Spork: An Intermediate Representation for Parallelism Management

COLIN MCDONALD, Carnegie Mellon University, USA
SAM WESTRICK, New York University, USA
MATTHEW FLUET, Rochester Institute of Technology, USA
UMUT A. ACAR, Carnegie Mellon University, USA

High-level constructs for parallelism such as the fork-join model and parallel loops can greatly aid writing parallel code by allowing programmers to express parallelism at a high level of abstraction without worrying about details like thread creation, scheduling, and synchronization. The problem is that parallelism is not free: parallel code incurs overheads to manage such tasks. As a result, high-level parallel code does not typically deliver the performance expected of a parallel program, requiring programmers to optimize their code manually to control the overheads of managing threads. Such optimizations demand a great degree of effort and experience, requiring programmers to reason about architectural constant factors hidden behind layers of software, and result in over-engineered code that is difficult to reason about. Recent advances in parallelism management show that it is feasible to manage parallelism fully automatically while guaranteeing reasonably high performance. Prior work on parallelism management, however, assumes a binary fork-join model of parallelism and does not provide direct support for the dominant form of parallelism: parallel loops.

In this paper, we propose techniques for parallelism management of parallel loops by bringing together language design, compilation, and implementation. Our approach starts with an SSA-based intermediate representation (IR) that includes a pair of primitives for managing parallelism. These primitives, called *spork* and *spoin*, enable code that executes sequentially by default but can "go parallel" when the runtime environment favors concurrent execution. We formalize the semantics of these primitives and establish key soundness theorems using the Lean theorem prover. We present techniques for encoding both high-level parallel loops and fork-join parallel code in the IR using the spork and spoin primitives. Then, we show how to couple these primitives with heartbeat scheduling to determine when they should go parallel and when they should stay sequential to guarantee efficiency. We implement our techniques by extending the MPL compiler for Parallel ML and conduct an experimental evaluation. The experiments show that our approach performs well in practice, delivering an average of <70% overhead on a single core vs. sequential and an average of 28x speedup on 80 cores, while requiring no human effort for performance optimization of parallelism overheads.

CCS Concepts: • **Software and its engineering → Compilers**; **Parallel programming languages**; *Formal language definitions.*

Additional Key Words and Phrases: parallel loops, parallelism management, granularity control

1 Introduction

In principle, it is not difficult to write parallel programs using high-level parallel constructs like parallel loops. But writing ***performant*** parallel programs, which compete with sequential code on small numbers of cores while also scaling to larger numbers, remains a major challenge. For example, just as we could implement a simple sequential matrix multiplication with three nested loops, we could implement a parallel matrix multiplication with three nested "parallel for" loops. Ideally, the parallel implementation would perform no worse than the sequential implementation and would offer significant speedups. In reality the parallel implementation will be significantly slower than its sequential counterpart (as much as an order of magnitude slower), and will struggle to catch up, even as we use more cores.

Authors' Contact Information: Colin McDonald, colinmcd@cs.cmu.edu, Carnegie Mellon University, USA; Sam Westrick, New York University, USA, swestric@cs.cmu.edu; Matthew Fluet, Rochester Institute of Technology, USA, mtf@cs.rit.edu; Umut A. Acar, Carnegie Mellon University, USA, umut@cs.cmu.edu.

Why would such a simple parallel program perform so poorly? The problem is that parallelism is not free: parallel code incurs overheads to spawn, schedule, and synchronize parallel tasks. For example, every iteration of a parallel loop can spawn a task to execute the body of the loop in parallel. Such a spawn operation requires thousands of cycles even with the fastest implementations on modern hardware [Ghosh et al. 2020a]. Yet, the body of a loop can be relatively tiny, even taking as few as a couple dozen cycles to complete (as in the parallel matrix multiplication example), causing the overhead of the spawn operations—rather than the actual matrix-multiplication work—to dominate execution time.

Today, we expect the programmer to control the cost-benefit ratio of parallelism by optimizing parallel loops via coarsening. Specifically, the programmer splits the loop into chunks and spawns only one task per chunk, thereby amortizing the cumulative overhead of parallelism [Acar et al. 2018; Tzannes et al. 2014; Westrick et al. 2024]. Such optimization requires great care, because if the chunks are too coarse, then they will reduce parallelism and harm scalability; if the chunks are too fine, then the overheads will be large. But what exactly are "too coarse" and "too fine"? This question is difficult, if not impossible, to answer because it depends on dynamic data, especially for high-level languages with generic or polymorphic data types. For example, the arguments to a parallel matrix multiplication function can be matrices of bits, floating point numbers, or an algebraic data structure; matrices may vary in size along each dimension and potentially from dense to sparse, or anything in between. All of these factors impact performance, but they are not statically known. Making matters worse, the architecture itself and even the software stack also impact performance. Thus, even if the programmer manages to coarsen perfectly, they end up overfitting the code to the architecture, software stack, and inputs considered, jeopardizing the portability of the program (e.g. [Acar et al. 2018; Tzannes et al. 2014; Westrick et al. 2024]).

Motivated by the challenges of manual performance optimization, researchers have sought automation. Prior work on heartbeat scheduling presented a scheduling technique that can provably efficiently amortize the overheads of parallelism [Acar et al. 2018; Rainey et al. 2021; Su et al. 2024]. Focusing on binary fork-join model of parallelism, more recent work [Westrick et al. 2024] has introduced the idea of "(automatic) parallelism management" by combining compiler and runtime techniques with heartbeat scheduling to manage parallelism fully automatically. Parallelism management allows the programmer to express all potential for parallelism without worrying about performance and relies on the programming language and the compiler suite to decide what and when to parallelize. Parallelism management shows encouraging results but is limited by a fundamental assumption: it considers only a binary fork-join model and does not provide direct support for the dominant form of parallelism: loops. Parallel loops may be encoded with binary fork-join parallelism but such an encoding introduces significant overheads and blocks a variety of loop-specific compiler optimizations.

In this paper we present Spork IR, an intermediate representation that enables parallelism management for key primitives including parallel loops, parallel reductions, and fork-join parallelism. Spork IR enables compiling parallel loops and reductions into a form which executes like a sequential, iterative loop by default, but can be parallelized at a moment's notice. Spork IR makes no restrictions on how loops may be used, and allows arbitrary nesting (including dynamically via function calls). To ensure efficiency without restricting generality, the IR includes two low-level control-flow constructs, called ***spork*** (**s**equential or **p**arallel f**ork**) and ***spoin*** (**s**equential or **p**arallel j**oin**). At a high level, spork marks points in a loop that may "go parallel" and symetrically, spoin marks the potential synchronization needed for the spork. From an operational perspective, each spork registers an alternative code path for a parallel task implicitly on the call stack, making its sequential execution cost essentially zero. If the runtime decides to "go parallel", it does so by

creating a proper task from the implicit representation. Each spork has a matching spoin that either continues sequentially or performs a task synchronization, decided automatically by the runtime.

The precise semantics of spork and spoin are subtle, and an efficient lowering to assembly code hinges on their use according to key invariants. We therefore formalize the semantics of spork and spoin in the context of an SSA-based IR, defining their operational semantics and a type system that enforces safety. We prove key type safety theorems in the Lean theorem prover.

To support performant parallel loops, we encode parallel loops by wrapping their body with a spork-spoin pair, registering a parallel task to complete the remainder of the iterations, while also executing sequentially when the runtime deems parallelism unnecessary. This allows the runtime to parallelize the loop at any moment by dynamically choosing to actually spawn the registered task. To guarantee that we can amortize the cost of the spawn operations, we use heartbeat scheduling [Acar et al. 2018; Rainey et al. 2021] to decide dynamically which spork primitives should go parallel and which should remain sequential.

We demonstrate that Spork IR is practical by implementing its formal semantics in the MPL compiler for the Parallel ML language. The implementation incorporates Spork IR and adapts existing optimizations and compilation passes appropriately. In addition to supporting many existing loop-specific optimizations, our implementation of spork (and spoin) allows for function inlining, a key property for efficient nested and/or tight loops. In addition to parallel for loops, Spork IR is capable of encoding parallel reductions and fork-join parallelism.

We evaluate the Spork IR approach to parallelism management by considering over a dozen benchmarks from the Parallel ML Benchmark Suite [Arora et al. 2021, 2023; Westrick et al. 2024]. These benchmarks cover a variety of problem domains, including graph analysis, computational geometry, sparse linear algebra, numerical algorithms, and text analysis, and include highly irregular and challenging instances of parallelism.

Our experiments show that managed parallel programs incur less than 70% overhead compared to their sequential counterparts while delivering good parallel speedups (an average of 28x on an 80-core machine). Notably, parallel programs fully managed by Spork IR, which require no manual optimizations for controlling parallelism overheads, are less than 32% slower on average than manually optimized parallel code for all core counts. These results show that parallelism management can enable programs written with high-level parallel constructs to achieve fast performance while managing thread creation, scheduling, and synchronization fully automatically.

The specific contributions of the paper include the following:

- the design of Spork IR with ***spork*** and ***spoin***, control-flow primitives enabling code to perform well both sequentially and in parallel,
- a formalization of the syntax, semantics, and type system of Spork IR, along with progress and preservation theorems proven in the Lean theorem prover,
- a compilation strategy for expressing parallel loops and fork-join parallelism using spork and spoin, including important optimizations,
- an end-to-end implementation in the MPL compiler and runtime system,
- and an empirical evaluation with over a dozen benchmarks written with high-level parallelism primitives, demonstrating that Spork IR is capable of guaranteeing low overhead and high scalability without requiring manual optimizations to control parallelism overheads.

2 Spork IR: An Intermediate Representation for Parallelism Management

We introduce Spork IR, an intermediate representation language suitable for efficient and automatic parallelism management at runtime. Spork IR is derived from static single assignment form (SSA), extending it with two additional control flow primitives for managing parallelism: *spork* and *spoin*.

Program	P	$::= \bar{F}$	*Atom*	m, n	$::= v \mid x$
Function	F, G	$::= \textbf{fun}\ f(\bar{x})\{\bar{B}\}$	*Value*	v	$\in \mathbb{Z}$
Basic block	B	$::= \textbf{block}\ b(\bar{x})\{C\}$	*Temporary*	x, y	
Block code	C	$::= x \leftarrow e; C \mid T$	*Function name*	f, g	
Expression	e	$::= m \mid -m \mid m + n \mid m < n \mid \ldots$	*Block label*	b	
Transfer	T	$::= \textbf{goto}\ b_{\text{next}}(\bar{x}) \mid \textbf{if}(e, b_{\text{then}}, b_{\text{else}}) \mid \textbf{call}\ f(\bar{x}) \triangleright b_{\text{ret}} \mid \textbf{return}(\bar{x})$			
		$\mid \textbf{spork}(b_{\text{body}} \parallel g_{\text{spwn}}(\bar{x})) \mid \textbf{spoin}(b_{\text{unpr}}, b_{\text{prom}})$			

Fig. 1. Syntax of Spork IR, with the new transfers highlighted: **spork** and **spoin**.

In the rest of this section, we define the syntax and semantics of Spork IR, describe its type system, and prove its type safety.

Note that Spork IR specifically aims to facilitate the compilation of efficient, automatically managed parallel loops, a goal that informed many of our design choices. The intermediate representation itself enforces several constraints which are not strictly necessary for a coherent semantics, but which prove absolutely crucial later on for lowering to efficient assembly code.

2.1 Syntax

Figure 1 defines the syntax of Spork IR. A program P is a list of first-order functions, one named main (we denote lists with a bar, e.g. $\bar{F}$). Each function has a name, list of parameters, and a list of basic blocks, including one marked as the function entry.

A basic block marks a sequence of straight-line code with no control flow except at the very end: it consists of a label, a list of parameters, and a list of assignments terminated by a control flow transfer. In addition to functions, basic blocks have parameters because static single assignment form requires every variable (hereafter called *temporary*) be assigned in exactly one place in the program, yet sometimes multiple source blocks need to send data to a shared target block (some SSA IRs use a ϕ function for this same goal). Assignments (e.g., $x \leftarrow y + z$) assign the value of an expression to a temporary, and transfers enable control flow across basic blocks (**goto**, **if**), functions (**call**, **return**), and in Spork IR, potentially across threads (**spork**, **spoin**).

Spork IR extends SSA by introducing the two new transfers highlighted in Figure 1. The $\textbf{spork}(b_{\text{body}} \parallel g_{\text{spwn}}(\bar{x}))$ ("**s**equential/**p**arallel f**ork**") transfer behaves as a $\textbf{goto}\ b_{\text{body}}()$, but it additionally opens a scope in which $g_{\text{spwn}}(\bar{x})$ is potential work for a new thread, should the program choose during execution to spawn a thread while inside the scope. The $\textbf{spoin}(b_{\text{unpr}}, b_{\text{prom}})$ transfer closes this scope, and performs a conditional jump: b_{unpr} ("unpromoted") if the program never spawned a thread executing $g_{\text{spwn}}(\bar{x})$, and b_{prom} ("promoted") if it did. In the latter case, the spawned child thread terminates when it reaches a $\textbf{return}(\bar{x})$ from the last stack frame in its call stack, which sends the values of $\bar{x}$ back to the parent thread and exits. Then, when the parent thread reaches $\textbf{spoin}(b_{\text{unpr}}, b_{\text{prom}})$, it synchronizes with the child thread and jumps to b_{prom}, passing it the values of $\bar{x}$ as arguments.

Thread pool	$\mathcal{R}$	$::=$	$\mathcal{T} \mid \mathcal{R}_p \curlywedge \mathcal{R}_c$
Thread state	$\mathcal{T}$	$::=$	$\mathcal{K} \diamond C$
Call stack	$\mathcal{K}$	$::=$	$\overline{k}$
Stack frame	k	$::=$	$\langle f, \rho, \mathcal{X}, b_{\text{ret}}? \rangle$
Spawn deque	ρ	$::=$	$\bar{\pi} : \bar{v}$
Spawn call	π, v	$::=$	$g_{\text{spwn}}(\bar{x})$
Value map	$\mathcal{X}, \mathcal{Y}$	$\in$	$(\textit{temp}) \rightharpoonup (\textit{value})$

Fig. 2. Definitions for Spork IR operational semantics

2.2 Operational Semantics

Formalizing this notion, we present definitions for the operational semantics of Spork IR in Figure 2 and the

$$\frac{\mathcal{R}_p \mapsto \mathcal{R}'_p}{\mathcal{R}_p \curlywedge \mathcal{R}_c \mapsto \mathcal{R}'_p \curlywedge \mathcal{R}_c}\textsc{cong-parent} \qquad \frac{\mathcal{R}_c \mapsto \mathcal{R}'_c}{\mathcal{R}_p \curlywedge \mathcal{R}_p \mapsto \mathcal{R}_p \curlywedge \mathcal{R}'_c}\textsc{cong-child}$$

$$\frac{\mathcal{X} \vdash e \Downarrow v}{\mathcal{K} \cdot \langle f, \rho, \mathcal{X} \rangle \diamond (x \leftarrow e); C \mapsto \mathcal{K} \cdot \langle f, \rho, \mathcal{X}[x \hookrightarrow v] \rangle \diamond C}\textsc{stmt}$$

$$\frac{\textbf{block } b_{\text{next}}(\bar{y})\{C\} \in f}{\mathcal{K} \cdot \langle f, \rho, \mathcal{X} \rangle \diamond \textbf{goto } b_{\text{next}}(\bar{x}) \mapsto \mathcal{K} \cdot \langle f, \rho, \mathcal{X}[\bar{y} \hookrightarrow \mathcal{X}(\bar{x})] \rangle \diamond C}\textsc{goto}$$

$$\frac{\mathcal{X} \vdash e \Downarrow v \quad v \neq 0 \quad \textbf{block } b_{\text{then}}()\{C\} \in f}{\mathcal{K} \cdot \langle f, \rho, \mathcal{X} \rangle \diamond \textbf{if}(e, b_{\text{then}}, \_) \mapsto \mathcal{K} \cdot \langle f, \rho, \mathcal{X} \rangle \diamond C}\textsc{if-then} \qquad \frac{\mathcal{X} \vdash e \Downarrow 0 \quad \textbf{block } b_{\text{else}}()\{C\} \in f}{\mathcal{K} \cdot \langle f, \rho, \mathcal{X} \rangle \diamond \textbf{if}(e, \_, b_{\text{else}}) \mapsto \mathcal{K} \cdot \langle f, \rho, \mathcal{X} \rangle \diamond C}\textsc{if-else}$$

$$\frac{\textbf{fun } g(\bar{y})\{\_\} \quad \textbf{block } \text{entry}()\{C\} \in g}{\mathcal{K} \cdot \langle f, \rho, \mathcal{X} \rangle \diamond \textbf{call } g(\bar{x}) \triangleright b_{\text{ret}} \mapsto \mathcal{K} \cdot \langle f, \rho, \mathcal{X}, b_{\text{ret}} \rangle \cdot \langle g, \varnothing, [\bar{y} \hookrightarrow \mathcal{X}(\bar{x})] \rangle \diamond C}\textsc{call}$$

$$\frac{\textbf{block } b_{\text{ret}}(\bar{x})\{C\} \in f}{\mathcal{K} \cdot \langle f, \rho, \mathcal{X}, b_{\text{ret}} \rangle \cdot \langle g, \varnothing, \mathcal{Y} \rangle \diamond \textbf{return}(\bar{y}) \mapsto \mathcal{K} \cdot \langle f, \rho, \mathcal{X}[\bar{x} \hookrightarrow \mathcal{Y}(\bar{y})] \rangle \diamond C}\textsc{return}$$

$$\frac{\textbf{block } b_{\text{body}}()\{C\} \in f}{\mathcal{K} \cdot \langle f, \bar{\pi} : \bar{v}, \mathcal{X} \rangle \diamond \textbf{spork}(b_{\text{body}} \parallel g_{\text{spwn}}(\bar{x})) \mapsto \mathcal{K} \cdot \langle f, \bar{\pi} : \bar{v} \cdot g_{\text{spwn}}(\bar{x}), \mathcal{X} \rangle \diamond C}\textsc{spork}$$

$$\frac{\forall \langle \_, \_ : \bar{v}, \_, \_ \rangle \in \mathcal{K}.\ \bar{v} = \varnothing \quad \textbf{fun } g(\bar{y})\{\_\} \quad \textbf{block } \text{entry}()\{C'\} \in g}{\mathcal{K} \cdot \langle f, \bar{\pi} : g_{\text{spwn}}(\bar{x}) \cdot \bar{v}, \mathcal{X}, b_{\text{ret}} \rangle \cdot \mathcal{K}' \diamond C \mapsto \mathcal{K} \cdot \langle f, \bar{\pi} \cdot g_{\text{spwn}}(\bar{x}) : \bar{v}, \mathcal{X}, b_{\text{ret}} \rangle \cdot \mathcal{K}' \diamond C \curlywedge \langle g_{\text{spwn}}, \varnothing, [\bar{y} \hookrightarrow \mathcal{X}(\bar{x})] \rangle \diamond C'}\textsc{promote}$$

$$\frac{\textbf{block } b_{\text{unpr}}()\{C\} \in f}{\mathcal{K} \cdot \langle f, \bar{\pi} : \bar{v} \cdot g_{\text{spwn}}(\bar{x}), \mathcal{X} \rangle \diamond \textbf{spoin}(b_{\text{unpr}}, \_) \mapsto \mathcal{K} \cdot \langle f, \bar{\pi} : \bar{v}, \mathcal{X} \rangle \diamond C}\textsc{spoin-unpr}$$

$$\frac{\textbf{block } b_{\text{prom}}(\bar{x})\{C\} \in f}{\mathcal{K} \cdot \langle f, \bar{\pi} \cdot g_{\text{spwn}}(\bar{x}) : , \mathcal{X} \rangle \diamond \textbf{spoin}(\_, b_{\text{prom}}) \curlywedge \langle g_{\text{spwn}}, \varnothing, \mathcal{Y} \rangle \diamond \textbf{return}(\bar{y}) \mapsto \mathcal{K} \cdot \langle f, \bar{\pi} : , \mathcal{X}[\bar{x} \hookrightarrow \mathcal{Y}(\bar{y})] \rangle \diamond C}\textsc{spoin-prom}$$

Fig. 3. Spork IR operational semantics

semantics themselves in Figure 3. We use $\varnothing$ for an empty list (or simply omit it) and write $\bar{x} \cdot \bar{y}$ for the concatenation of $\bar{x}$ and $\bar{y}$. A thread pool has the structure of a binary tree, with threads for leaves and with a node $\mathcal{R}_p \curlywedge \mathcal{R}_c$ denoting a fork where $\mathcal{R}_p$ is the parent of $\mathcal{R}_c$. Each thread consists of a call stack paired with the remaining code from the basic block it is executing. A call stack is a nonempty list of stack frames, each with four components: (1) the function f being executed by the frame, (2) a spawn deque ρ of spawn calls, one for each spork scope we are inside (local to this stack frame, i.e., those entered while this was the current stack frame) with the delimiter ":" separating promoted calls $\bar{\pi}$ from unpromoted calls $\bar{v}$, (3) a mapping $\mathcal{X}$ that stores the value of each temporary in scope, and (4) an optional continuation block b_{ret} for resuming this stack frame after a return, present in all but the current stack frame.

We define the execution of Spork IR via the small-step operational semantics in Figure 3. Each rule is of the form $\mathcal{R} \mapsto \mathcal{R}$, modifying the thread pool:

- **CONG-PARENT** and **CONG-CHILD** allow arbitrarily stepping in parts of the thread pool
- **STMT** evaluates e, associating x with its value in the current frame's value mapping.
- **GOTO** jumps to a new block, assigning values to its parameters from the arguments provided.
- **IF-THEN** and **IF-ELSE** rules perform conditional jumps: b_{then} if e evaluates to a nonzero value and b_{else} otherwise.
- **CALL** saves which block to return to, pushes a new stack frame onto the call stack, and initializes it by mapping from function parameters to the values of the arguments.
- **RETURN**, conversely, pops the current stack frame and returns to the caller's, passing the returned value(s) as arguments to b_{ret}.

- **SPORK** always jumps to b_{body}, but additionally pushes $g_{\text{spwn}}(\bar{x})$ to the end of the current frame's spawn deque, allowing it to be promoted to a thread later.
- **PROMOTE** may happen nondeterministically at any point while at least one frame's spawn deque on the call stack is nonempty. It finds the oldest unpromoted $g_{\text{spwn}}(\bar{x})$ across all stack frames (including the current frame), marks it as promoted, and spawns a new thread executing that call. **Importantly, this ensures spawn calls are promoted in order of oldest to newest.**
- **SPOIN-PROM** happens at a spoin when its associated spork was promoted, popping its spawn call $g_{\text{spwn}}(\bar{x})$ from the end of the (fully promoted) spawn deque. No unpromoted spawn calls remain, since promotions happen oldest-first and, conversely, spoin closes the most recent spork scope. Once the child thread ends with a $\textbf{return}(\bar{y})$ from its last stack frame, the original thread passes the values of $\bar{y}$ as arguments to the b_{prom} block.
- **SPOIN-UNPR** happens at a spoin when its associated spork remained unpromoted. It closes the spork scope by popping from the end of the spawn deque (which prevents $g_{\text{spwn}}(\bar{x})$ from being promoted in the future), then jumps to the b_{unpr} block.

2.3 Type System

The type system of Spork IR requires that programs satisfy certain properties necessary for the operational semantics and for lowering to efficient assembly code. Integers are the sole type of values in Spork IR, and while this suffices for our purposes, extending the language to include more types should be straightforward if desired. As a result, our typing rules are judgments of the form d WF ("well-formed"), where d can be a program, thread pool, or any of their constituent parts (expression, basic block, call stack, etc.), optionally with antecedents to the left of a turnstile $\vdash$.

To facilitate these *well-formedness* rules, we first enrich the syntax definitions by adding subscripted attributes to functions and basic blocks: $\textbf{fun}_r\, f(\bar{x})$ is a function where all returns have r values, and $\textbf{block}_{\Gamma;\rho}\, b(\bar{x})$ is a block statically nested under ρ sporks and with local temporary scope Γ. These attributes are statically inferred and have exactly one value for each function and basic block. A subscript may be omitted when its value does not matter.

Figure 4 lists the typing rules of well-formed Spork IR programs. Broadly speaking, a well-formed program is composed of well-formed functions, including a main function. Similarly, a well-formed function in program P is composed of well-formed basic blocks, one of which is named entry. A basic block is well-formed in program P and function f if its code is and its arguments do not shadow any temporaries bound elsewhere (since this is static single assignment). A rule of the form $P; f; \Gamma; \rho \vdash C\ \text{WF}_{\text{code}}$ states that code C is well-formed in program P and function f so long as every control flow path leading to this point binds at least those temporaries in Γ and always passes through the same sequence of open sporks—those without a closing spoin—with corresponding spawn calls ρ (which we also call its *spork nesting*). The auxiliary $f; \Gamma; \rho \vdash b(\cdot^r)\ \text{WF}_{\text{cont}}$ rule ("continuation") stipulates that b is a block in f with arity r, temporary scope Γ, and spork nesting ρ. Note, these rules do not distinguish between promoted and unpromoted spawn calls since they only consider a static program, not its dynamic execution.

These typing rules enforce a set of relatively standard invariants for an an SSA-based IR, e.g., temporaries are in scope everywhere they are used and function calls have the correct number of arguments. Additionaly, they enforce that sporks and spoins come in matching pairs, with no other control flow into, or out of, the code between. This invariant results in every basic block having a statically inferred spork nesting. Further, functions can only return after all sporks have been closed by a spoin, ensuring any potentially necessary synchronization has happened. In addition to being important for the type safety of the operational semantics, these properties prove absolutely crucial for lowering Spork IR to efficient assembly code, as they impose a static bound on the size

$$\frac{\forall F \in P.\ P \vdash F\ \mathrm{WF}_{\mathrm{fun}} \quad \mathbf{fun}\ \mathrm{main}()\{\_\} \in P}{P\ \mathrm{WF}_{\mathrm{program}}} \qquad \frac{\forall B \in \bar{B}.\ P; f \vdash B\ \mathrm{WF}_{\mathrm{block}} \quad \mathbf{block}_{\bar{x};\varnothing}\ \mathrm{entry}()\{\_\} \in \bar{B}}{P \vdash \mathbf{fun}\ f(\bar{x})\{\bar{B}\}\ \mathrm{WF}_{\mathrm{fun}}} \qquad \frac{P; f; \Gamma \cup \bar{x}; \rho \vdash C\ \mathrm{WF}_{\mathrm{code}} \quad \Gamma \cap \bar{x} = \varnothing}{P; f \vdash \mathbf{block}_{\Gamma;\rho}\ b(\bar{x})\{C\}\ \mathrm{WF}_{\mathrm{block}}}$$

$$\frac{\cdots}{\Gamma \vdash e\ \mathrm{WF}_{\mathrm{expr}}} \qquad \frac{\Gamma \vdash e\ \mathrm{WF}_{\mathrm{expr}} \quad P; f; \Gamma \cup \{x\}; \rho \vdash C\ \mathrm{WF}_{\mathrm{code}} \quad x \notin \Gamma}{P; f; \Gamma; \rho \vdash x \leftarrow e; C\ \mathrm{WF}_{\mathrm{code}}}$$

$$\frac{\bar{x} \subseteq \Gamma \quad f; \Gamma; \rho \vdash b_{\mathrm{next}}(\cdot^{|\bar{x}|})\ \mathrm{WF}_{\mathrm{cont}}}{P; f; \Gamma; \rho \vdash \mathbf{goto}\ b_{\mathrm{next}}(\bar{x})\ \mathrm{WF}_{\mathrm{code}}} \qquad \frac{\Gamma \vdash e\ \mathrm{WF}_{\mathrm{expr}} \quad \begin{array}{r} f; \Gamma; \rho \vdash b_{\mathrm{then}}(\cdot^0)\ \mathrm{WF}_{\mathrm{cont}} \\ f; \Gamma; \rho \vdash b_{\mathrm{else}}\ (\cdot^0)\ \mathrm{WF}_{\mathrm{cont}} \end{array}}{P; f; \Gamma; \rho \vdash \mathbf{if}(e, b_{\mathrm{then}}, b_{\mathrm{else}})\ \mathrm{WF}_{\mathrm{code}}}$$

$$\frac{P; \Gamma \vdash g(\bar{x})\ \mathrm{WF}_{\mathrm{call}} \quad f; \Gamma, \rho \vdash b_{\mathrm{ret}}(\cdot^r)\ \mathrm{WF}_{\mathrm{cont}}}{P; f; \Gamma; \rho \vdash \mathbf{call}\ g(\bar{x}) \triangleright b_{\mathrm{ret}}\ \mathrm{WF}_{\mathrm{code}}} \qquad \frac{\mathbf{fun}_r\ f(\_)\{\_\} \quad r = |\bar{x}| \quad \bar{x} \subseteq \Gamma}{P; f; \Gamma; \varnothing \vdash \mathbf{return}(\bar{x})\ \mathrm{WF}_{\mathrm{code}}}$$

$$\frac{P; \Gamma \vdash g_{\mathrm{spwn}}(\bar{x})\ \mathrm{WF}_{\mathrm{call}} \quad f; \Gamma; \rho \cdot g_{\mathrm{spwn}}(\bar{x}) \vdash b_{\mathrm{body}}(\cdot^0)\ \mathrm{WF}_{\mathrm{cont}}}{P; f; \Gamma; \rho \vdash \mathbf{spork}(b_{\mathrm{body}} \,\|\, g_{\mathrm{spwn}}(\bar{x}))\ \mathrm{WF}_{\mathrm{code}}} \qquad \frac{\mathbf{fun}_r\ g_{\mathrm{spwn}}(\_)\{\_\} \in P \quad \begin{array}{r} f; \Gamma; \rho \vdash b_{\mathrm{unpr}}\ (\cdot^0)\ \mathrm{WF}_{\mathrm{cont}} \\ f; \Gamma; \rho \vdash b_{\mathrm{prom}}(\cdot^r)\ \mathrm{WF}_{\mathrm{cont}} \end{array}}{P; f; \Gamma; \rho \cdot g_{\mathrm{spwn}}(\bar{x}) \vdash \mathbf{spoin}(b_{\mathrm{unpr}}, b_{\mathrm{prom}})\ \mathrm{WF}_{\mathrm{code}}}$$

$$\frac{\bar{x} \subseteq \Gamma \quad \mathbf{fun}\ f(\bar{y})\{\_\} \in P \quad |\bar{x}| = |\bar{y}|}{P; \Gamma \vdash f(\bar{x})\ \mathrm{WF}_{\mathrm{call}}} \qquad \frac{\mathbf{block}_{\Gamma';\rho}\ b(\bar{x})\{\_\} \in f \quad \Gamma' \subseteq \Gamma \quad |\bar{x}| = r}{f; \Gamma; \rho \vdash b(\cdot^r)\ \mathrm{WF}_{\mathrm{cont}}}$$

Fig. 4. Typing rules of well-formed programs

of ρ in every function and allow the compiler to know what the (local) spawn deque is for every program point.

We also define typing rules for thread pools, formalizing what it means for a program's execution state to be well-formed. While the semantics of Spork IR are inherently nondeterministic, they follow certain constraints that our typing rules for thread pools (and their constituents) capture. One such constraint is that promotions happen in order from oldest to newest. In Section 4.3 we discuss how this promotion order guarantees certain work- and span-efficiency bounds. A well-formed thread pool ensures invariants such as every stack frame except the current one expects as many arguments to be returned from its successor frame as the function that frame is executing returns. Also, it requires that there is a child thread running each promoted spawn call. For a well-formed program, we can derive that its initial thread pool—a leaf thread running the entry block of the main function—is well-formed. The full typing rules for thread pools are given in Appendix A.

2.4 Formalization of Spork IR

We have encoded the syntax and semantics of Spork IR in the Lean theorem prover [Moura and Ullrich 2021], along with our typing rules for programs and thread pools. Using this encoding, we have proven the following theorems for Spork IR in Lean:

Theorem 2.1 (Progress). *For a well-formed thread pool $\mathcal{R}$, either*

(1) $\mathcal{R} = \varnothing \cdot \langle f, \varnothing, \mathcal{X} \rangle \diamond \mathbf{return}(\bar{x})$ is a final leaf thread returning from its last stack frame,
(2) $\mathcal{R} = \mathcal{K} \cdot \langle f, \bar{\pi} \cdot g_{\mathrm{spwn}}(\bar{x}) : \varnothing, \mathcal{X} \rangle \diamond \mathbf{spoin}(b_{\mathrm{unpr}}, b_{\mathrm{prom}})$ is a blocked leaf thread awaiting the spork's promoted spawn call to finish,
(3) or $\mathcal{R} \mapsto \mathcal{R}'$ for some $\mathcal{R}'$

Theorem 2.2 (Preservation). *If $\mathcal{R} \mapsto \mathcal{R}'$ and $\mathcal{R}$ is well-formed, then $\mathcal{R}'$ is also well-formed.*

Theorem 2.3 (Inlining). *Inlining a function call preserves well-formedness.*

Theorems 2.1 and 2.2 (Progress and Preservation) demonstrate the type safety of Spork IR, guaranteeing that executing a well-formed program via the operational semantics never reaches an ill-formed state, either terminating with a final result or continuing with another step. Due to well-formedness, the second (blocked) case of Theorem 2.1 can only happend when $\mathcal{R}$ is the parent of another thread, i.e., it is the left branch of a fork node; consequently, it is not possible of a top-level thread pool node.

Function call inlining in Spork IR can be performed in the typical way. For example, $\textbf{call}\ f(\bar{x}) \triangleright b_{\text{ret}}$ would be inlined by adding the basic blocks of f to the current function's, changing the call to $\textbf{goto}\ \text{entry}_f(\bar{x})$, and replacing each $\textbf{return}(\bar{y})$ in the newly added blocks with $\textbf{goto}\ b_{\text{ret}}(\bar{y})$. Theorem 2.3 (Inlining) provides a certain sense of *composability*, where well-formed code may be nested inside other well-formed code and remain well-formed. This property is important in the following section, as it guarantees our implementation of parallel loops can be arbitrarily nested.

Our Lean formalization follows the definitions outlined in this paper very closely, but with one non-negligible difference which aids mechanization: rather than by names, our proofs refer to temporaries by their indices in the local scope. For example, $\textbf{block}_{\Gamma;\rho}\ b(x,y)\{z \leftarrow x+y; \textbf{goto}\ b'(z)\}$ for some $|\Gamma| = 2$ would instead be written $\textbf{block}_{\rho}\ b(4)\{\Gamma[2] + \Gamma[3]; \textbf{goto}\ b'(\Gamma[4])\}$. In place of block parameters is the number of temporaries already in scope plus the number of parameters—in this case, $b(x,y)$ becomes $b(|\Gamma| + 2) = b(4)$. Similarly, the assignment implicitly binds the fifth index (counting from 0), $\Gamma[4]$, instead of giving it the name z. In order to give a more accessible and familiar presentation in this paper, we use named temporaries and an implicit local scope instead.

3 Encoding Parallelism in Spork IR

In this section, we consider an ML-like (higher-order, polymorphic, etc.) source language supporting parallelism via several primitive higher-order functions and introduce encodings of these functions in Spork IR. In particular, we consider:

$$\begin{aligned}
\texttt{par} &: (\texttt{unit} \to \alpha) \times (\texttt{unit} \to \beta) \to \alpha \times \beta \\
\texttt{parfor} &: \texttt{int} \times \texttt{int} \times (\texttt{int} \to \texttt{unit}) \to \texttt{unit} \\
\texttt{reduce} &: \texttt{int} \times \texttt{int} \times (\texttt{int} \to \alpha) \times (\alpha \times \alpha \to \alpha) \times \alpha \to \alpha
\end{aligned}$$

The semantics of $\texttt{par}(f, g)$ is to execute $f()$ and $g()$ in parallel, returning a tuple of their results. Semantically, $\texttt{parfor}(i, j, f)$ executes all of $\{f(i), f(i+1), \ldots, f(j-1)\}$ in parallel. In a similar way, $\texttt{reduce}(i, j, f, c, z)$ computes the "sum" of $\{f(i), f(i+1), \ldots, f(j-1)\}$ with respect to the binary, associative "combining" function c and corresponding "zero" element z. In fact, `parfor` is just a special case of `reduce`, using the trivial combining function over the type `unit`.[1] Therefore, throughout the paper, we will refer to `reduce` as a "parallel loop", where the function f is the "body" of the loop. This primitive parallel loop can be used to implement a wide variety of common parallel operations on sequences, such as `map`, `filter`, `scan` (prefix sums), `flatten`, and many others [Westrick et al. 2022b].

While our approach is fully able to express efficient fork-join parallelism, the central contribution of this paper is a technique for compiling and executing parallel loops efficiently. Our approach guarantees low overhead relative to a sequential, iterative loop on a single core while maintaining high scalability on many cores, regardless of what code appears within the loop body. This is difficult because loops can contain arbitrary code in their loop bodies, including other (nested) loops, which might be hidden behind function calls, perhaps recursively. It is also common to see

[1] In our actual implementation, this is optimized away by the compiler, producing an efficient implementation of `parfor`.

"tight loops" with just a handful of instructions in the loop body. Loops may also be irregular and/or data-dependent, with no statically predictable cost within each body, and varying costs across different iterations within the same loop. Our goal is to ensure that all parallel loops perform well, in all possible cases, with no need for programmer intervention.

3.1 Encoding par in Spork IR

Before a program is lowered to Spork IR, we assume it has been subjected to standard compiler transformations (monomorphization, defunctionalization, ...) which for each source-level call to `par`(f, g) result in a specialized, first-order variant $\text{PAR}_{f,g}()$. Once the program is lowered to Spork IR, we introduce definitions for each of these PAR functions as shown in Figure 5.

$\text{PAR}_{f,g}()$ begins in the *entry* block, which immediately sporks, jumping to the *evalF* block and then calling $f()$. If the program decides to promote something while evaluating $f()$ and there are no older unpromoted sporks, it spawns a new thread running $g()$ in parallel. When the original thread finishes evaluating $f()$, it continues execution at *check*, with x storing the result of $f()$. Then, it spoins, checking if a promotion occurred. If another thread did run $g()$, it synchronizes with that thread by waiting for it to finish and then jumping to *join* with y_1 storing the result of $g()$. If the spork remained unpromoted, spoin takes the other path and jumps to *evalG*. In this case it still needs to execute $g()$, doing so in the same thread and then returning the two results.

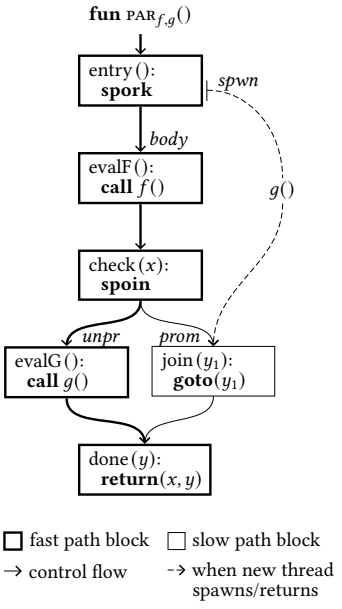


Fig. 5. Implementing par in Spork IR for a particular f, g.

The common case of $\text{PAR}_{f,g}()$ is when no promotion occurs, indicated by bolded boxes in the figure. In this (sequential) case, the fast path only incurs the overhead of a jump at spork (usually eliminated by block ordering) and a conditional at spoin. Nonetheless, it remains able to be parallelized when needed. This implementation of $\text{PAR}_{f,g}()$ can itself be inlined by an optimizing compiler if beneficial, and in no way interferes with the contents of f and g being inlined on the fast path as well (though g must remain defined for the spawn call on the slow path). Since spork and spoin can be safely viewed by most compiler optimizations as simple control flow transfers, using $\text{PAR}_{f,g}()$ to enable parallelism when needed instead of calling $f()$ and $g()$ in sequence does little to disrupt existing compiler optimizations for the fast path.

3.2 Encoding Parallel reduce in Spork IR

To meet our goal of low-overhead, fully parallelizable loops that perform well in any context, we propose a novel technique for encoding parallel `reduce` in Spork IR. Our approach functions very similarly to an intraprocedural, iterative loop, but nevertheless is able to be split into an arbitrary number of parallel tasks. Our design is motivated by the principle that *fast sequential performance leads to fast parallel performance*: because the common case is sequential, optimizing for this case results in each parallel thread performing its own (sequential) work more efficiently, resulting in a faster program even in a parallel setting.

Towards this end, we wrap the loop body inside a spork-spoin pair. If the program promotes the spork during the execution of a loop iteration, it breaks off all remaining iterations, splits them in half, and executes each half (potentially) in parallel. Therefore, this typically results in three threads: one completing only the rest of the original loop iteration and two new threads executing each half

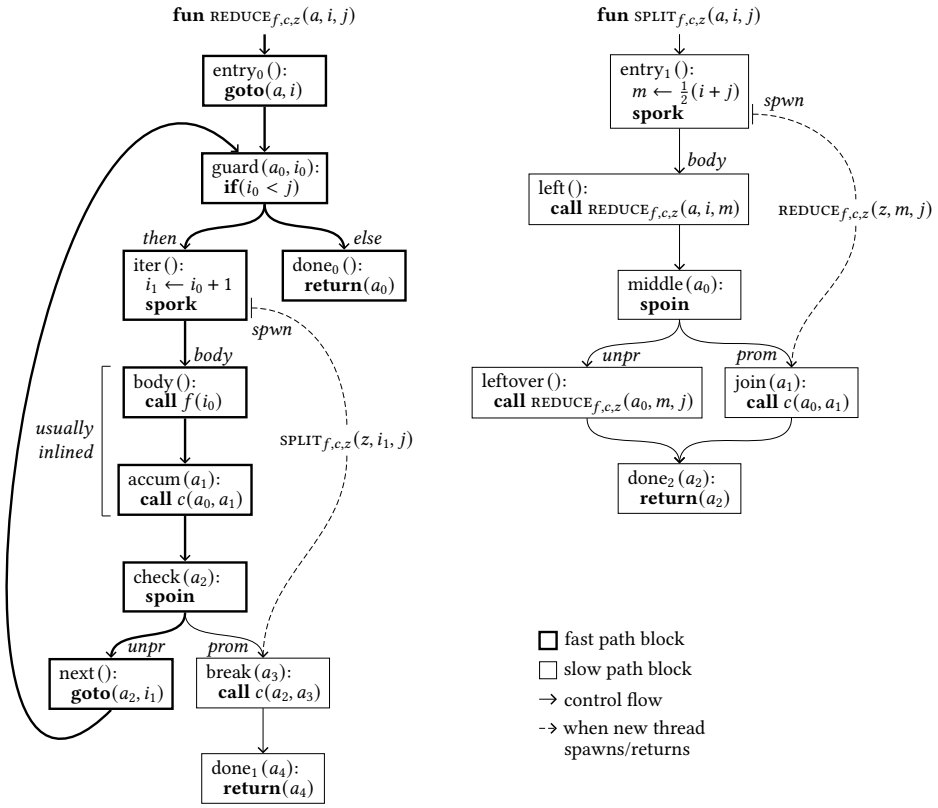


Fig. 6. Implementing parallel reduce in Spork IR for a particular f, c, z. For each call to the higher-order reduce(i, j, f, c, z), we generate specialized first-order functions REDUCE$_{f,c,z}(a, i, j)$ and SPLIT$_{f,c,z}(a, i, j)$ unique to that call. The calls to f and c on the fast path may (and often will) be inlined.

of all remaining iterations. Each of these new threads reverts to the initial, iterative behavior of the loop, but remains ready to be split even further in the same way.

Just as with par, we assume the program has already been subjected to various compiler transformations by the time it is lowered to Spork IR, having changed each source-level call to reduce(c, z, i, j, f) into a specialized, first-order variant REDUCE$_{f,c,z}(z, i, j)$.

Using spork and spoin, we implement parallel reduce in Spork IR for a particular f, c, and z as in Figure 6. Aside from promotions (which are amortized by sequential work), this REDUCE is much like an iterative, sequential loop. The function's implementation starts with the *guard* block, which checks if the loop is incomplete (that is, $i < j$). If there remains work to do, it sporks: by default, the program continues to the *body* block, which calls f with the current iteration index i_0 and then combines the result with the current accumulator a_0 by calling c. If the body of the loop (blocks *body* and *accum*) completes without the spork being promoted, spoin jumps to the unpromoted continuation *next*, which returns to the loop guard for the next iteration i_1.

However, if the program decides to parallelize while evaluating the loop body and finds that this is the oldest unpromoted spork, it creates a new thread running SPLIT$_{f,c,z}(z, i_1, j)$. Then, the original thread resumes its execution and, when finished, spoins: since the spork's potential parallelism was promoted, it waits for the newly spawned thread to finish, passing its final return value as an

argument to *break*. It then calls the combine operator c with the accumulated result of the iterations up to this point (a_1) and the result of the rest as computed on the spawned thread (a_2), returning that value.

When a thread is spawned running $\textsc{split}_{f,c,z}(z, i_1, j)$, it splits the range in half and executes each half potentially in parallel, in a very similar way to PAR (though $\textsc{split}_{f,c,z}(z, i_1, j)$ makes a small optimization by passing a_0 as the starting accumulator to the $\textsc{reduce}_{f,c,z}(a_0, m, j)$ in the unpromoted case, rather than having an additional combine call c). Each half of the remaining loop iterations runs by switching back to the iterative approach of REDUCE.

This implementation allows for every single loop iteration to become a task with its own thread if needed. Additionally, f and c can be inlined in the loop body, allowing arbitrary nesting of `reduce`. This is important for the performance of short, tight loops and nested parallel loops: the modest overhead of a function call for every loop iteration can be detrimental to the overall performance of the program. Our design allows for inlining to avoid this, as the fast path of `reduce` becomes entirely intraprocedural (having no function call) when f and c are inlined. Moreover, since the control flow of the loop is intraprocedural, it enables existing (sequential) loop optimizations to work on the fast path (e.g. code motion, loop unrolling, loop unswitching).

4 MPL$^{\text{sp}}$: A Compiler for Automatically Managed Parallel Loops

We present **MPL$^{\text{sp}}$**, a compiler that produces efficient parallel loops by implementing Spork IR, and a runtime system that uses heartbeat scheduling to schedule promotions, and work-stealing scheduling to load-balance promoted tasks. **MPL$^{\text{sp}}$** is derived from MPL, which itself extends MLton [MLton nd; Weeks 2006], a whole-program optimizing compiler for Standard ML, with support for fork-join style parallelism. MPL inherits many features from MLton, especially in terms of the compiler proper; the most substantial changes are localized to the runtime system to support thread scheduling and memory management and to the implementation of the (extended) standard library, where a significant portion of thread scheduling and memory management is implemented in source SML code with calls to MPL runtime-system functions as necessary.

The most challenging aspect of the implementation is the maintenance of the call-stack, which serves as the interface between the runtime system and the generated code. Our goal is to lower Spork IR into executable code that carefully constructs and maintains a call-stack which can be interrupted at regular intervals by the runtime system and subjected to a promotion operation. Following Acar et al. [2018], promotions are scheduled periodically by the runtime system (across all active threads), and each promotion selects the oldest promotable spork-spoin pair, to guarantee that the asymptotic parallelism of the computation is preserved. We describe the promotion scheduling algorithm in more detail in Section 4.3.

Key Ideas and Section Overview. At a high level, the key idea for call-stack maintenance is to statically associate each spork-spoin pair with a designated stack slot called a *spork slot*. The spork slot is used to keep track of whether or not its associated spork has already been promoted; the corresponding spoin can then inspect this slot to conditionally continue down the fast path or redirect onto the slow (synchronization) path. To promote the oldest spork-spoin pair within a call-stack, the runtime system executes the following steps.

(1) Walk the call-stack to find the oldest promotable frame, and identify the oldest promotable spork-spoin pair within this frame.
(2) Allocate a *join token*, i.e., a synchronization object which will be used to synchronize the newly exposed task.
(3) Store (a pointer to) the join token into the appropriate spork slot of the identified frame.

(4) Copy the frame, and initialize the copy as a new thread to execute the "spwn" branch of the spork.

The remainder of this section explains how exactly the idea above is implemented in **MPL^{sp}**. In Section 4.1, we describe how Spork IR is extended with explicit handling of join tokens, and how the use of join tokens allows us to decouple the implementation of the compiler from the scheduler. In Section 4.2, we explain how spork slots are associated with spork-spoin pairs (which is complicated by the fact that spork-spoin pairs can be freely nested), illustrate a promotion operation on an example program, and make precise how spork and spoin are lowered into assembly-level instructions. Section 4.2.2 describes how spork and spoin are embedded into the compiler proper and how these primitives interact with closure conversion, exceptions, and exceptional control-flow. Section 4.3 describes how promotions are scheduled, in particular, making use of a token accounting method previously proposed by Westrick et al. [2024]. Token accounting is closely integrated with work-stealing scheduling and the source-level embedding of spork-spoin pairs; we wrap up this section by describing these interactions in Section 4.4 and Section 4.5, respectively.

4.1 Join Tokens and the Separation of Compiler and Scheduler

Implementing the semantics of Spork IR in **MPL^{sp}** requires integration with the thread-scheduling components of MPL. In particular, the **PROMOTE** rule creates a new thread and the **SPOIN-PROM** rule synchronizes two threads. This creates a tension, because MPL's thread scheduling and memory management is implemented outside of the compiler proper, in the runtime system and in source SML code with calls to runtime-system functions. This separation is good engineering practice, as it allows the thread-scheduling (including promotion and synchronization) and memory-management components of MPL to be implemented in high-level programming languages, rather than a low-level compiler intermediate representation. A direct implementation of Spork IR would require the back-end of the compiler to lower spoin and (some) return transfers to uses of synchronization operations, but those operations are only indirectly available to the compiler, in the sense that they are part of the program being compiled but are not otherwise distinguished.

To resolve this tension, we implement the promotion and synchronization aspects of Spork IR in source SML code and the runtime system and the control-flow aspects in the compiler proper. To motivate our implementation, we briefly sketch a simple variant of Spork IR using a revised implementation of `par` (Figure 7). There are two changes made in this variant. First, the spawn argument of spork is no longer a function call $g_{\text{spwn}}(\bar{x})$, but the label of a new basic block b_{spwn} which simply performs **call** $g_{\text{spwn}}(\bar{x})$ The second is that a parent and child synchronize explicitly via a *join token*[2] that is created at the time of a promotion. The join token is passed to the b_{spwn} block of the promoted spork transfer and to the b_{prom} block of the matching spoin transfer. The compiler only "knows" about spork and spoin transfers, while the SETJOIN and GETJOIN functions are implemented in source SML code with calls to MPL runtime-system functions. While the Spork IR **SPOIN-UNPROM** rule only allows a parent to synchronize with the termination of its most recent child, join tokens are more general, allowing any thread executing GETJOIN to synchronize with and receive the value of any other

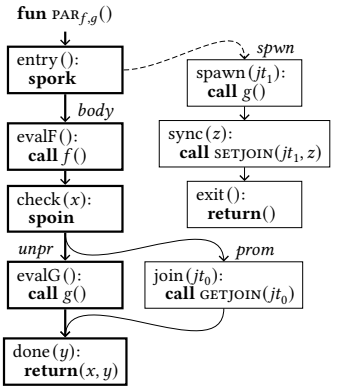


Fig. 7. Revised implementation of `par`.

[2]Essentially, an Id-style write-once synchronous variable [Arvind et al. 1989]

thread executing SETJOIN with the same join token; such a synchronization mechanism is widely available in parallel languages.

4.2 Implementing Spork IR in the Compiler

4.2.1 Back-end Changes: Using Frames to Implement spork, spoin, and Promotion. The most challenging aspect of the implementation is to efficiently track the nesting of sporks in a manner that both (1) allows the runtime system to identify the oldest spork that can be promoted, and (2) admits efficient implementations of spork and spoin transfers, particularly the determination of whether or not the last spork was promoted.

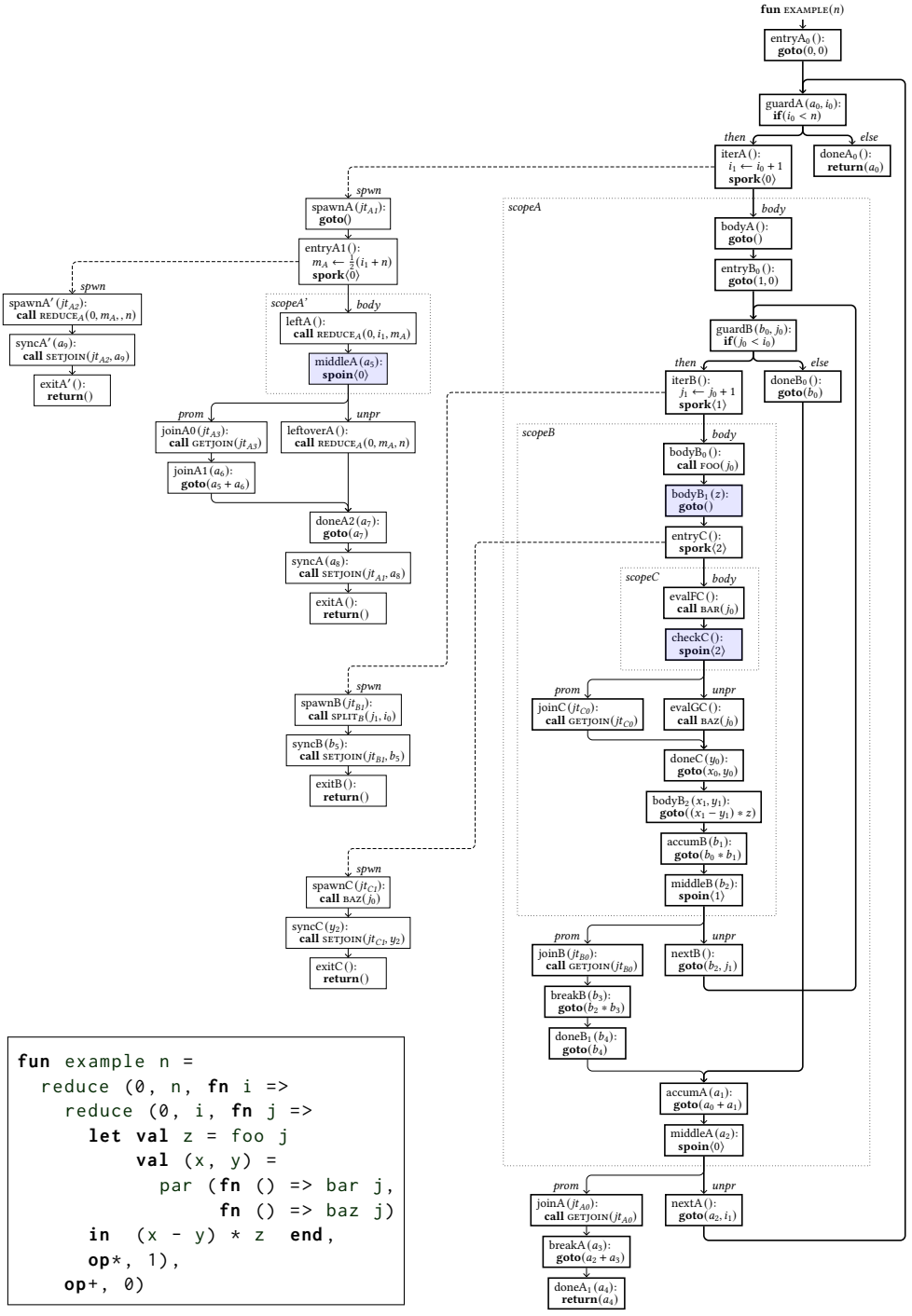
Properly nested spork scopes and spork nestings. The primary insight is that the idiomatic use of spork and spoin to implement `reduce` and `par` introduces spork and spoin in matching pairs that delimit *spork scopes* that are *properly nested* (if, due to inlining, there are multiple spork-spoin pairs in a function). Informally, a spork's scope is the region of the control-flow graph that must be entered via the b_{body} of the spork and exited via the matching spoin. Proper nesting means that, for any distinct pair of sporks in a function, their scopes are either disjoint or one is a proper subset of the other. This property is asserted for Spork IR by the well-formedness rules discussed in Section 2.4 and Appendix A.

Figure 8 shows the control-flow graph, after inlining, for an example function that performs a `par` within doubly-nested `reduce`s. The dotted boxes denote the spork scopes, where *scopeC* nests within *scopeB*, *scopeB* nests within *scopeA*, and *scopeA′* is disjoint from the other spork scopes.

Given the control-flow graph of a function with properly nested spork scopes, we can perform a simple analysis to statically determine, for each control-flow point, its *spork nesting*. A spork nesting is a sequence, where the first element is the outermost (largest) scope and the last element is the innermost (smallest) scope. Each scope can also be associated with the b_{spwn} label of its delimiting spork. For example, at block bodyB_1, the spork nesting is [*scopeA*/spawnA, *scopeB*/spawnB] while at block checkC, the spork nesting is [*scopeA*/spawnA, *scopeB*/spawnB, *scopeC*/spawnC]. Note that each spork scope occurs at the same index in each spork nesting of which it is a member; this index can be associated with the scope's delimiting spork and spoin. For example, the spork in block iterB is annotated with $\langle 1 \rangle$ because *scopeB* always occurs at index 1.

Lowering spork and spoin. Using these observations, we can give a realization of the spawn deque and implementations of spork and spoin transfers and of promotion. During lowering, when the call stack is made explicit, the back-end reserves *spork slots*: a contiguous sequence of N slots at the bottom of a function's stack frame, where N is the maximum length of any spork nesting of the function. At a control-flow point with an associated spork nesting of length n, the bottom n spork slots are *active* and the remaining slots are *inactive*. For example, the stack frame for the function from Figure 8 requires three spork slots and, when control is at block bodyB_1, slots 0 and 1 are active, corresponding to *scopeA* and *scopeB*, and slot 2 is inactive; see Figure 10a. Our invariant is that, when at a control-flow point, each inactive spork slot is `NULL` and that each active spork slot is `NULL` when it corresponds to a spork scope that has not been promoted and is non-`NULL` when it corresponds to a spork scope that has been promoted. To establish this invariant, the back-end extends the function prologue with a write of `NULL` to each of the spork slots, since function execution begins in an empty spork nesting and all spork slots are inactive.

Figure 9 shows the lowering of spork and spoin transfers to (pseudo) assembly code. A spork transfer is lowered to nothing more than a jump to b_{body}; the corresponding spork slot is transitioning from inactive to unpromoted active, so no change to the spork slot is required. A spoin transfer is lowered to a sequence that reads from the spork slot at the spoin's associated index (by loading from an offset of the frame pointer), compares the read value with `NULL`, conditionally branches




```
fun example n =
  reduce (0, n, fn i =>
    reduce (0, i, fn j =>
      let val z = foo j
          val (x, y) =
            par (fn () => bar j,
                 fn () => baz j)
      in  (x - y) * z  end,
      op*, 1),
    op+, 0)
```

Fig. 8. Control-flow graph for **fun** `example n = ...` after inlining, demonstrating nested **spork** scopes. The highlighted basic blocks are control-flow points referenced in text and Figure 10.

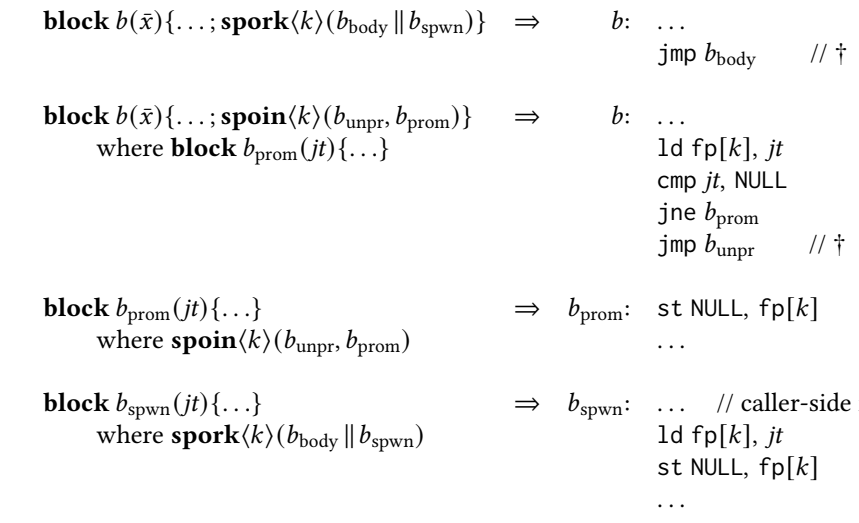


Fig. 9. Lowering of **spork** and **spoin**. Marked jmp instructions are likely eliminated by basic-block ordering.

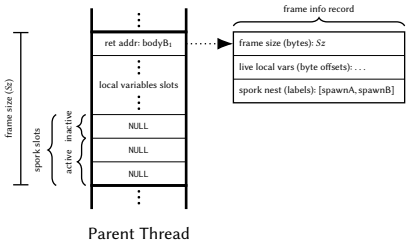
when false to b_{prom}, and (when true) jumps to b_{unpr}. The lowering of a b_{prom} block begins by storing NULL to the slot since the corresponding spork slot is transitioning from promoted active to inactive. The b_{prom} block argument jt is the non-NULL value read from the spork slot in the lowering of the spoin transfer; as described earlier, the non-NULL value that is passed to b_{prom} will be (a pointer to) a join token used to obtain the final value from the child thread, although the entire compiler is agnostic to the meaning of the non-NULL value.

Note that these lowerings yield an extremely efficient fast (sequential) path: a spork performs only a jump and a matching spoin performs only a read, a comparison, a (failing) conditional branch, and a jump; moreover, the jumps will typically be eliminated by basic-block ordering.

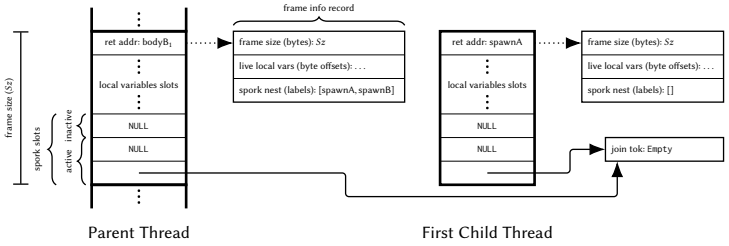
Promotion. Although promotion is implemented in the runtime system, it must manipulate call stacks, and therefore, we discuss it here. The promotion procedure is invoked with a call stack and a fresh join token and must walk the call stack to find and promote the oldest unpromoted spork.

From MLton, a call-stack is a contiguous sequence of frames delimited by stack-bottom and stack-top pointers; a frame collects local variables that are live when a function is suspended at a call and stores a return address at the top of the frame. Each return address can be mapped, via static data emitted by the compiler, to a *frame information record* that includes a frame size and an array recording the frame offsets of live pointers for precise garbage collection. To walk the call stack, the promotion procedure initializes a walk pointer with the stack-top pointer and iterates over each frame by reading the return address pointed to by the walk pointer and decrementing the walk pointer by the size recorded in the corresponding frame info until the walk pointer is equal to the stack-bottom pointer. **MPL$^{\text{sp}}$**, in addition to reserving spork slots at the bottom of each frame, extends the frame info with the spork nesting (as an array of b_{spwn} labels) of the control-flow point that corresponds to the return address. For example, the frame info for bodyB_1 (the return address for the **call** FOO(j_0) of block bodyB_0) includes the spork nest [spawnA, spawnB] (Figure 10a).

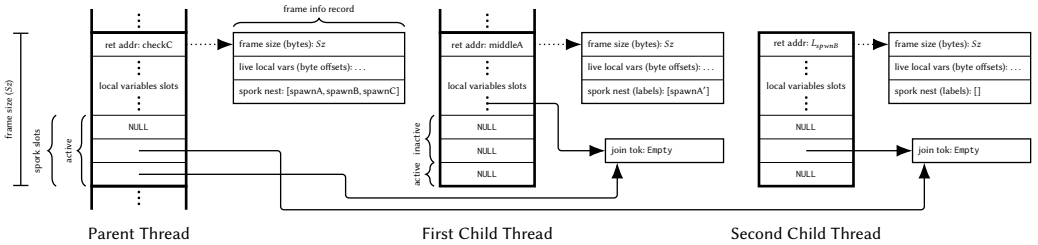
Based on the invariant for active spork slots, the promotion procedure must find the oldest (lowest in the call stack) NULL active spork. In order to distinguish between active and inactive NULL spork slots, the promotion procedure uses the length of the spork nesting from the frame info. Once the promotion procedure has found the correct frame and active spork slot, it writes



(a) Immediately before first promotion; parent thread is executing **call** FOO(j_0) (that returns to bodyB$_1$).



(b) Immediately after first promotion.



(c) Immediately after second promotion; parent thread has proceeded and is now executing **call** BAR(j_0) (that returns to checkC); first child thread has started and is now executing **call** REDUCE$_A(0, i_1, m_A)$ (that returns to middleA). Note the state of the first child thread has no influence on the second promotion.

Fig. 10. Call stacks of parent and child threads during promotions. Note that stacks grows upwards.

the (non-NULL) join token into the found active spork slot. Next, the found frame (including the newly written non-NULL join token) is copied to the bottom of a new call stack, the b_{spwn} label from the found frame's frame info's spork nesting at the index corresponding to the found active spork slot is written to the copied frame's return address, and NULL is written to all of the spork slots with lower indices than the found spork slot. These writes correspond to inactivating spork slots, since the b_{spwn} control-flow point is not in any spork scope. Figure 10 illustrates the effects of two promotions on a parent thread executing the example function from Figure 8.

The lowering of the b_{spwn} block of a spork transfer is handled specially (Figure 9). It is treated as the return block of a call that returns no results and, after performing the caller-side of the returning convention (e.g., adjusting the frame pointer), a value is read from the spork slot at the spork's associated index, NULL is written to that slot (inactivating it, since the b_{spwn} control-flow point is not in any spork scope), and execution continues with the read value as the b_{spwn} argument. For example, in Figure 10, when the first child thread starts executing, the join token passed via a

spork slot (Figure 10b) is moved to a local variable slot (Figure 10c), from which it will be accessed for the eventual SETJOIN operation.

4.2.2 Front-End and Closure-Conversion Changes. No changes to the syntax or type checking of the source language were made to support spork and spoin. Instead, we added a polymorphic, higher-order `prim_spork_spoin` primitive to the compiler. Compiler primitives are exposed as functions in a generic manner and `prim_spork_spoin` required no special handling. Because Standard ML is a higher-order language, it is easy to expose the non-trivial control-flow of spork and spoin as a higher-order primitive. The earliest phase of the compiler that required changes was the closure-conversion phase, which is responsible for transforming a higher-order IR into a first-order SSA IR, using defunctionalization [Reynolds 1972] guided by a monovariant whole-program control-flow analysis [Cejtin et al. 2000].

To the source program, the primitive is simply a polymorphic higher-order function, used as

$$\begin{aligned}\texttt{prim\_spork\_spoin}\,(&\mathit{tag} : \mathsf{int}, f_{\text{body}} : \mathsf{unit} \to \alpha, f_{\text{spwn}} : \delta \to \zeta,\\ &f_{\text{unprVal}} : \alpha \to \gamma, f_{\text{unprExn}} : \mathsf{exn} \to \gamma,\\ &f_{\text{promVal}} : \alpha \times \delta \to \gamma, f_{\text{promExn}} : \mathsf{exn} \times \delta \to \gamma) : \gamma\end{aligned}$$

The *tag*, which must be a compile-time constant, is associated with the spork and included in the spork nestings added to frame infos; it is used to communicate a policy that is used at promotion (see Section 4.3). The f_{body} and f_{spwn} functions correspond to the code for the homonymous edges of the introduced spork. Instead of a single matching spoin, the lowering of `prim_spork_spoin` introduces *two* matching spoins; one spoin, with the f_{unprVal} and f_{promVal} functions corresponding to the code for the b_{unpr} and b_{prom} edges, is executed if f_{body} terminates with a value and the other spoin, with f_{unprExn} and f_{promExn} for b_{unpr} and b_{prom}, is executed if f_{body} terminates with an uncaught exception. If, during optimization, f_{body} and the functions it calls are inlined (as is often the case), the resulting control-flow graph will goto directly from the returning of a value to the value spoin and goto directly from the raising of an exception to the exception spoin. One motivation for this value/exception split is that it would be incorrect for control to leave the spork scope via an uncaught exception (rather than via a matching spoin). We describe a second performance motivation in Section 4.5. The δ argument corresponds to the arbitrary data value stored in the spork slot when promoted. Although this data value will always be a join token used for synchronization, making the `prim_spork_spoin` polymorphic with respect to it emphasizes that the compiler makes no assumptions about it and treats it opaquely.

The primitive posed little difficulty for the control-flow analysis or defunctionalization transformation of the closure-conversion phase. Translating a `prim_spork_spoin` simply amounts to building an SSA control-flow-graph fragment that performs the appropriate defunctionalized calls in the code executed by a spork and its two matching spoins. The complexity of building SSA IR control-flow graphs is mediated by a direct-style interface that is inspired by the CPS translation [Kelsey 1995]. Importantly, this translation of `prim_spork_spoin` guarantees that the resulting SSA IR functions have properly nested spork scopes.

4.3 Parallelism Management: Heartbeat Scheduling Promotions

While the **MPL$^{\text{sp}}$** compiler is responsible for the low-level compilation that yields an efficient implementation of spork and spoin transfers, the thread-scheduling component of **MPL$^{\text{sp}}$**, implemented in source SML code and the runtime system, is responsible for the promotion strategy. **MPL$^{\text{sp}}$** uses heartbeat scheduling [Acar et al. 2018; Rainey et al. 2021] with a token accounting algorithm [Westrick et al. 2024]: each time a thread performs N units of work, it receives C tokens that must be eagerly spent to promote the oldest unpromoted sporks on the thread's call stack

(with each promotion costing one token), but can be banked if the thread has no promotable sporks. Eager spending means that a thread must check for unspent tokens when entering a spork scope (and spend one immediately to promote this spork); we handle this aspect in Section 4.5. This algorithm guarantees work- and span-efficiency [Westrick et al. 2024]: if a program has work W and span S (excluding the costs of promotions) and a promotion costs τ, then the program will perform at most $\frac{C}{N}W$ promotions and have at most total work $(1 + \frac{C \cdot \tau}{N})W$ and total span $(\tau + N)S$ (including the costs of promotions).

Explicitly counting and checking steps of (non-promotion) work by each thread would be prohibitively expensive; a practical application of heartbeat scheduling approximates work done by the passage of (wall-clock) time. An interval timer delivers a `SIGALRM` to the program with period N and a signal handler grants each active thread C heartbeat tokens and attempts promotions. The N and C parameters are tuned for a particular hardware-software stack, but not for a particular program. In **MPL$^{\text{sp}}$** for the hardware described in Section 5, we set N to $500\mu s$ and C to 30 to ensure, on average, $500\mu s/30 \approx 16\mu s$ of work per promotion.

When a parent has excess tokens at a promotion, it has the option of giving some of those tokens to the spawned child (without violating the efficiency guarantees). A spork is tagged with a token-sharing policy: either give half of the parent's excess tokens to the child or give all of them. The sporks in PAR and in the generated SPLIT helper function of a REDUCE use the first policy (see Figures 5 and 6), since the body and the (potential) child thread are typically of comparable work, while the spork in REDUCE uses the second policy, since the remaining loop iterations are expected to be significantly more work than the one current loop iteration. We consider the reverse—when a child with excess tokens joins with its parent—in the next section.

4.4 Work-Stealing Scheduler

To execute threads on processors, **MPL$^{\text{sp}}$** uses a fork-join work-stealing scheduler, which provides an opportunity for additional behavior. When a child is spawned at a promotion, it is pushed to the back of a scheduler deque, from which it can be stolen by a worker for execution. With work-stealing, the GETJOIN operation first observes, by attempting to pop from the back of the scheduler deque, whether or not the child was stolen.[3] If it was, then a full synchronization with the corresponding SETJOIN must occur to obtain a value from the child. But, if it was not, then the parent can choose how to proceed. It could interpret this as though no promotion happened, in which case it jumps to the b_{unpr} code. This is the choice we make for the spoins in PAR and SPLIT. But, for the spoin in REDUCE, we execute the spawn call *in the current thread* and then jump to the b_{prom} block. Even though the child was not stolen, the fact that a promotion occurred prompts splitting the loop.

When a stolen child joins with its parent, it gives all of its excess tokens (not necessarily the same ones that it was given at its promotion) to its parent. When an unstolen child is observed by its parent, the treatment of its excess tokens (necessarily the same ones that it was given at its promotion) depends on the token-sharing policy of the spork. If the child received half of its parent's excess tokens, then they are discarded; it is typically unhelpful to encourage additional promotions with more tokens if child threads are not being stolen for execution. But, if the child received all of its parent's excess tokens, then they are all returned to the parent; in REDUCE, this means that the excess tokens will be available to be fairly shared by the spork of SPLIT.

[3] If the deque is empty, then the child was stolen; otherwise, the back element is the unstolen child.

```
fun promote thrd = runtime_promote (thrd, newJoinTok ())
datatype 'a result = Val of 'a | Exn of exception
fun extract res = case res of Val v => v | Exn exn => raise exn
datatype tokshr_policy = GIVE_NONE | GIVE_HALF | GIVE_ALL
fun spork_spoin (policy: tokshr_policy, body: unit -> 'a, spwn: unit -> 'b,
                 unpr: 'a -> 'c, prom: 'a * 'b -> 'c, unstolen: 'a -> 'c): 'c =
 let
  fun body' () =
   let val _ = if tokens () > 0 then promote (Thread.current ()) else ()
   in  body ()  end
  fun spwn' (jt: 'b jointok) = let val sr = Val (spwn ()) handle exn => Exn exn
                               in  setJoin (jt, sr) ; Thread.exit ()  end
  fun unprVal' bv = unpr bv
  fun unprExn' exn = raise exn
  fun promVal' (bv, jt: 'b jointok) = case getJoin jt of
                                        NONE => unstolen bv
                                      | SOME sr => prom (bv, extract sr)
  fun promExn' (exn, jt: 'b jointok) = (getJoin jt ; raise exn)
  val tag = encodePolicy policy
 in
  prim_spork_spoin (tag, body', spwn', unprVal', unprExn', promVal', promExn')
 end
```

Fig. 11. spork_spoin function that wraps a use of the prim_spork_spoin primitive

4.5 Integration via Source SML Code

A spork_spoin function finishes the implementation of the Spork IR semantics, by performing the necessary integration with the synchronization, parallelism management, and work-stealing components around a use of prim_spork_spoin. We must ensure that the f_{spwn} function seen by the primitive ends with a SETJOIN followed by a thread exit and that the f_{promVal} and f_{promExn} functions begin with a GETJOIN (Section 4.1). We must immediately trigger a promotion if the current thread has excess tokens (Section 4.3) and we safely expose the token-sharing policies (Section 4.3). We expose an additional possible code path to be used when a child is spawned by a promotion but is not stolen (Section 4.4). And, we reify (and later propagate) exceptions raised by the execution of a child thread (giving precedence to exceptions raised by the body). This well-behaved spork_spoin function (Figure 11) can be used to robustly implement higher-level parallel operations.

We focus again on the fast (sequential) path that excludes "user code": the (implicit, compiler-implemented) spork, execution of body' without an eager promotion and excluding body, the (implicit, compiler-implemented) spoin, execution of unprVal' excluding unpr. Compared to the fast pass described at the end of Section 4.2.1, this adds only a read of the current thread's tokens (stored as thread-local metadata), a comparison, and a (failing) conditional branch.

We also provide a performance reason for prim_spork_spoin to handle exceptions. Suppose the lowering of prim_spork_spoin only introduced one matching spoin. spork_spoin would be responsible for ensuring that an exception raised by the spork body is propagated across the spoin, using reification as with the child thread. body' would end with `Val (body ()) handle exn => Exn exn`, which incurs an allocation, and, instead of both unprVal' and unprExn', there would be a single `fun unpr' br = unpr (extract br)`, which incurs a case analysis. Although **MPL$^{\text{sp}}$** employs an efficient bump allocator, even this single allocation and case analysis can add significant overhead to an otherwise non-allocating loop that is executed many times; moreover, these allocations are extremely short lived and can induce additional garbage collections. Although we do not give a

detailed evaluation along this dimension in Section 5, we observe that having this allocation and case analysis on the fast path is 1.14x slower on average on both single core and 80 cores.

Using `spork_spoin`, we implement `reduce` and `par` entirely in source SML code. The combination of monomorphisation, defunctionalization, inlining, and SSA IR optimizations specializes each use of `reduce` and `par` to their call-sites, yielding the control-flow graphs from Figures 6, 5, and 8.

5 Evaluation

Our goal is to eliminate the burden of manually tuning parallelism grains, automatically achieving good performance across any number of cores. We evaluate the effectiveness of our approach with **MPL^{sp}** towards this goal on a benchmark suite of parallel programs. In our evaluation, we study three parts:

(1) In Section 5.2, we show that our technique achieves low overheads on a single core relative to sequential elision, averaging **1.69x** slower. At the same time, it maintains good parallel scalability, delivering **46.9x** self-speedup on 80 cores (a **27.7x** speedup over sequential).
(2) In Section 5.3, we demonstrate that compared to manually tuned parallel code, our technique needs no tuning yet introduces only **1.13x** overhead on a single core and **1.32x** on 80 cores.
(3) In Section 5.5, we find our implementation of parallel loops improves upon the divide-and-conquer approach, getting **1.74x** faster on a single core and **1.45x** on 80 cores.

5.1 Experimental Setup and Benchmarks

Experiments are run on an 80-core machine equipped with two 2.30GHz Intel Xeon (40-core) Platinum 8380 CPUs and 256GB of memory, running Ubuntu 22.04.4 LTS and Linux kernel version 5.15.0-101-generic. We use MLton version 20241230. Benchmark timings are evaluated with a 5 second warmup and then by taking the average of 20 back-to-back runs. For more stable results, we disable hyperthreading and pin experiments to particular cores.

We consider 16 benchmarks from the Parallel ML Benchmark Suite [Arora et al. 2021, 2023; Westrick et al. 2024], covering a variety of problem domains such as graph analysis, computational geometry, sparse linear algebra, numerical algorithms, and text analysis. In all our experiments, the code for the benchmarks is identical across systems except for the implementation strategy of `reduce` and `par` and the presence (or absence) of manually tuned parallelism grains.

5.2 MPL^{sp} has low sequential overhead and good parallel scalability

We evaluate parallel each **MPL^{sp}** program against its sequential elision to determine (a) the overheads of our fully parallelizable approach in comparison to a fast sequential implementation, and (b) the scalability of our approach on an increasing number of processors. Our sequential implementations of `par` and `reduce` are shown in Figure 12, and compiled with MLton.

Table 1 shows our results on 1 and 80 cores (T_1 and T_{80}) alongside the sequential elision (T_s) and the corresponding sequential overheads and parallel speedups. The column titled T_1/T_s shows the overhead of using potentially parallel code in **MPL^{sp}** instead of purely sequential code even when only one core is available, with an average of 1.69x overhead. In 12 of the 16 benchmarks, **MPL^{sp}** has less than 2x overhead. **MPL^{sp}** also maintains good parallel scalability, with 27.7x average speedup in comparison to sequential on 80 cores. In Figure 13, we also plot the self-speedup of **MPL^{sp}** across a variety of core counts and observe generally that performance improves as the

```
fun par (f, g) = (f (), g ())
fun reduce (i, j, f, c, a) =
 if i >= j then a else
  reduce (i+1, j, f, c, c (a, f i))
```

Fig. 12. Sequential implementation of `par` and `reduce`.

Table 1. Single-core (T_1) and 80-core (T_{80}) times for **MPL^{sp}** vs. sequential elision (T_s), measured in seconds, alongside sequential overhead and parallel speedup on 80 cores vs. sequential.

		MPL^{sp}		Overhead	Speedup
Benchmark	T_s	T_1	T_{80}	T_1/T_s	T_s/T_{80}
bfs	2.86	3.10	.092	**1.08**	**31.0**
bignum-add	.384	.861	.015	**2.24**	**25.4**
delaunay	4.91	7.54	.458	**1.54**	**10.7**
grep	1.74	2.41	.041	**1.39**	**42.1**
line-fit	.327	1.28	.037	**3.90**	**8.89**
mandelbrot	1.80	2.66	.039	**1.48**	**46.6**
map-heavy	3.41	4.21	.055	**1.23**	**61.9**
map-light	.326	.982	.034	**3.02**	**9.60**
merge-sort	3.40	6.13	.091	**1.80**	**37.5**
nearest-nbrs	.965	1.34	.027	**1.39**	**35.4**
nqueens	1.14	1.47	.022	**1.28**	**52.1**
primes	1.28	2.12	.057	**1.65**	**22.5**
sparse-mxv	1.03	1.75	.036	**1.71**	**28.4**
suffix-array	2.30	2.77	.060	**1.21**	**38.1**
triangle-count	5.35	8.94	.144	**1.67**	**37.1**
word-count	.488	1.11	.022	**2.28**	**21.7**
geomean				**1.69**	**27.7**

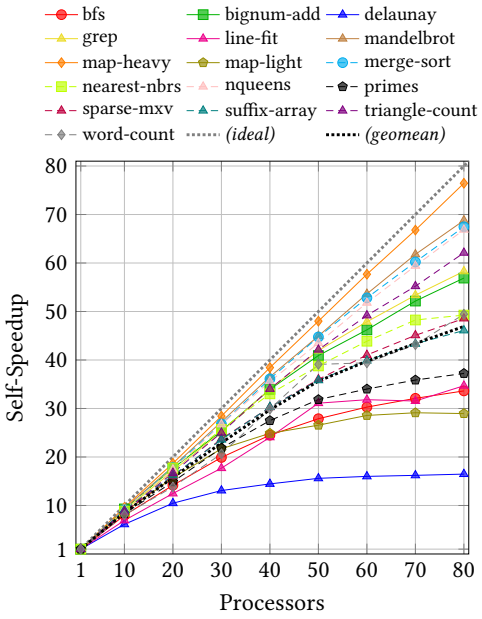


Fig. 13. Self-scalability of **MPL^{sp}** to different processor counts.

number of cores increases, with 46.9x average self-speedup on 80 cores relative to **MPL^{sp}**'s single-core time T_1. These results demonstrate that our approach is able to maintain high scalability, even without any manual tuning or chunking of parallel loops.

The benchmarks *bignum-add*, *line-fit*, *map-light*, and *word-count* exhibit larger overheads. These benchmarks are dominated by an extremely tight loop with only a few instructions per iteration, which stresses our approach and magnifies any per-loop overhead. We inspected the code generated for *map-light* and observed that some of the overhead is due to inefficient register allocation, resulting in unnecessary stack spilling on the fast path, which could be avoided with further optimization effort. The primitives spork and spoin offer new opportunities for compiler optimizations, in particular by identifying performance-sensitive loop bodies and explicitly distinguishing between fast and slow paths. We believe that this information could be exploited in future work to further close the gap between sequential and parallel implementations.

5.3 MPL^{sp} competes with manually tuned parallelism

In this experiment, we compare against manually tuned parallel code. Since this code has manually amortized the overheads of parallelism already, we use eager implementations of primitives as shown in Figure 14. In comparison, **MPL^{sp}** eliminates the need for any manual tuning while averaging only 1.13x overhead on a single core and 1.32x overhead on 80 cores. Each of the manually tuned programs compiled requires a grain size at each `reduce` call site, specifying the number of loop iterations to allocate to each task for that operation. This is in contrast to the otherwise identical programs compiled with **MPL^{sp}**, which needs no

```
fun par_e (f, g) = [primitive]
fun foldl (i, j, f, c, a) =
  if i >= j then a else
    foldl (i+1, j, f, c, c (a, f i))
fun reduce (GR, i, j, f, c, z) =
 if j-i <= GR then foldl (c, z, i, j, f) else
  c (par_e (reduce (GR, i, (i+j)/2, f, c, z),
            reduce (GR, (i+j)/2, j, f, c, z)))
```

Fig. 14. Manually tuned implementation of `reduce`, which uses eager $\texttt{par}_e$ and requires a grain size `GR` (highlighted above) to be chosen at every call-site.

Table 2. Overheads of our approach (**MPL^{sp}**) vs. manually tuned code, which needs call site-specific parallelism grains.

	Manual		Overhead	
Benchmark	T_1	T_{80}	$\frac{T_1(\textbf{MPL}^{sp})}{T_1(\text{Manual})}$	$\frac{T_{80}(\textbf{MPL}^{sp})}{T_{80}(\text{Manual})}$
bfs	3.07	.071	**1.01**	**1.30**
bignum-add	.737	.011	**1.17**	**1.41**
delaunay	7.26	.234	**1.04**	**1.96**
grep	2.36	.034	**1.02**	**1.21**
line-fit	.675	.020	**1.89**	**1.83**
mandelbrot	2.38	.031	**1.12**	**1.24**
map-heavy	4.23	.055	**.996**	**1.01**
map-light	1.08	.032	**.909**	**1.06**
merge-sort	4.53	.064	**1.35**	**1.42**
nearest-nbrs	1.35	.025	**.997**	**1.11**
nqueens	1.65	.023	**.889**	**.966**
primes	1.99	.053	**1.06**	**1.08**
sparse-mxv	1.75	.034	**1.00**	**1.06**
suffix-array	3.82	.069	**.726**	**.871**
triangle-count	4.26	.064	**2.10**	**2.23**
word-count	.703	.010	**1.58**	**2.35**
geomean			**1.13**	**1.32**

Table 3. Improvement factors of our `reduce` (**MPL^{sp}**) over divide-and-conquer implementation.

	D&C		Improvement	
	T_1	T_{80}	$\frac{T_1(\text{D\&C})}{T_1(\textbf{MPL}^{sp})}$	$\frac{T_{80}(\text{D\&C})}{T_{80}(\textbf{MPL}^{sp})}$
	6.17	.154	**1.99**	**1.67**
	1.76	.027	**2.04**	**1.80**
	7.99	.396	**1.06**	**.866**
	4.91	.073	**2.04**	**1.76**
	2.60	.039	**2.03**	**1.06**
	4.16	.056	**1.56**	**1.46**
	4.25	.055	**1.01**	**1.01**
	4.48	.110	**4.56**	**3.24**
	6.02	.094	**.981**	**1.03**
	1.43	.028	**1.06**	**1.01**
	2.20	.030	**1.50**	**1.38**
	7.26	.153	**3.43**	**2.69**
	5.87	.078	**3.35**	**2.16**
	4.29	.091	**1.54**	**1.51**
	9.90	.150	**1.11**	**1.04**
	2.06	.028	**1.85**	**1.23**
			1.74	**1.45**

parallelism grain control and automatically manages task creation at heartbeats. Full results of this comparison are shown in Table 2.

5.4 Our spork and spoin outperform Westrick et al. [2024]'s PCall

Westrick et al. [2024] provide an automatically managed implementation of `par`; their approach is based on a single primitive called PCall which is essentially a promotable function call. As shown in Section 3.1 and Figure 5, `par` can also be implemented using our spork and spoin primitives. An immediate question is how this spork-spoin-based implementation of `par` compares against Westrick et al. [2024]'s PCall-based implementation. We have measured that using spork and spoin to implement `par` is approximately 15% faster on average. In other words, spork and spoin appear to be both more general and more efficient than prior work on PCall-based parallelism management. The spork and spoin primitives are more general in the sense that they can efficiently encode not just `par`, but also parallel loops and reductions, as we develop in this paper. We believe the reason for the 15% speedup on average is due to better compiler optimizations, especially inlining, which can be blocked in certain cases by Westrick et al. [2024]'s PCall-based approach. Full results of this comparison are available in Appendix B.

5.5 MPL^{sp}'s `reduce` outperforms the divide-and-conquer implementation

In this section, we compare our implementation of `reduce` against the divide-and-conquer implementation using `par` in Figure 15. This divide-and-conquer approach is that taken by Westrick et al. [2024], which assumes binary fork-join parallelism and does not have primitive support for parallel loops. Both implementations use automatic parallelism management to amortize the cost of task creation at heartbeats, but our approach additionally benefits from a fast path which avoids the cost of splitting the loop. In particular, our approach amortizes not just the cost of task creation, but also the cost of computing loop-range midpoints and performing recursive calls for the left and right halves of the range.

This comparison uses **MPL^{sp}** as the underlying system for all measurements, only measuring the difference between two implementations of `reduce`. For the divide-and-conquer code, we use **MPL^{sp}**'s spork-spoin-based `par`.

```
fun par (f, g) = [primitive]
fun reduce (i, j, f, c, z) =
 if i >= j then z else
 if i+1 = j then f i else
   c (par (reduce (i,(i+j)/2,f,c,z),
           reduce ((i+j)/2,j,f,c,z)))
```

Fig. 15. Divide-and-conquer `reduce`, implemented using the automatically managed `par` primitive.

We observe that benchmarks using our `reduce` are on average 1.74x and 1.45x faster than when they use the divide-and-conquer approach on 1 and 80 cores, respectively, as shown in Table 3. The biggest improvements are in the benchmarks that most heavily rely on parallel loops, particularly those with very tight and/or nested loops. For example, on *map-light*, our **MPL^{sp}** exhibits 4.56x improvement on a single core; this benchmark simply iterates over a large array (200 million elements) and increments every element by 1. Both *primes* and *sparse-mxv-csr* employ nested parallel loops with tight inner loops, and we observe 3.43x and 3.35x improvement on a single core.

Improvements on 80 cores are similar but smaller, which is expected because the additional splitting costs incurred by the divide-and-conquer `reduce` are all local overheads which parallelize well. Of the 80-core benchmarks, **MPL^{sp}**'s `reduce` also outperforms divide-and-conquer in all but one case, *delaunay*.

The *delaunay* benchmark is challenging because it has little theoretical parallelism. Indeed, Figure 13 shows that *delaunay* parallelizes less than any of the other benchmarks, gaining only 16.5x self speedup on 80 cores. The benchmark performs many short bursts of parallel computation interspersed by sequential work, making the end-to-end running time highly sensitive to how quickly each parallel section "ramps up". Our automatically managed implementation of `reduce` in **MPL^{sp}** can take approximately twice as many promotion tokens to disperse computation across all processors compared to divide-and-conquer, due to the way it splits: the first promotion generates a new task, which immediately begins executing the first half of the remaining loop iterations. However, this task must wait for a second promotion before it can being executing the second half. Existing work has shown that it is possible to increase the heartbeat rate on stock hardware [Rainey et al. 2021; Su et al. 2024], which if applied in this case could improve scalability by decreasing the delay between successive heartbeats and thereby supplying promotion tokens more rapidly. Nevertheless, even in the case of low parallelism in *delaunay*, **MPL^{sp}**'s `reduce` is only 13% slower than divide-and-conquer on 80 cores.

This supports our claim that *fast sequential performance leads to fast parallel performance*, since despite the fact that it takes as many as twice the promotions to effectively split a loop, our **MPL^{sp}**'s `reduce` beats the divide-and-conquer implementation (with a sizeable average margin of 45% faster) on all but one benchmark for 80 cores, and even then, is only 13% slower.

6 Related Work

The most closely related work to our paper is the recent work on parallelism management [Westrick et al. 2024], which proposed the idea of fully automating parallelism management by combining compilation and runtime techniques with recent advances in parallel programming languages and scheduling. Our work differs from this prior work on parallelism management in that it supports parallel loops as well as fork-join parallelism, whereas prior work considers fork-join only. Even though parallel loops may be encoded in terms of fork-join parallelism, such an encoding incurs significant overhead and prohibits loop optimizations by translating loops into recursive functions. To support parallel loops (and fork-join parallelism) efficiently, we propose extending the intermediate representation used by compilers with two key control-flow constructs, enabling an encoding of a variety of parallelism primitives while facilitating parallelism management. Our

work builds on a relatively broad array of prior and recent research works, which we review in the rest of this section.

Scheduling techniques. All high-level parallel programming languages rely on a runtime scheduler for managing tasks/threads, including their creation and load-balancing among the available cores. Nearly all known schedulers today go back to Brent's seminal work in 1970s [Brent 1974], which established a bound of $\frac{W}{P} + S$ for scheduling a task-parallel program on P processors in terms of total work W and span S. Subsequent work generalized the bound to greedy scheduling [Arora et al. 2001; Eager et al. 1989], to randomized work-stealing [Arora et al. 2001; Blumofe and Leiserson 1999], and to account for data locality [Acar et al. 2015, 2002; Blelloch and Gibbons 2004; Chowdhury and Ramachandran 2008; Lee et al. 2015; Spoonhower et al. 2009] and responsiveness [Muller et al. 2020; Muller and Acar 2016; Muller et al. 2017, 2018, 2023, 2019]. None of this work accounts for the cost of spawning a task/thread.

Lazy task creation and lazy scheduling. In early 1990s, Mohr introduced lazy task creation to mitigate task overheads [Mohr et al. 1991] and efficient implementation techniques have been developed for futures and parallel calls [Feeley 1992, 1993a; Goldstein et al. 1996]. Follow-up work adopted the idea for work-stealing schedulers [Bergstrom et al. 2012; Hiraishi et al. 2009; Kumar et al. 2012; Tzannes 2012; Tzannes et al. 2010, 2014] and developed related techniques such as the *clone optimization* [Frigo et al. 1998] to further mitigate scheduler overheads. These techniques are able to spawn additional tasks in response to system load imbalance, and can help guarantee low overhead for "sequentialized" tasks, i.e., tasks that are never spawned, or tasks that are spawned but never migrated to another processor.

Granularity control. Task creation overheads can also be tamed using *granularity control* techniques, where the goal is to ensure that every spawned task executes a sizeable amount of work. Granularity control can be performed manually (e.g., by hardcoding sequential cutoffs and/or task size parameters), but this approach has major limitations with respect to portability, accuracy, and code modularity [Tzannes 2012; Westrick et al. 2024]. Numerous approaches and techniques have been proposed to address the limitations of manual granularity control [Duran et al. 2008; Huelsbergen et al. 1994; Iwasaki and Taura 2016; Loidl and Hammond 1995; Lopez et al. 1996; Pehoushek and Weening 1990; Shen et al. 1999; Weening 1989], relying on assumptions such as statically predictable time complexities, user annotations, or access to dynamic profiling data. Subsequent work combines static annotations and dynamic profiling to provide the first provable guarantee of low overhead and high scalability, using an approach called oracle-guided granularity control [Acar et al. 2019, 2011, 2016a]. This approach requires the user to supply cost functions for parallel code, which is sometimes difficult and in general not always possible.

Heartbeat scheduling. Recent work has taken a new approach based on a technique called hartbeat scheduling [Acar et al. 2018], which in principle is both provably efficient—ensuring low overhead and high scalability in all cases—and fully automatic—requiring no user annotation or manual tuning. The idea is to lazily create tasks according to a regular periodic pulse, a "heartbeat". At every pulse, each processor spawns the oldest queued task. This approach guarantees every spawn can be charged against work completed between heartbeats; additionally, as proven by Acar et al. [2018], it guarantees that the critical path length of the computation is stretched by at most a constant factor, i.e., all theoretical parallelism is asymptotically preserved.

Implementing heartbeat scheduling in practice requires a low-level preemption mechanism (such as software polling [Basu et al. 2021; Feeley 1993b; Ghosh et al. 2020b]) to respond to heartbeats in a timely manner, which can be challenging to incorporate automatically into compiler-generated code without sacrificing sequential efficiency. Early implementations of heartbeat scheduling had minimal compiler support and required significant manual rewriting to ensure efficiency in

practice [Acar et al. 2018; Rainey 2023; Rainey et al. 2021]. Recently, Su et al. [2024] demonstrated that heartbeat scheduling is capable of outperforming manual tuning for data-dependent and/or irregular workloads. Their approach places some restrictions on loop bodies (e.g., they do not support nested loops hidden behind a function call) and more generally they do not consider higher-order functions and integration with automatic memory management and scheduling. Our approach is most similar to *automatic parallelism management* [Westrick et al. 2024] which guarantees efficiency and scalability in a high-level fork-join language. This prior work only supports a binary fork-join model of parallelism, which is insufficient to guarantee low overheads relative to sequential loops, a limitation which we address in this paper.

MPL. We implemented our approach in MPL ("MaPLe"), which has been exploring efficient and scalable parallel functional programming by coupling thread scheduling and memory management for nested fork-join parallelism [Acar et al. 2015] through disentanglement [Arora et al. 2021; Westrick et al. 2022a, 2020] and hierarchical heaps [Guatto et al. 2018; Raghunathan et al. 2016]. **MPL$^{\text{sp}}$** is the second version of MPL that employs heartbeat scheduling for automatic parallelism management; it succeeds MPL$^{\text{s}}$ ("Sugar MaPLe") [Westrick et al. 2024], which used a *potentially parallel function call* (PCall) primitive to efficiently implement the coarse-grained `par`, but does not provide primitive support for fine-grained parallel loops and `reduce` as we do in this paper.

Language-level support for parallelism. A variety of languages have been developed with parallel primitives built directly into the compiler and runtime system. Examples include multiLisp [Halstead 1984], NESL [Blelloch 1996], Cilk [Frigo et al. 1998; Schardl and Lee 2023; Schardl et al. 2017], OpenMP [OpenMP Architecture Review Board [n. d.]], several extensions of Java [Bocchino et al. 2009; Imam and Sarkar 2014; Lea 2000], X10 [Charles et al. 2005], parallel Haskell [Li et al. 2007; Marlow and Peyton Jones 2011; Peyton Jones et al. 2008], and several forms of parallel ML [Arora et al. 2021, 2023; Elsman and Henriksen 2023; Fluet et al. 2011, 2007; Guatto et al. 2018; Raghunathan et al. 2016; Sivaramakrishnan et al. 2020, 2014; Spoonhower 2009; Westrick et al. 2024, 2020]. Language-level support for parallelism often comes as structured parallel primitives like fork-join (e.g., a binary "par"), parallel loops, futures, and async-finish, all closely related [Acar et al. 2016b].

Automatic parallelization. There has been significant research on automatic parallelization of sequential programs. As a starting point, automatic parallelization takes a sequential program and attempts to identify sections of the program that may be executed in parallel with a variety of techniques such as static and dynamic analysis (e.g., [Misailovic et al. 2013; Rinard and Diniz 1996; Rugina and Rinard 1999]) and speculation (e.g., [Prabhu and Olukotun 2005]). In contrast, we do not attempt to automatically convert a sequential program into a parallel equivalent. Rather, the goal of our work is to start from a "fully parallel" program—where the programmer has explicitly specified what is often *too much* parallelism—and to automatically manage the existing parallelism, coarsening where needed, and thereby avoid the need for any manual granularity control.

Automatic parallelization techniques may also need to manage parallelism [Rugina and Rinard 1999] and can benefit from our techniques. Although automatic parallelization can be effective in certain cases, it is limited by the degree of parallelism it extracts from sequential programs, which can severely limit opportunities for parallelism by creating long dependency chains.

7 Conclusion

We present Spork IR, an intermediate representation for parallelism management that can express parallel loops, parallel reductions, and fork-join parallelism.

The paper formalizes the semantics of Spork IR, establishes its key soundness properties in the Lean theorem prover and presents an implementation of the IR by extending the MPL compiler for Parallel ML. The implementation leverages heartbeat scheduling to amortize the overheads of

parallel task creation. Our evaluation with a broad set of benchmarks shows that our parallelism management techniques deliver excellent performance while requiring no effort to control the overhead of parallelism. Notably, the implementation delivers performance within 32% on average of manually optimized benchmarks across all core counts. These results show that parallelism management can make programs written with high-level parallelism primitives performant, thus making progress on the long-standing challenge of bridging the benefits of high-level parallelism abstractions with high performance.

Attribute TODO: NDSEG etc.

References

Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. 2019. Provably and Practically Efficient Granularity Control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. 214–228.

Umut A. Acar, Guy Blelloch, Matthew Fluet, Stefan K. Muller, and Ram Raghunathan. 2015. Coupling Memory and Computation for Locality Management. In *Summit on Advances in Programming Languages (SNAPL)*.

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The Data Locality of Work Stealing. *Theory of Computing Systems* 35, 3 (2002), 321–347.

Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. 769–782.

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2011. Oracle Scheduling: Controlling Granularity in Implicitly Parallel Languages. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 499–518.

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2016a. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming (JFP)* 26 (2016), e23.

Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. 2016b. Dag-calculus: A Calculus for Parallel Computation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. 18–32.

Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space Efficient Parallel Functional Programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)*.

Jatin Arora, Sam Westrick, and Umut A. Acar. 2023. Efficient Parallel Functional Programming with Effects. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1558–1583. doi:10.1145/3591284

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* 34, 2 (2001), 115–144.

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989), 598–632.

Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. 2021. Frequent Background Polling on a Shared Thread, Using Light-Weight Compiler Interrupts. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1249–1263. doi:10.1145/3453483.3454107

Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2012. Lazy Tree Splitting. *J. Funct. Program.* 22, 4-5 (Aug. 2012), 382–438.

Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

Guy E. Blelloch and Phillip B. Gibbons. 2004. Effectively sharing a cache among threads. In *SPAA* (Barcelona, Spain).

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46 (Sept. 1999), 720–748. Issue 5.

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (Orlando, Florida, USA) *(OOPSLA '09)*. 97–116.

Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM* 21, 2 (1974), 201–206.

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-directed Closure Conversion for Typed Languages. In *Proceedings of the Annual European Symposium on Programming (ESOP)*. 56–71.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the*

20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (San Diego, CA, USA) *(OOPSLA '05)*. ACM, 519–538.

Rezaul Alam Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures* (Munich, Germany). ACM, New York, NY, USA, 207–216.

A. Duran, J. Corbalan, and E. Ayguade. 2008. An adaptive cut-off for task parallelism. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

Derek L. Eager, John Zahorjan, and Edward D. Lazowska. 1989. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computing* 38, 3 (1989), 408–423.

Martin Elsman and Troels Henriksen. 2023. Parallelism in a Region Inference Context. *Proc. ACM Program. Lang.* 7, PLDI (2023), 884–906. doi:10.1145/3591256

Marc Feeley. 1992. A Message Passing Implementation of Lazy Task Creation. In *Parallel Symbolic Computing*. 94–107.

Marc Feeley. 1993a. *An efficient and general implementation of futures on large scale shared-memory multiprocessors*. Ph. D. Dissertation. Brandeis University, Waltham, MA, USA. UMI Order No. GAX93-22348.

Marc Feeley. 1993b. Polling Efficiently on Stock Hardware. In *Proceedings of the 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture*. Copenhagen, Denmark, 179–187.

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2011. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (2011), 1–40.

Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Manticore: A Heterogeneous Parallel Language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming* (Nice, France) *(DAMP '07)*. 37–44.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.

Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. 2020a. Compiler-Based Timing for Extremely Fine-Grain Preemptive Parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 53, 15 pages.

Souradip Ghosh, Michael Cuevas, Simone Campanoni, and Peter Dinda. 2020b. Compiler-Based Timing For Extremely Fine-Grain Preemptive Parallelism. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. doi:10.1109/SC41405.2020.00057

Seth Copen Goldstein, Klaus Erik Schauser, and David E Culler. 1996. Lazy threads: Implementing a fast parallel call. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 5–20.

Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. 81–93.

Robert H. Halstead, Jr. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (Austin, Texas, United States) *(LFP '84)*. ACM, 9–17.

Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. 2009. Backtracking-based load balancing. In *PPoPP '09* (Raleigh, NC, USA). ACM, 55–64.

Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. 1994. Using the Run-time Sizes of Data Structures to Guide Parallel-thread Creation. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming* (Orlando, Florida, USA) *(LFP '94)*. 79–90.

Shams Mahmood Imam and Vivek Sarkar. 2014. Habanero-Java library: a Java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14*. 75–86.

Shintaro Iwasaki and Kenjiro Taura. 2016. A static cut-off for task parallel programs. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 139–150.

Richard A. Kelsey. 1995. A correspondence between continuation passing style and static single assignment form. In *ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, California, United States). 13–22.

Vivek Kumar, Daniel Frampton, Stephen M. Blackburn, David Grove, and Olivier Tardieu. 2012. Work-stealing without the baggage. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 297–314. doi:10.1145/2384616.2384639

Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (San Francisco, California, USA) *(JAVA '00)*. 36–43.

I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-Fly Pipeline Parallelism. *TOPC* 2, 3 (2015), 17:1–17:42.

Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach. 2007. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*. 107–118.

Hans-Wolfgang Loidl and Kevin Hammond. 1995. On the granularity of divide-and-conquer parallelism. In *Proceedings of the 1995 Glasgow Workshop on Functional Programming*. 1–10.

P. Lopez, M. Hermenegildo, and S. Debray. 1996. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation* 21 (June 1996), 715–734. Issue 4-6.

Simon Marlow and Simon L. Peyton Jones. 2011. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, Hans-Juergen Boehm and David F. Bacon (Eds.). ACM, 21–32.

Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2013. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 2s (2013), 1–26.

MLton n.d.. MLton web site. http://www.mlton.org.

E. Mohr, D. A. Kranz, and R. H. Halstead. 1991. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems* 2, 3 (1991), 264–280.

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635.

Stefan Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Responsive Parallelism with Futures and State. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

Stefan K. Muller and Umut A. Acar. 2016. Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 71–82.

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2017. Responsive Parallel Computation: Bridging Competitive and Cooperative Threading. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 677–692.

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Types and Cost Models for Responsive Parallelism. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '18)*.

Stefan K. Muller, Kyle Singer, Devyn Terra Keeney, Andrew Neth, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2023. Responsive Parallelism with Synchronization. *Proc. ACM Program. Lang.* 7, PLDI (2023), 712–735.

Stefan K. Muller, Sam Westrick, and Umut A. Acar. 2019. Fairness in Responsive Parallelism. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP 2019)*.

OpenMP Architecture Review Board. [n. d.]. OpenMP Application Program Interface. http://www.openmp.org/

Joseph Pehoushek and Joseph Weening. 1990. Low-cost process creation and dynamic partitioning in Qlisp. In *Parallel Lisp: Languages and Systems*, Takayasu Ito and Robert Halstead (Eds.). Lecture Notes in Computer Science, Vol. 441. Springer Berlin / Heidelberg, 182–199.

Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS*. 383–414.

Manohar K Prabhu and Kunle Olukotun. 2005. Exposing speculative thread parallelism in SPEC2000. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 142–152.

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. ACM, New York, NY, USA, 392–406.

Mike Rainey. 2023. The best multicore-parallelization refactoring you've never heard of. arXiv:2307.10556 [cs.DC]

Mike Rainey, Ryan R. Newton, Kyle C. Hale, Nikos Hardavellas, Simone Campanoni, Peter A. Dinda, and Umut A. Acar. 2021. Task parallel assembly language for uncompromising parallelism. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1064–1079.

John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the 25th ACM National Conference*. 717–740.

Martin C. Rinard and Pedro C. Diniz. 1996. Commutativity Analysis: A New Analysis Framework for Parallelizing Compilers. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 54–67.

Radu Rugina and Martin Rinard. 1999. Automatic parallelization of divide and conquer algorithms. *ACM SIGPLAN Notices* 34, 8 (1999), 72–83.

Tao B. Schardl and I-Ting Angelina Lee. 2023. OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy (Eds.). ACM, 189–203. doi:10.1145/3572848.3577509

Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel*

Programming (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 249–265. doi:10.1145/3018743.3018758

Kish Shen, Vitor Santos Costa, and Andy King. 1999. Distance: A new metric for controlling granularity for parallel execution. *Journal of Functional and Logic Programming* 1999 (1999), 1–23.

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 113:1–113:30.

K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for standard ML. *Journal of Functional Programming* FirstView (6 2014), 1–62.

Daniel Spoonhower. 2009. *Scheduling Deterministic Parallel Programs*. Ph. D. Dissertation. Carnegie Mellon University. https://www.cs.cmu.edu/~rwh/theses/spoonhower.pdf

Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. 2009. Beyond Nested Parallelism: Tight Bounds on Work-stealing Overheads for Parallel Futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA '09)*. ACM, New York, NY, USA, 91–100.

Yian Su, Mike Rainey, Nicholas Wanninger, Nadharm Dhiantravan, Jasper Liang, Umut A. Acar, Peter Dinda, and Simone Campanoni. 2024. Compiling Loop-Based Nested Parallelism for Irregular Workloads. In *International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*.

Alexandros Tzannes. 2012. *Enhancing Productivity and Performance Portability of General-Purpose Parallel Programming*. Ph. D. Dissertation. University of Maryland.

Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. 2010. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP '10*. 179–190.

Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. 2014. Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism. *TOPLAS* 36, 3, Article 10 (Sept. 2014), 51 pages.

Stephen Weeks. 2006. Whole-program compilation in MLton. In *ML '06: Proceedings of the 2006 workshop on ML* (Portland, Oregon, USA). ACM, 1–1.

Joseph S. Weening. 1989. *Parallel Execution of Lisp Programs*. Ph. D. Dissertation. Stanford University. Computer Science Technical Report STAN-CS-89-1265.

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022a. Entanglement Detection With Near-Zero Cost. In *Proceedings of the 27th ACM SIGPLAN International Conference on Functional Programming (ICFP 2022)*.

Sam Westrick, Matthew Fluet, Mike Rainey, and Umut A. Acar. 2024. Automatic Parallelism Management. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages (POPL) (POPL '24)*.

Sam Westrick, Mike Rainey, Daniel Anderson, and Guy E. Blelloch. 2022b. Parallel block-delayed sequences. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 61–75. doi:10.1145/3503221.3508434

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)*.

A Typing Rules of Thread Pools

$$\frac{P \vdash \mathcal{R}_p \ \mathrm{WF}_{\mathrm{pool}} \quad P \vdash \mathcal{R}_c \ \mathrm{WF}_{\mathrm{pool}} \quad \textsc{prom}(\mathcal{R}_c) = \emptyset \quad \textsc{root}(\mathcal{R}_c) = g_{\mathrm{spwn}} \quad \textsc{prom}(\mathcal{R}_p) = \bar{\pi} \cdot g_{\mathrm{spwn}}(\bar{x})}{P \vdash \mathcal{R}_p \curlywedge \mathcal{R}_c \ \mathrm{WF}_{\mathrm{pool}}} \qquad \frac{P; \mathcal{K}; \mathcal{X} \vdash \rho \ \mathrm{WF}_{\mathrm{deque}} \quad P; f \vdash \mathcal{K} \ \mathrm{WF}_{\mathrm{stack}} \quad P; f; \mathcal{X}; \rho \vdash C \ \mathrm{WF}_{\mathrm{code}}}{P \vdash \mathcal{K} \cdot \langle f, \rho, \mathcal{X} \rangle \diamond C \ \mathrm{WF}_{\mathrm{pool}}}$$

$$\frac{P; g \vdash \mathcal{K} \ \mathrm{WF}_{\mathrm{stack}} \quad P; \mathcal{K}; f \vdash \langle g, \rho, \mathcal{X}, b_{\mathrm{ret}} \rangle \ \mathrm{WF}_{\mathrm{frame}}}{P; f \vdash \mathcal{K} \cdot \langle g, \rho, \mathcal{X}, b_{\mathrm{ret}} \rangle \ \mathrm{WF}_{\mathrm{stack}}} \qquad \frac{g \in P \quad \mathbf{fun}_r \, f(\_)\{\_\} \in P \quad P; \mathcal{K}; \mathcal{X} \vdash \rho \ \mathrm{WF}_{\mathrm{deque}} \quad g; \mathcal{X}; \rho \vdash b_{\mathrm{ret}}(\cdot^r) \ \mathrm{WF}_{\mathrm{cont}}}{P; \mathcal{K}; f \vdash \langle g, \rho, \mathcal{X}, b_{\mathrm{ret}} \rangle \ \mathrm{WF}_{\mathrm{frame}}}$$

$$\frac{\forall g(\bar{x}) \in \bar{\pi} \cup \bar{v}.\ P; \Gamma \vdash g(\bar{x}) \ \mathrm{WF}_{\mathrm{call}} \quad (\bar{\pi} = \emptyset \vee \textsc{unpr}(\mathcal{K}) = \emptyset)}{P; \mathcal{K}; \Gamma \vdash \bar{\pi} : \bar{v} \ \mathrm{WF}_{\mathrm{deque}}}$$

$$\frac{\textsc{root}(\mathcal{R}_p) = f}{\textsc{root}(\mathcal{R}_p \curlywedge \mathcal{R}_c) = f} \qquad \frac{\mathcal{K} = \langle f, \ldots \rangle \cdot \mathcal{K}'}{\textsc{root}(\mathcal{K} \diamond C) = f} \qquad \frac{\textsc{prom}(\mathcal{R}_p) = \bar{\pi} \cdot g_{\mathrm{prom}}(\bar{x})}{\textsc{prom}(\mathcal{R}_p \curlywedge \mathcal{R}_c) = \bar{\pi}} \qquad \frac{\textsc{prom}(K) = \bar{\pi}}{\textsc{prom}(\mathcal{K} \diamond C) = \bar{\pi}}$$

$$\frac{\textsc{prom}(\mathcal{K}) = \bar{\pi}'}{\textsc{prom}(\mathcal{K} \cdot \langle \_, \bar{\pi} : \_, \_, \_ \rangle) = \bar{\pi}' \cdot \bar{\pi}} \qquad \frac{}{\textsc{prom}(\emptyset) = \emptyset} \qquad \frac{\textsc{unpr}(\mathcal{K}) = \bar{v}'}{\textsc{unpr}(\mathcal{K} \cdot \langle \_, \_ : \bar{v}, \_, \_ \rangle) = \bar{v}' \cdot \bar{v}} \qquad \frac{}{\textsc{unpr}(\emptyset) = \emptyset}$$

Fig. 16. Rules for well-formed thread pools

B Comparing MPL^{sp}'s par to MPL^{s}

Table 4. Comparing our approach's (**MPL^{sp}**) automatically managed implementation of par to Westrick et al. [2024]'s MPL^{s}. For this comparison, both systems use the same code and same divide-and-conquer implementation of reduce in order to isolate the difference in par performance. To this end, we intentially do not use the faster reduce available to **MPL^{sp}**.

	T_1		Speedup	T_{80}		Speedup
Benchmark	MPL^{s}	**MPL^{sp}**	$\frac{T_1(\text{MPL}^s)}{T_1(\textbf{MPL}^{sp})}$	MPL^{s}	**MPL^{sp}**	$\frac{T_{80}(\text{MPL}^s)}{T_{80}(\textbf{MPL}^{sp})}$
bfs	6.01	6.17	**.973**	.159	.154	**1.03**
bignum-add	1.84	1.76	**1.05**	.029	.027	**1.06**
delaunay	7.88	7.99	**.987**	.399	.396	**1.01**
grep	5.99	4.91	**1.22**	.093	.073	**1.27**
line-fit	3.41	2.60	**1.31**	.057	.039	**1.47**
mandelbrot	3.89	4.16	**.936**	.056	.056	**.989**
map-heavy	3.41	4.25	**.802**	.045	.055	**.802**
map-light	7.40	4.48	**1.65**	.137	.110	**1.25**
merge-sort	6.15	6.02	**1.02**	.097	.094	**1.04**
nearest-nbrs	1.48	1.43	**1.04**	.029	.028	**1.03**
nqueens	2.32	2.20	**1.05**	.033	.030	**1.11**
primes	14.9	7.26	**2.06**	.263	.153	**1.72**
sparse-mxv	6.62	5.87	**1.13**	.095	.078	**1.22**
suffix-array	5.27	4.29	**1.23**	.104	.091	**1.14**
triangle-count	10.2	9.90	**1.03**	.153	.150	**1.02**
word-count	2.73	2.06	**1.33**	.041	.028	**1.50**
geomean			**1.14**			**1.15**