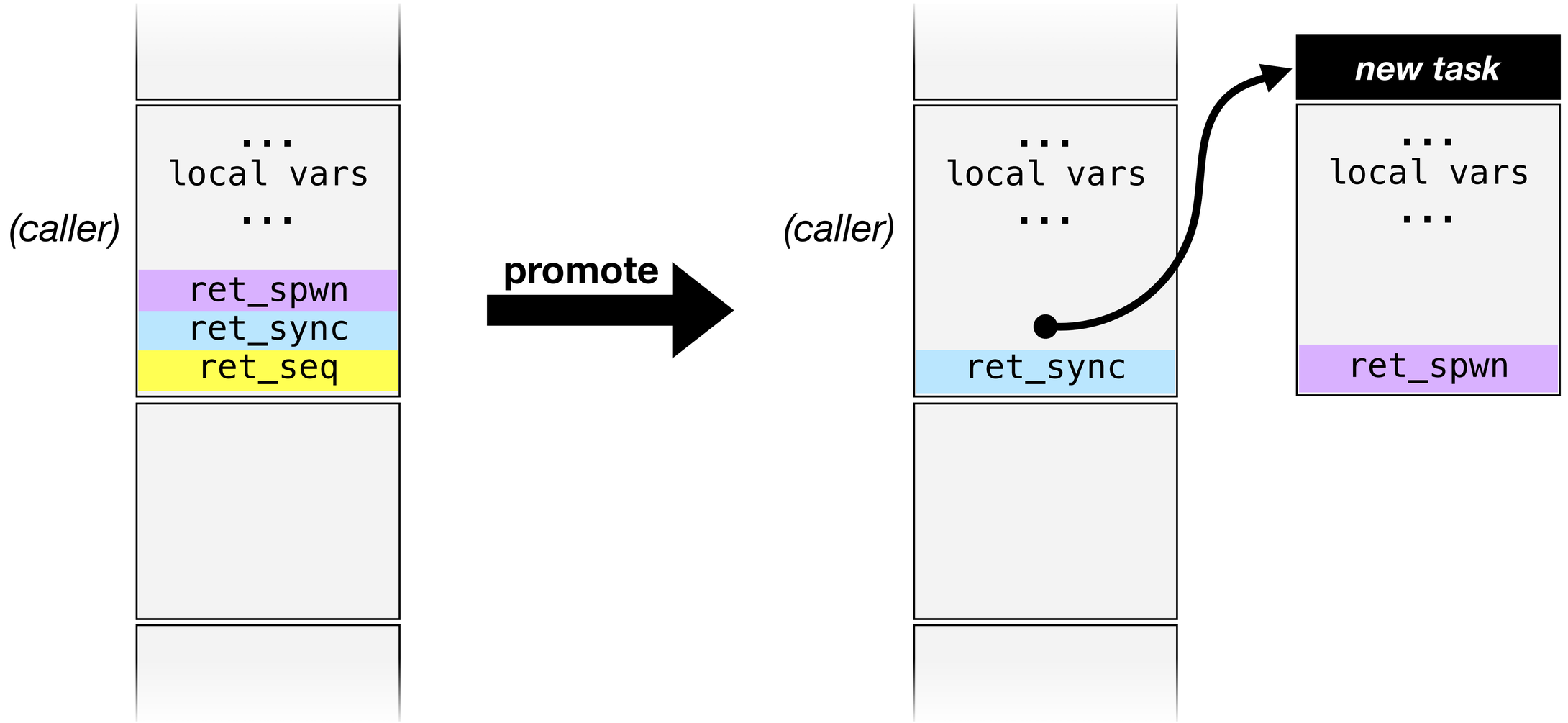
`PCall``(func, args, ret_seq, ret_sync, ret_spwn)`



IF NEVER PROMOTED...

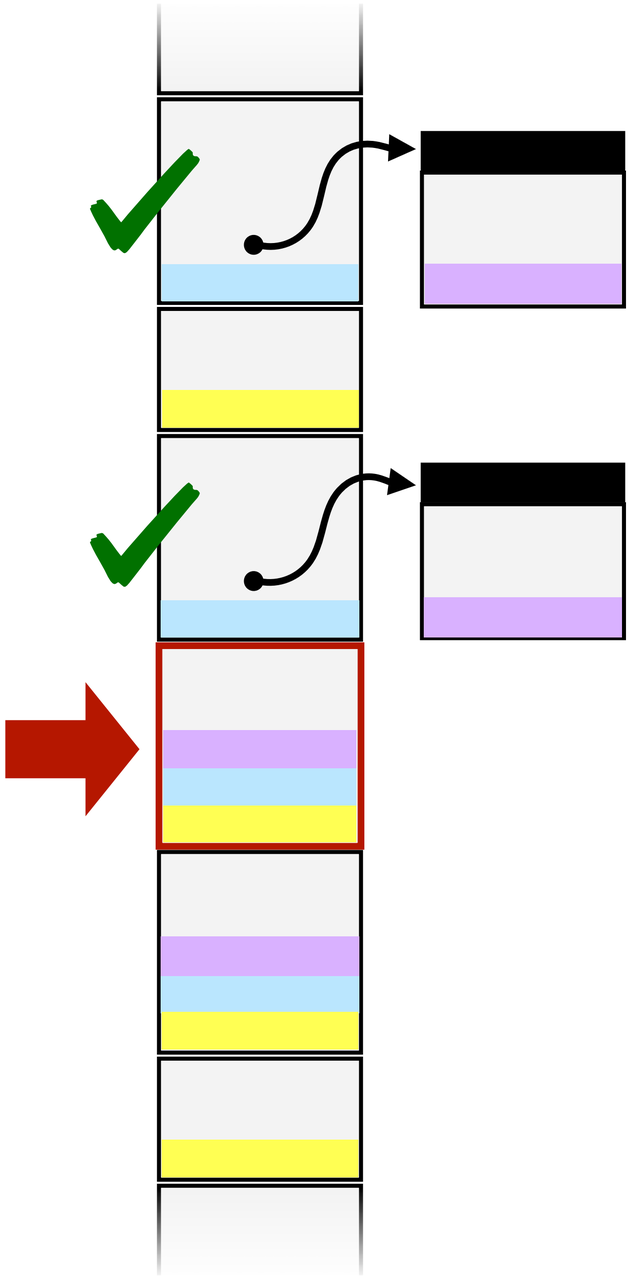
- behaves the same as normal **`Call`**
- caller resumes at `ret_seq`
- `ret_sync` and `ret_spwn` are discarded

`PCall` Promotion



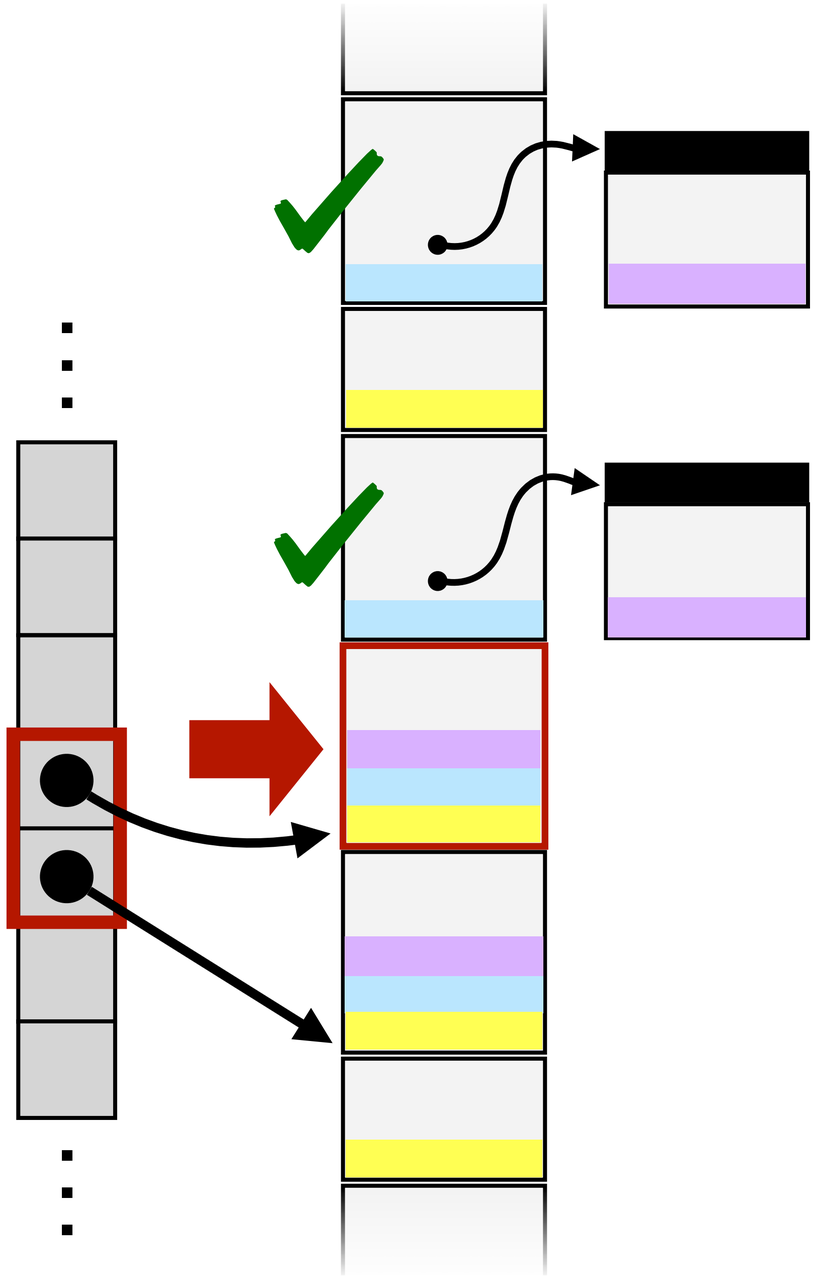
Scheduling Promotions

- each promotion exposes parallelism but incurs a cost
- idea: amortize cost of promotion against *“true”* work
- **algorithm**
  - every N microseconds, each thread receives C tokens
  - any thread may spend one token to promote the ***outermost*** (oldest) outstanding `PCall`
    (in the thread’s own call-stack)



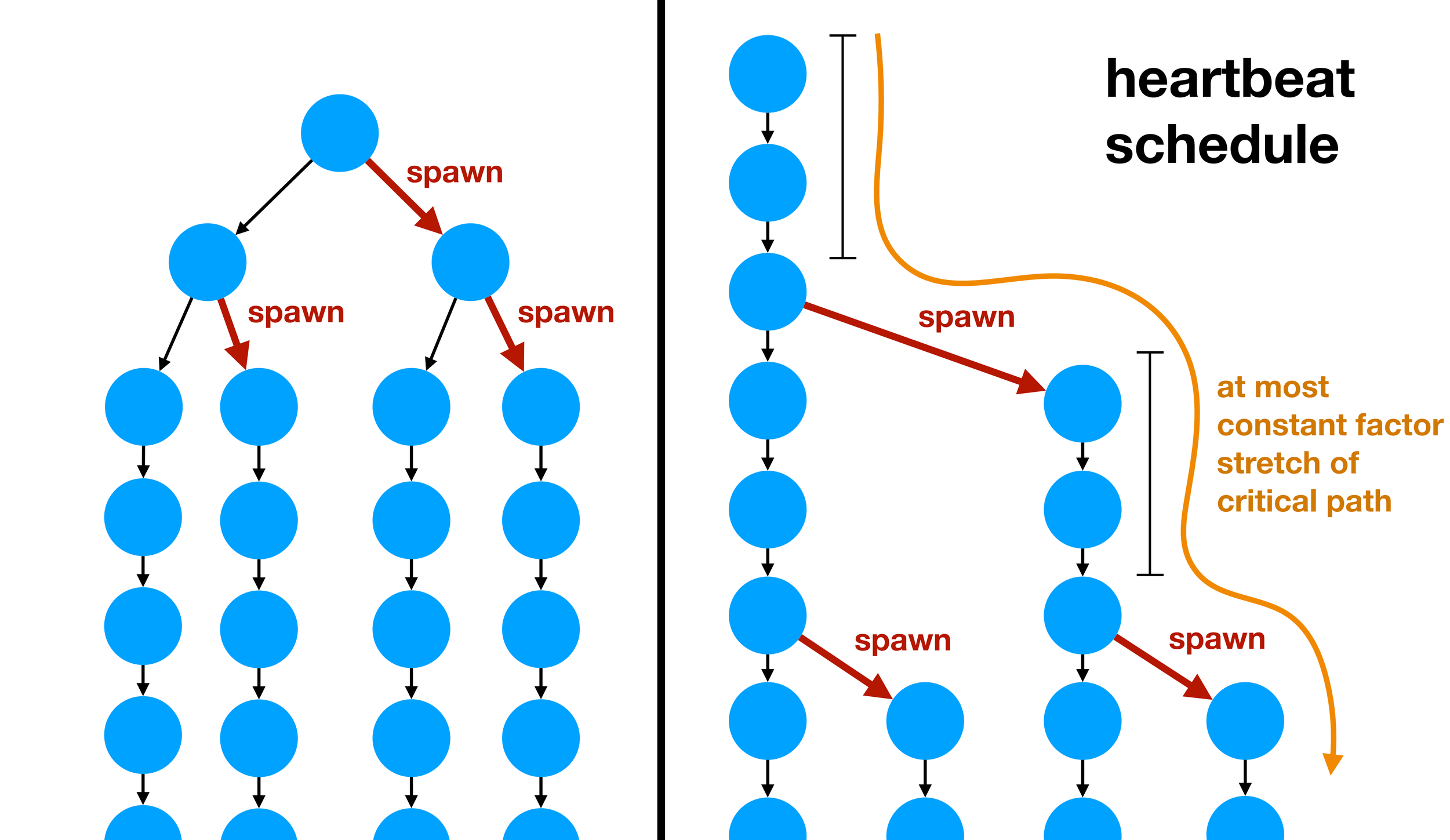
Promotion Stacks

- for **outermost promotion**:
  need O(1) access to oldest promotable frame
- dynamically maintain promotion stacks during execution:
  - at **`PCall`**, push onto bottom of promotion stack:
    ```
    promo_stack[bot] = <frame_pointer>;
    bot++;
    ```
  - at `ret_seq`, unconditional pop:
    ```
    bot--;
    ```
  - at `ret_sync`, unconditional pop and reset top:
    ```
    bot--; top = bot;
    ```
  - at **promotion**:
    ```
    promote(promo_stack[top])
    top++;
    ```

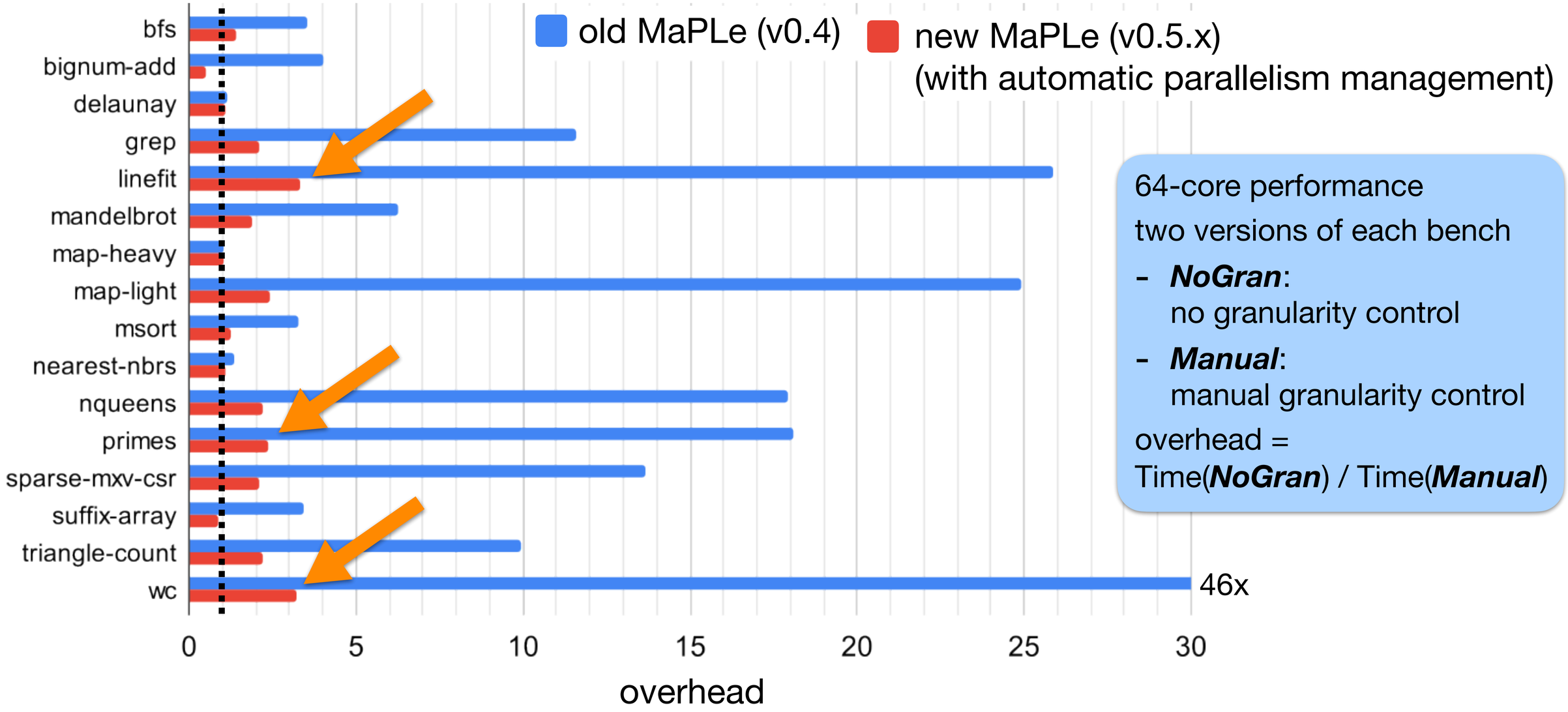


Work and Span Efficiency

- **our algorithm**
  - every N microseconds, each thread receives C tokens
  - any thread may spend one token to promote the ***outermost*** (oldest) frame
- token accounting overcomes rapid heartbeat requirement: **now can use much larger N**
- work analysis: straightforward
  - sequential work W increases to at most (1 + τC/N) W
- span analysis:
  - naive reduction to Heartbeat Scheduling yields loose bound
  - from Heartbeat Scheduling (C = 1, can never “save” a token):
    - idealized span S increases to at most (1 + N/τ) S
  - so, with tokens:
    - idealized span S increases to at most O(S)


spawn
spawn
spawn
heartbeat
schedule
spawn
at most
constant factor
stretch of
critical path
spawn
spawn

Parallelism Overhead (lower is better)



64-core performance

two versions of each bench

- ***NoGran***:
  no granularity control
- ***Manual***:
  manual granularity control

overhead =
Time(***NoGran***) / Time(***Manual***)

```
(* combine {f(i): lo <= i < hi}
 * w.r.t. associative g(-,-) *)
fun reduce(g, z, lo, hi, f) =
  if lo >= hi then z else
  if lo+1 = hi then f(lo) else
  let val mid = lo + (hi-lo) div 2
      val (a, b) = par(fn () => reduce(lo, mid, f),
                       fn () => reduce(mid, hi, f))
  in g(a, b)
  end
```

n-1 midpoint calculations

n-1 PCalls + n-1 regular calls:

2(n-1) stack frame push/pop in total

```
(* combine {f(i): lo <= i < hi}
 * w.r.t. associative g(-,-) *)
fun reduce(g, z, lo, hi, f) =
  if lo >= hi then z else
  if lo+1 = hi then f(lo) else
  let val mid = lo + (hi-lo) div 2
      val (a, b) = par(fn () => reduce(lo, mid, f),
                       fn () => reduce(mid, hi, f))
  in g(a, b)
  end
```

“algorithmic” gap between sequential and parallel alternatives

```
(* sequential accumulating loop *)
fun reduce(g, a, lo, hi, f) =
  if lo >= hi then a else
  reduce(g, g(a,f(lo)), lo+1, hi, f)
```


Zero-Overhead Parallel Scans for Multi-Core CPUs

Ivo Gabe de Wolff
Utrecht University
Netherlands

David P. van Balen
Utrecht University
Netherlands

Gabriele K. Keller
Utrecht University
Netherlands

Trevor L. McDonell
Utrecht University
Netherlands

Abstract

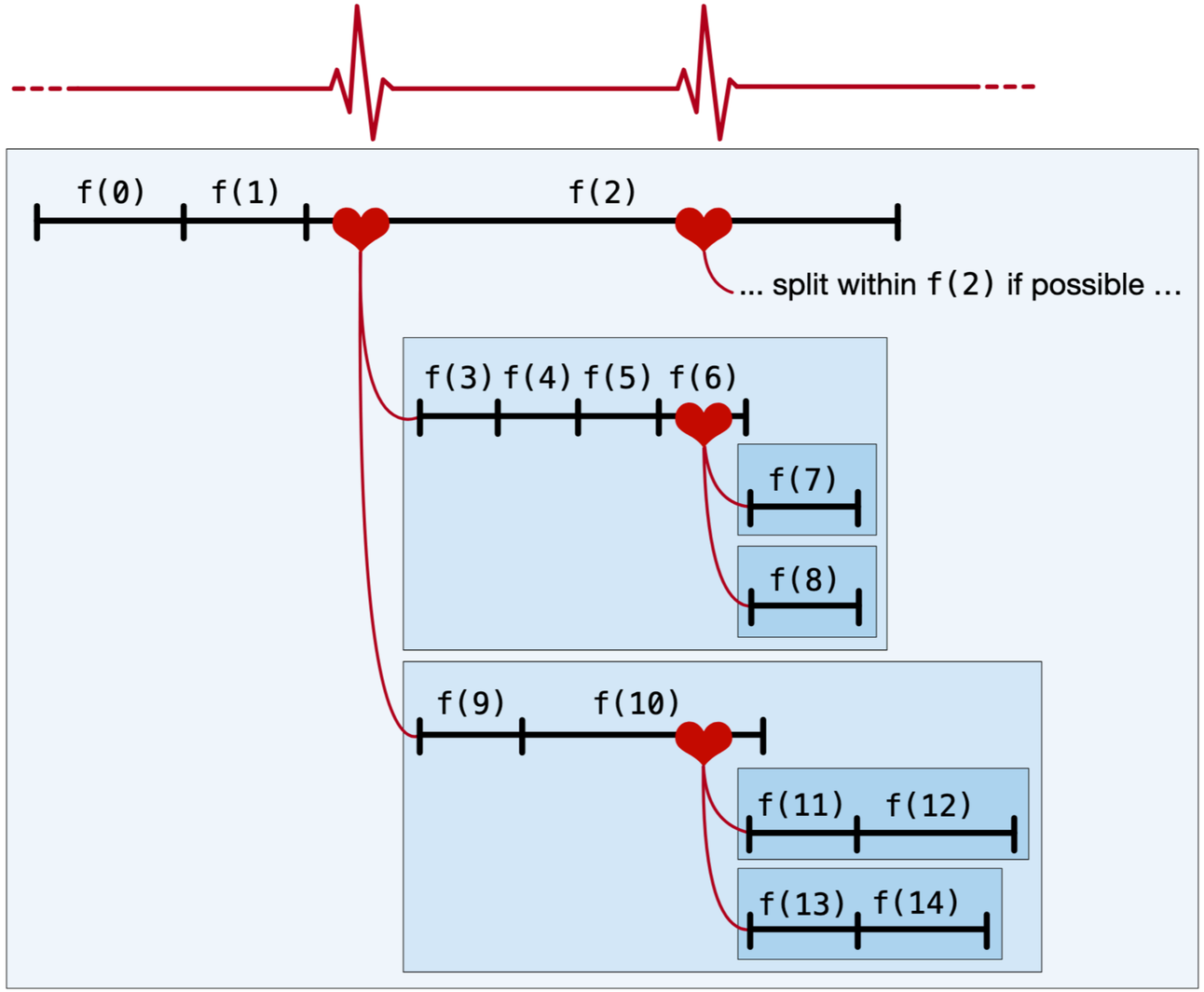
We present three novel parallel scan algorithms for multi-core CPUs which do not need to fix the number of available cores at the start, and have zero overhead compared to sequential scans when executed on a single core. These two properties are in contrast with most existing parallel scan algorithms, which are asymptotically optimal, but have a constant factor overhead compared to sequential scans when executed on a single core. We achieve these properties by adapting the classic three-phase scan algorithms. The resulting algorithms also exhibit better performance than the original ones on multiple cores. Furthermore, we adapt the chained scan with decoupled look-back algorithm to also have these two properties. While this algorithm was originally designed for GPUs, we show it is also suitable for multi-core CPUs, outperforming the classic three-phase scans in our benchmarks, by better using the caches of the processor at the cost of more synchronisation. In general our *adaptive chained scan* is the fastest parallel scan, but in

Given an array, a scan computes for each element the combined value of all prior elements (including or excluding its own value, for respectively an inclusive or exclusive scan). The scan uses a binary operator ⊕ to combine two elements, and can be implemented sequentially as follows:

function SCAN_SEQ(T* *input*, T* *output*, *size*, T *initial*)
  accum ← *initial*
  for *i* **in** 0 . . . *size* **do**
    accum ← *accum* ⊕ *input*[*i*]
    output[*i*] ← *accum*
  return *accum*

If the operator ⊕ is associative, the scan can be executed in parallel. In this paper, we introduce scan algorithms for thread-level parallelism on CPUs which exhibit faster performance than the state of the art algorithms. We address the cost of performing scans in parallel on multi-core CPUs. Most existing parallel scan algorithms are asymptotically optimal. However, they typically have a constant factor overhead over sequential scans, as they traverse the data multiple

Auto Par Management of Loops+Reductions (WIP)



Parallelism Overhead (lower is better)