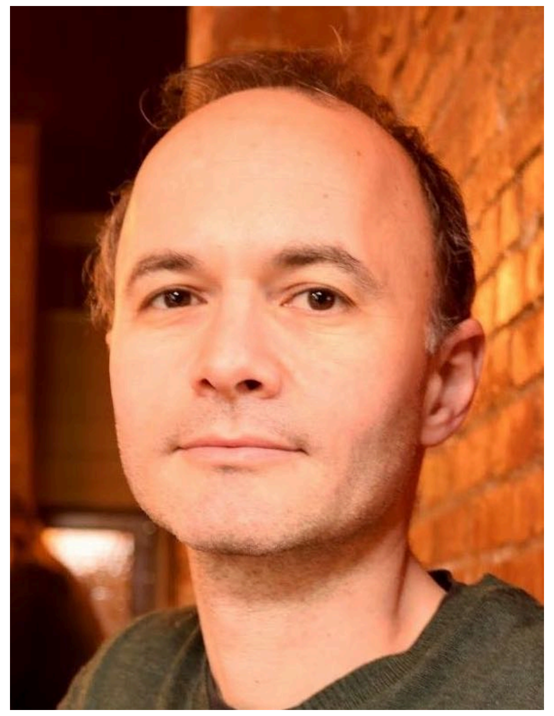
Automatic Parallelism Management

Sam Westrick
Carnegie Mellon University

POPL 2024
London, UK

joint work with:



Matthew Fluet
Rochester Institute of Technology

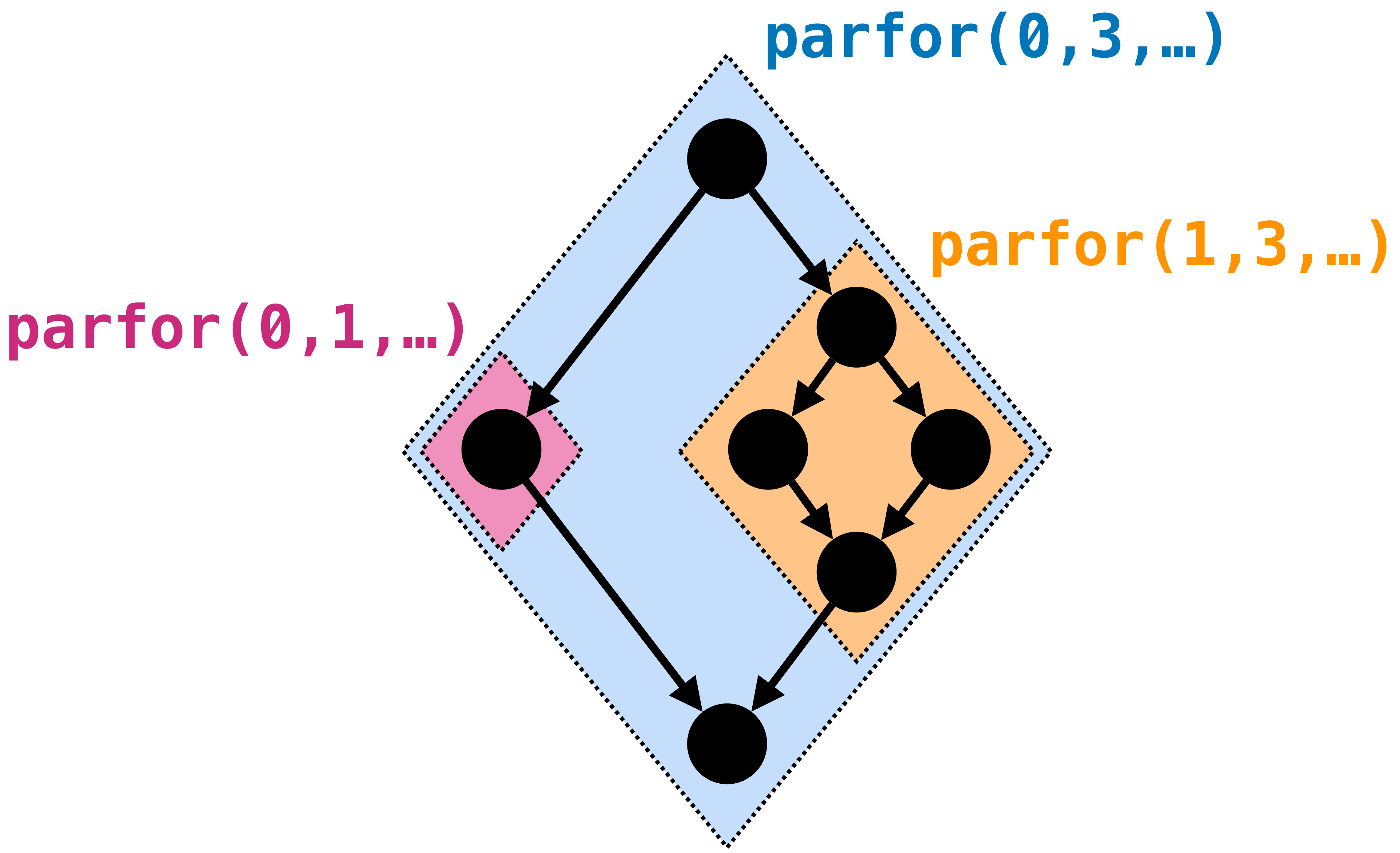
Mike Rainey
Carnegie Mellon University

Umut Acar
Carnegie Mellon University

Fine-Grained Task Parallelism

- **par**: `(unit -> 'a) * (unit -> 'b) -> 'a * 'b`
- scheduler guarantees efficient execution on any number of processors

```
(* do body(i) for each i: lo <= i < hi *)
fun parfor(lo, hi, body) =
  if lo >= hi then () else
  if lo+1 = hi then body(lo) else
  let val mid = lo + (hi-lo) div 2
  in par(fn () => parfor(lo, mid, body),
         fn () => parfor(mid, hi, body));
     ()
  end
```



Parallelism Isn't Free

```
(* do body(i) for each i: lo <= i < hi *)
fun parfor(lo, hi, body) =
  if lo >= hi then () else
  if lo+1 = hi then body(lo) else
  let val mid = lo + (hi-lo) div 2
  in par(fn () => parfor(lo, mid, body),
         fn () => parfor(mid, hi, body));
     ()
  end
```

```
fun parfor(lo, hi, body) =
  if hi-lo <= GRAIN_SIZE then
    sequential_for_loop(lo, hi, body)
  let val mid = lo + (hi-lo) div 2
  in par ...
  end
```

up to 50x performance gap in practice

The Granularity Control Problem

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

what grain size should you pick?

```
(* do body(i) for each i: lo <= i < hi *)
fun parfor(lo, hi, body) =
  if hi-lo <= GRAIN_SIZE then
    sequential_for_loop(lo, hi, body)
  let val mid = lo + (hi-lo) div 2
  in par(fn () => parfor(lo, mid, body),
         fn () => parfor(mid, hi, body));
     ()
  end
```

```
parfor(0, 1000, expensive_func)
```

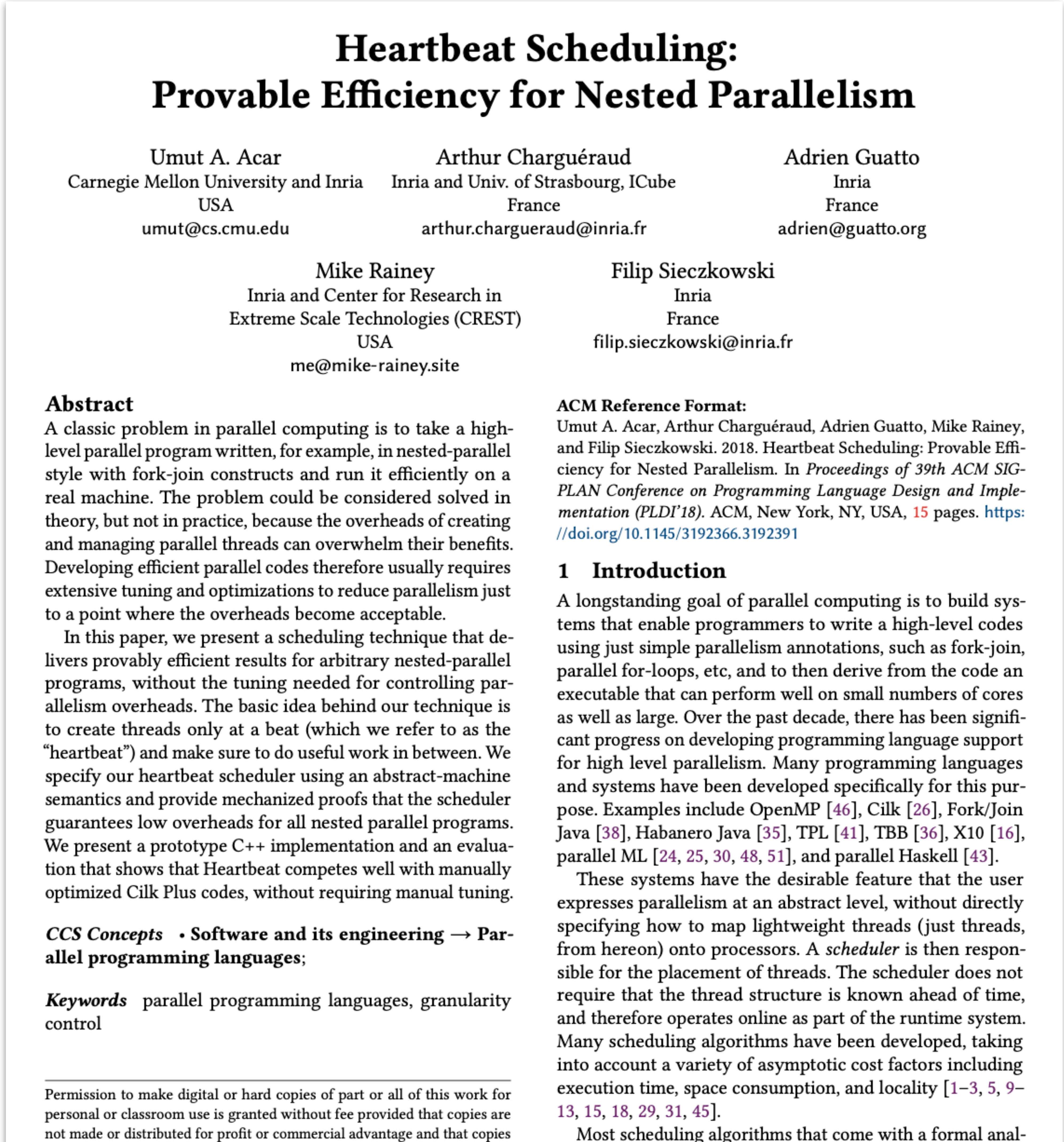
```
parfor(0, 100000000, cheap_func)
```

```
parfor(0, N, fn i =>
  let M = foo(i)
  parfor(0, M, fn j => ...)
)
```

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



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 extensive tuning and optimizations to reduce parallelism just to a point where the overheads become acceptable.

In this paper, we present a scheduling technique that delivers provably efficient results for arbitrary nested-parallel programs, without the tuning needed for controlling parallelism overheads. The basic idea behind our technique is to create threads only at a beat (which we refer to as the "heartbeat") and make sure to do useful work in between. We specify our heartbeat scheduler using an abstract-machine semantics and provide mechanized proofs that the scheduler 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 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, 15 pages. https://doi.org/10.1145/3192366.3192391

1 Introduction

A longstanding goal of parallel computing is to build systems that enable programmers to write a high-level codes using just simple parallelism annotations, such as fork-join, parallel for-loops, etc, and to then derive from the code an executable that can perform well on small numbers of cores as well as large. Over the past decade, there has been significant progress on developing programming language support for high level parallelism. Many programming languages and systems have been developed specifically for this purpose. Examples include OpenMP [46], Cilk [26], Fork/Join 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 anal-

Automatic Parallelism Management

Our Approach

static

programmer uses **`par`** liberally to express opportunities for parallelism

- **`PCall`**: new compilation technique for **`par`** with ***nearly zero cost***
- **`PCall`** behaves sequentially by default (avoids task creation by default)
- each **`PCall`** can be dynamically ***promoted*** into an actual parallel task

dynamic

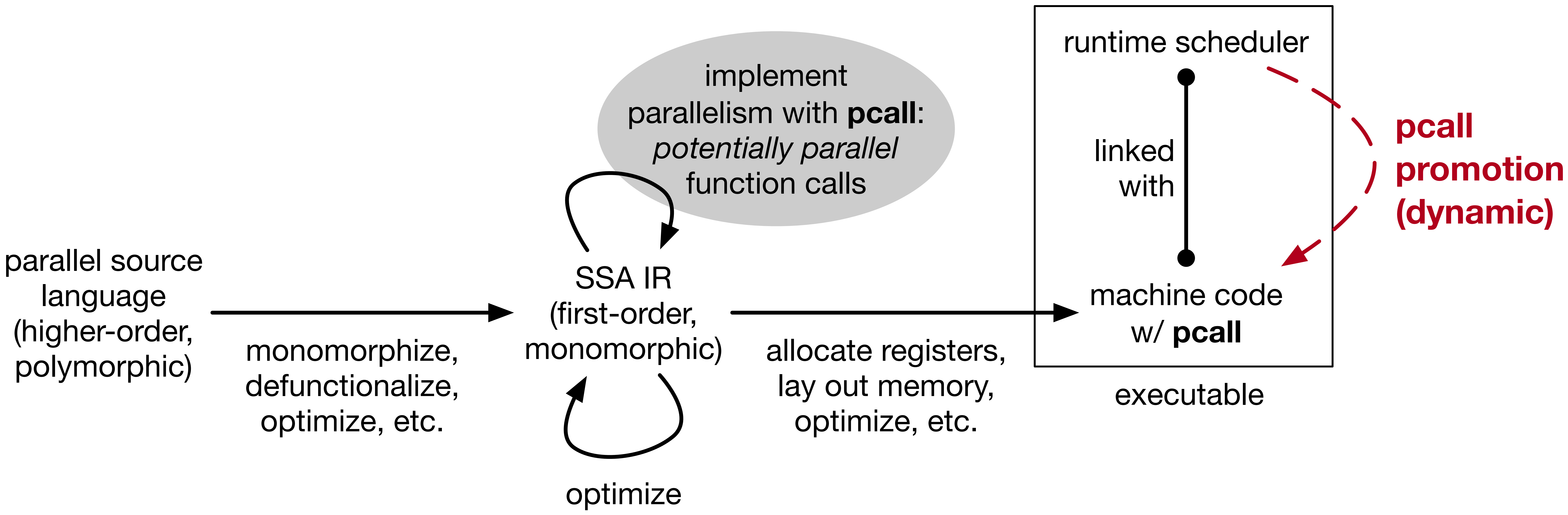
provably efficient scheduling of ***promotions***

- each promotion releases parallelism but also incurs a cost
- our algorithm guarantees...
  - *work-efficiency* (cost of all promotions is amortized)
  - *span-efficiency* (theoretical parallelism is preserved)

full implementation in MaPLe

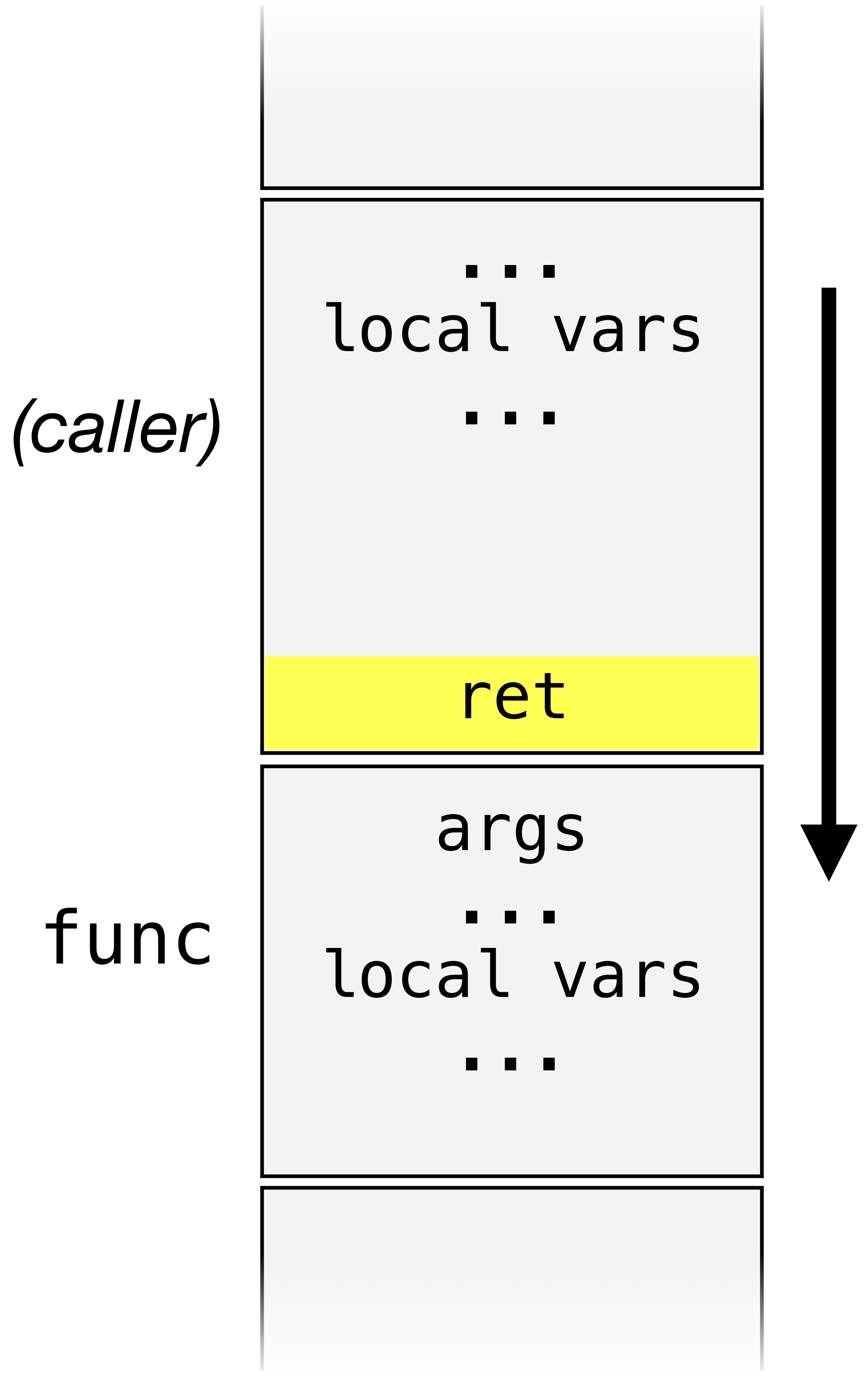
`github.com/MPLLang/mpl`

Compilation

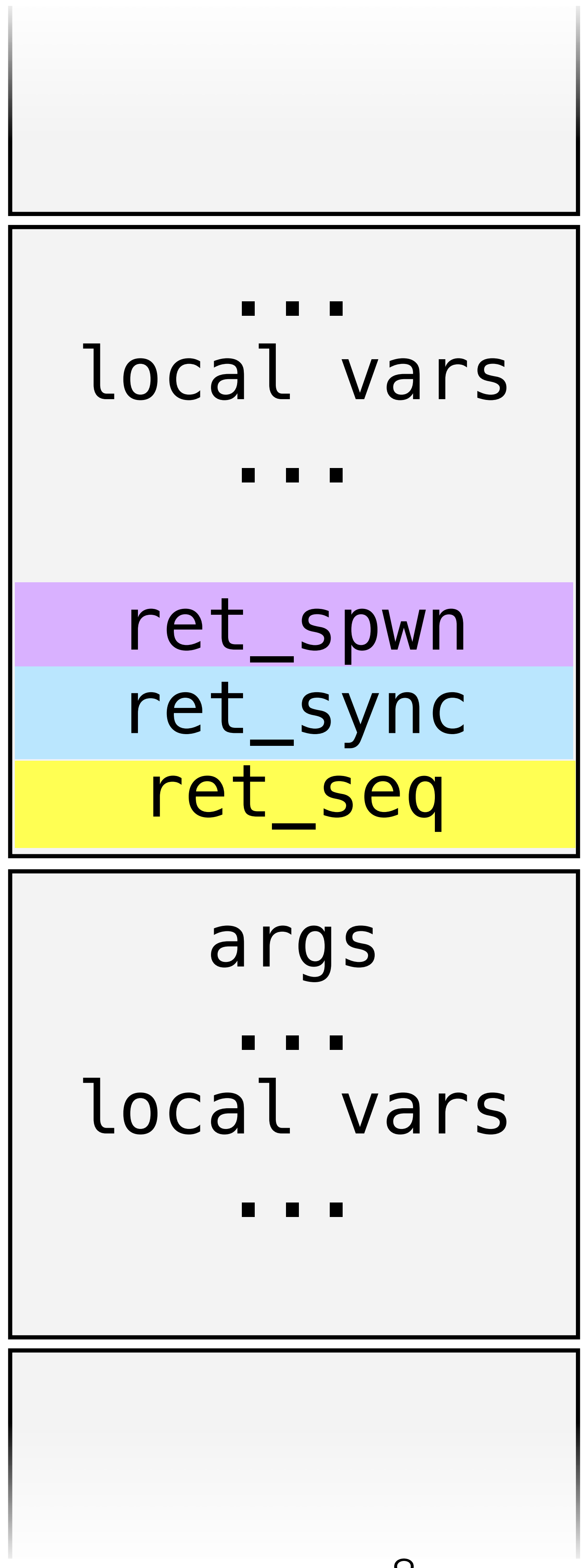


PCall Calling Convention

Call(func, args, ret)



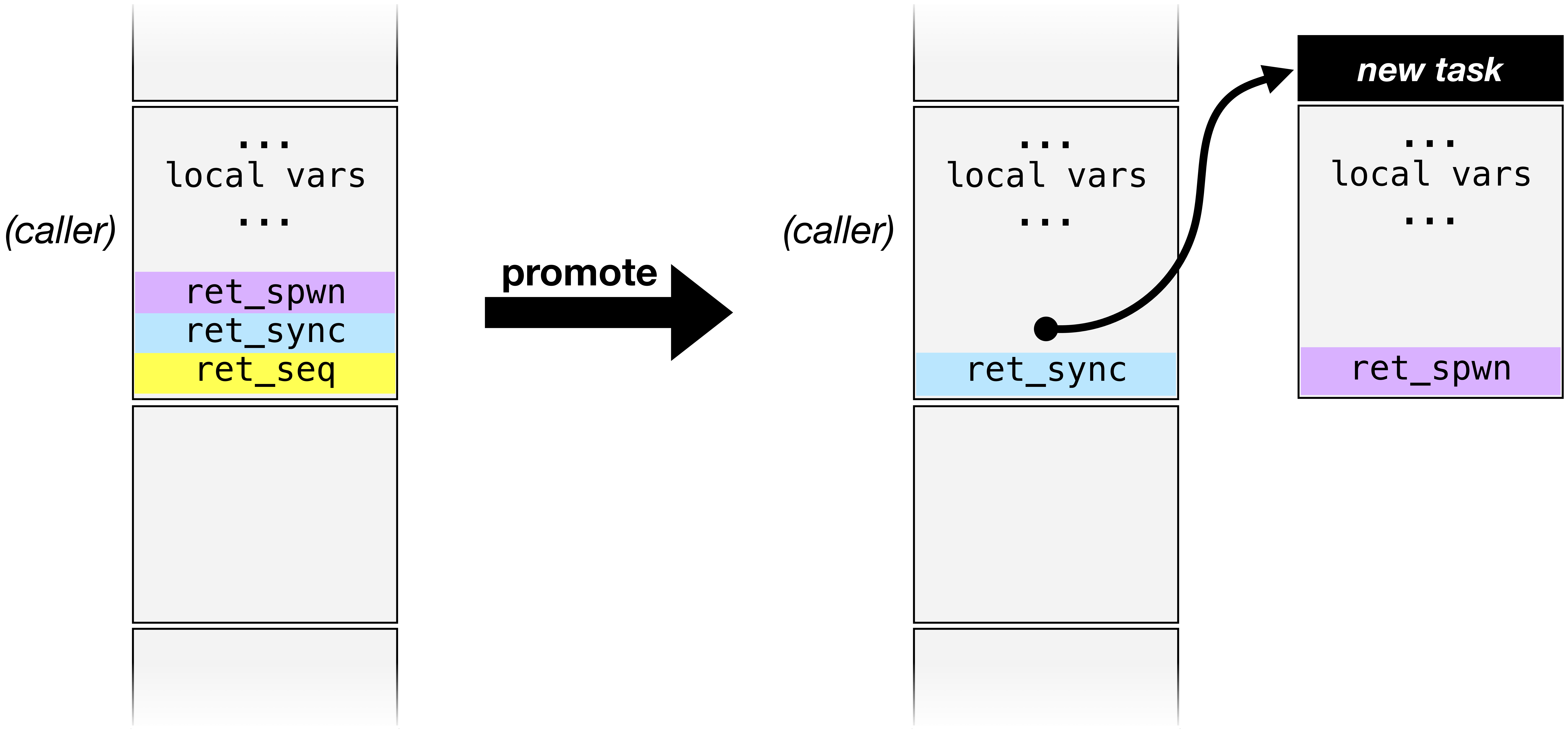
PCall(func, args, ret_seq, ret_sync, ret_spwn)



IF NEVER PROMOTED...

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

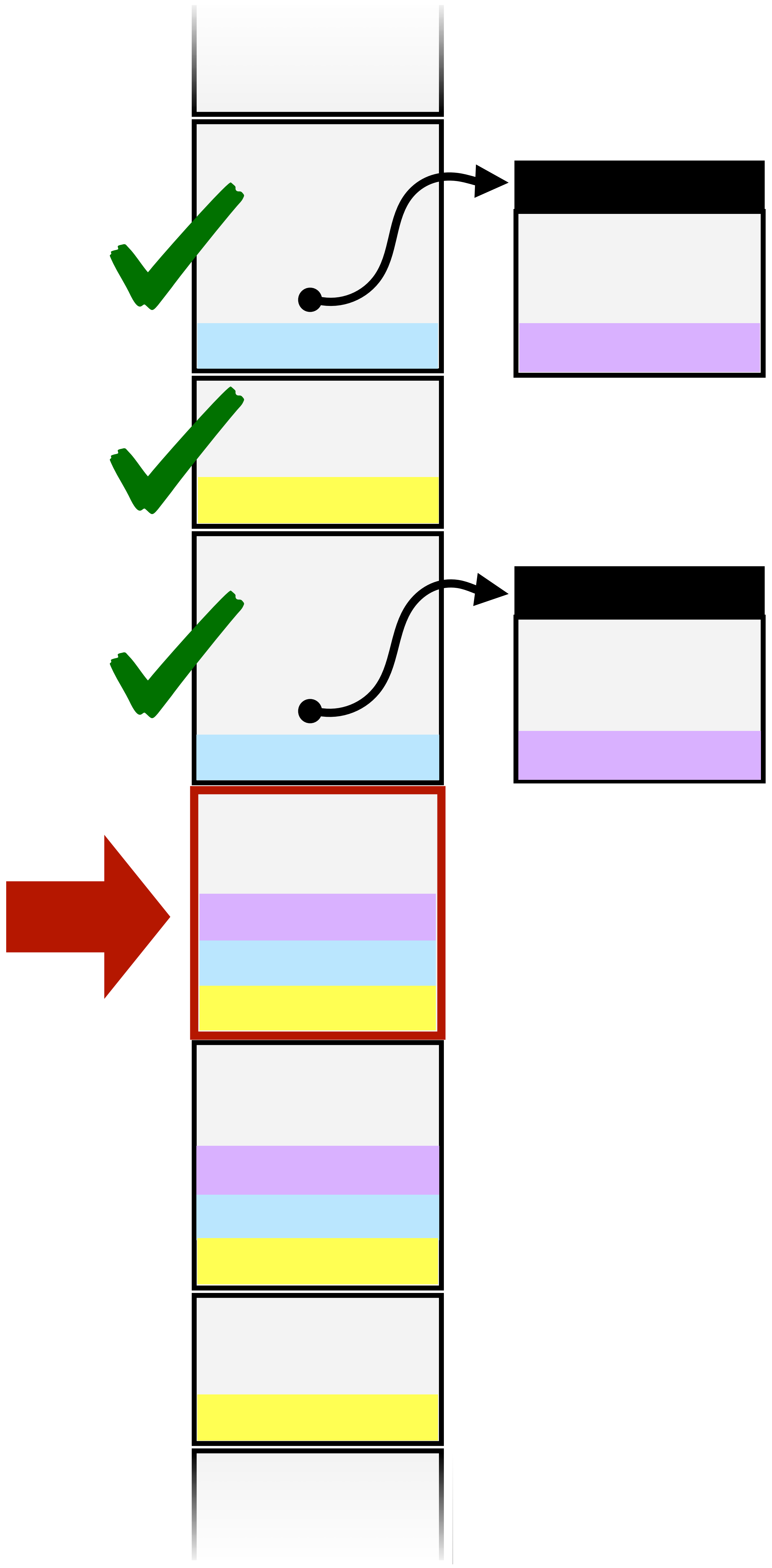
`PCall` Promotion



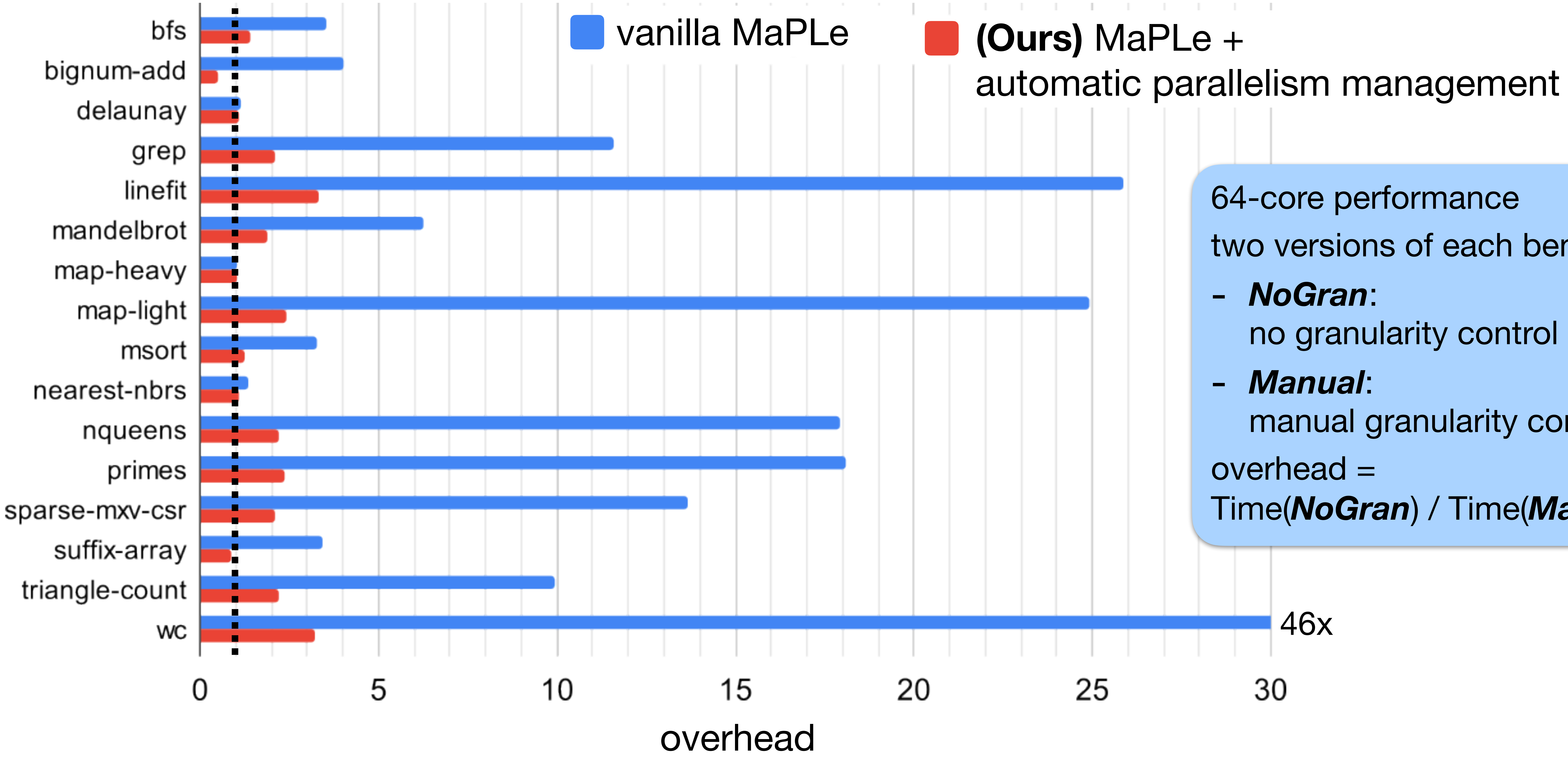
Scheduling Promotions

- 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)

theorems:
work-efficiency and span-efficiency



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***)

Automatic Parallelism Management
Summary

- *nearly zero cost* compilation technique for **`par`** (**`PCall`** + promotions)
- provable and practical efficiency, even without granularity control

see the paper for...

- SSA formalism, **`PCall`** semantics
- theorems: work- and span-efficiency
- description of changes to MLton/MaPLe compiler and run-time system
- in-depth empirical evaluation

`github.com/MPLLang/mpl`

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 by managing parallelism without the programmer effort needed to ensure performance. This paper proposes techniques for such automatic management of parallelism by combining static (compilation) and run-time techniques. Specifically, we consider the Parallel ML language with task parallelism, and describe a compiler pipeline that embeds "potential parallelism" directly into the call-stack and avoids the cost of task creation by default. We then pair this compilation pipeline with a run-time system that dynamically converts potential parallelism into actual parallel tasks. Together, the compiler and run-time system guarantee that the cost of parallelism remains low without losing its benefit. We prove that our techniques have no asymptotic impact on the work and span of parallel programs and thus preserve their asymptotic properties. We implement the proposed techniques by extending the MPL compiler for Parallel ML and show that it can eliminate the burden of manual optimization while delivering good practical performance.

CCS Concepts: • **Software and its engineering** → **Parallel programming languages**; *Functional languages*; *Procedures, functions and subroutines*; **Compilers**.

Additional Key Words and Phrases: parallel programming languages, granularity control, compilers