Spork: Automatic Parallelism Management for Loops

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 like the fork-join model and parallel loops allow programmers to express parallelism 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 these overheads. 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. These works, however, assume a binary fork-join model of parallelism and do not provide direct support for the dominant form of parallelism: parallel loops.

In this paper, we extend an intermediate representation (IR) with a pair of primitives called *spork* and *spoin*, enabling 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 prove soundness using the Lean theorem prover. We present encodings of both high-level parallel loops and fork-join parallel code in the IR using spork and spoin. Then, we show how to integrate heartbeat scheduling to determine when these primitives should go parallel and when they should stay sequential to guarantee efficiency. We implement our techniques in the MPL compiler for Parallel ML and conduct an experimental evaluation. The experiments show significant improvement over prior work on automatic parallelism management, offering as much as 6.1x improvement on 80 cores and bringing the overheads of automatic parallelism management relative to manually optimized to within reasonable levels (averaging 36% on 80 cores) across a wide range of benchmarks.

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

Programmers today can write parallel programs in much the same way as sequential ones by using parallel abstractions such as parallel loops and higher order functions (e.g., `map` and `reduce`). For example, a higher-order `map(A,f)` operation, which applies a function `f` across an array, can be implemented by using a sequential or a parallel loop over the array. In theory, the parallel implementation should only improve performance relative to the sequential one and scale well as the degree of parallelism or the number of cores increases. In practice, however, the parallel implementation may in fact run slower than the sequential one even with many cores.

This theory-practice gap can be formulated by the following question: *can programmers write parallel programs at a high level of abstraction and expect them to perform well in practice?* Decades of research have shown that answering this is not easy. The challenge is that parallelism is not free: any parallel implementation incurs overheads to spawn (create), schedule, and synchronize parallel tasks. A single "task spawn" operation (including the cost of manipulating a concurrent scheduler queue) can require hundreds of processor cycles or more, even with the fastest implementations on

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

```
(* run {f(i), ..., f(j-1)} in parallel *)
fun parfor_dc(i, j, f) =
  if j-i = 1 then f(i) else
  let val mid = i + (j-i) div 2
  in par(fn () => parfor_dc(i, mid, f),
         fn () => parfor_dc(mid, j, f))
  end
```

Fig. 1. Westrick et al. [2024]'s divide-and-conquer parallel for loop, using their automatically managed parallel primitive, par. It incurs "splitting costs": $j - i - 1$ midpoint calculations (in yellow) and $2(j - i - 1)$ non-tail calls (in red). Even if par is zero-cost, the splitting costs can dominate.

```
fun sequential_for(i, j, f) =
  if i >= j then () else
  (f(i); sequential_for(i+1, j, f))
```

Fig. 2. Simple sequential for loop: no midpoint calculations, no non-tail calls (can be compiled to a loop by tail-call optimization).

```
fun parfor(i, j, f) =
(* ... implemented in SporkIR ... *)
```

Fig. 3. This paper: we present a compilation strategy for parallel loops which automatically amortizes splitting costs and preserves all parallelism.

modern hardware [Ghosh et al. 2020]. This overhead is modest for a single task, but can accumulate—or, in the presence of fine-grained parallelism, even dominate—when tasks become too small in ways that are not statically predictable. For example, a loop may require only a handful of processor cycles per iteration; on any such loop, spawning a task for each iteration to run in parallel is doomed to be as much as an order of magnitude slower than the tight sequential loop.

Historically, we expected the programmer to control the cost-benefit ratio of parallelism by "coarsening" parallelism [Acar et al. 2019, 2018; Tzannes et al. 2014; Westrick et al. 2024]. To coarsen parallelism, the programmer combines small pieces of work into larger chunks and spawns only one task per chunk, thereby amortizing the cumulative overhead of parallelism. But determining how large the chunks should be has proved difficult (and, in some cases, impossible) to answer statically, since it can depend on dynamic data. For example, what should the coarsening criteria be for a polymorphic function `map(A,f)` which operates on an arbitrary array `A` with a statically unknown function (`f`)? 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 difficulty and limitations of coarsening, recent research proposed to automate the entire process by using *automatic parallelism management (APM)* [Westrick et al. 2024], which allows the compiler and runtime system to manage parallelism automatically by coarsening dynamically at run time. The appeal of automatic parallelism management is that it completely eliminates the burden of coarsening needed for controlling parallelism overheads. With it, the programmer can write code in a way that exposes all available parallelism and leave it to the compiler and runtime system to decide how to make the most of it. The authors show that by building on prior work on *heartbeat scheduling* [Acar et al. 2018; Rainey et al. 2021; Su et al. 2024], automatic parallelism management can work well both in theory and in practice.

Yet existing work on automatic parallelism management has a significant limitation: it assumes fork-join parallelism and performs efficiently only for recursive divide-and-conquer parallel programs. Specifically, existing work supports the parallel primitive `par`, which performs two function calls in parallel, and is therefore well-suited for recursive divide-and-conquer algorithms. They are then able to encode parallel for loops using a divide-and-conquer approach, as shown in Figure 1 with the higher-order function `parfor_dc`. This approach is fully parallel—it exposes all opportunities for parallelism—but does not always perform well relative to a simple sequential for loop like that shown in Figure 2. We have measured that in one of our benchmarks, for a tight loop body with only a handful of instructions per iteration, it incurs nearly 14x overhead even if `par` itself has no overhead. The problem is that `parfor_dc` has to perform $n - 1$ "splits" for a loop of size n, and every split incurs the modest overheads of a midpoint calculation and two non-tail calls. These overheads accumulate even if the `par` primitive itself is free.

Although Westrick et al. [2024]'s automatically managed `par` can successfully amortize the overheads of task creation, the fundamental problem is that it is **unable to amortize the overheads of loop splitting**. To implement a parallel loop with `par` alone, the programmer is forced to either manually coarsen (to amortize the overheads of loop splitting) or suffer these overheads. The `par` primitive is therefore insufficient to achieve automatic parallelism management for parallel loops.

In this paper, we consider the challenge of extending APM to efficiently support parallel loops. Our goal is to automatically manage the overheads of both task creation *and* loop splitting, allowing the programmer to freely use parallel loops without worrying about the overheads of parallelism.

We present Spork IR, an SSA-level intermediate representation that enables automatic parallelism management for parallel loops, parallel reductions (loops with aggregation), as well as fork-join parallel tasks. Roughly speaking, our approach is to compile parallel loops into a form that moves the overheads of splitting away from the "fast path". To do so, we introduce new SSA-level control-flow constructs which allow for loops and reductions to proceed sequentially by default, with nearly zero overhead, while still allowing for dynamic splits at any moment. These splits are then scheduled by the runtime system according to a heartbeat-based policy.

Our two new control-flow constructs are called ***spork*** (**s**equential or **p**arallel f**ork**) and ***spoin*** (**s**equential or **p**arallel j**oin**). spork and spoin always come in matching pairs and are used to delimit the loop body. At the beginning of each loop iteration, spork registers an alternative code path for a parallel task implicitly on the call stack. If the runtime system decides to split the loop, it does so by creating a proper task from the implicit representation; this is called a *promotion*. The corresponding spoin then checks whether or not promotion occurred, and if so, control flow at the spoin is dynamically redirected to perform a synchronization with the promoted task. Otherwise (if no promotion occurred during the execution of the loop body), control flow proceeds sequentially to the next loop iteration, and the implicit task is discarded. Similar to prior work, we leverage heartbeat scheduling to guarantee that the promotion costs are fully amortized.

The semantics of spork and spoin in Spork IR are somewhat permissive, but an efficient lowering to assembly code hinges on their use according to key invariants. For example, a crucial property of spork and spoin is that they allow for function inlining optimizations, which are critical for avoiding interprocedural overheads in tight loops. Function inlining can lead to nested spork-spoin pairs (e.g., in nested parallel loops) and these spork-spoin pairs have to remain well-nested during compilation in order to be lowered to assembly. We therefore formalize the operational semantics of Spork IR, define a set of well-formedness rules which both enforce safety and enable the efficient lowering described in Section 4, and prove key soundness theorems for Spork IR in the Lean theorem prover.

We demonstrate that Spork IR is practical by extending the MPL compiler [Acar et al. 2020], a whole-program optimizing compiler for a parallel dialect of Standard ML, which is also the setting for Westrick et al. [2024]'s prior work on APM. Our implementation is more general than this prior work, as it retains support for the divide-and-conquer primitive (`par`) while additionally supporting the new compilation strategy for parallel loops and reductions. We incorporate Spork IR and adapt existing optimizations and compilation passes appropriately.

We evaluate our implementation 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. These benchmarks have already been manually coarsened by experts, and have been shown to perform well in prior work, with reasonably low overheads in comparison to sequential baselines and scalability to high core counts [Arora et al. 2021, 2023].

For our evaluation, we take all manually coarsened loops in the Parallel ML Benchmark Suite and make them fully parallel by removing coarsening. We then compare the automatically managed, fully parallel versions of benchmarks against their original (manually coarsened) implementations. Our results show low overheads for automatic management across a wide range of core counts, with only 17% and 36% overhead on average on 1 and 80 cores, respectively. This is a significant improvement over prior work on APM. To demonstrate this, we compare against Westrick et al. [2024]'s implementation and measure as much as 12x improvement on 1 core and 6x on 80 cores, with an average improvement of approximately 2x across all benchmarks. These improvements are due primarily to successful amortization of the splitting overheads in parallel loops and the ability to eliminate interprocedural overheads by inlining loop bodies. Altogether, our results show that automatic parallelism management extended with parallel loops can enable programs written with high-level parallel constructs to achieve fast performance while managing thread creation, scheduling, and synchronization fully automatically, without any manual coarsening by the programmer. The specific contributions of the paper include:

- 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 well-formedness rules of Spork IR, along with a proof of soundness in the Lean theorem prover,
- a strategy for expressing parallel loops and fork-join parallelism using spork and spoin,
- an end-to-end implementation in the MPL compiler and runtime system, and
- an empirical evaluation of 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 coarsening 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*. In the rest of this section, we define the syntax and semantics of Spork IR, ascribe to it a set of well-formedness rules, and prove the semantics are sound.

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, while not strictly necessary for a coherent semantics, prove absolutely crucial later on for lowering to efficient assembly code.

2.1 Syntax

Figure 4 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 (usually omitted when nil), and a series of assignments terminated by a control flow transfer. In addition to functions, basic blocks may have parameters because static single assignment form requires every variable (hereafter called *temporary*) be assigned at 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., $x \leftarrow y + z$) store the value of an

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

Fig. 4. Syntax of Spork IR, new transfers highlighted: **spork** and **spoin** ("sequential/parallel fork, join").

expression in a temporary, and transfers enable control flow across 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 4. The $\textbf{spork}(b_{\text{body}} \parallel b_{\text{spwn}})$ transfer behaves as a $\textbf{goto}\ b_{\text{body}}$, but additionally opens a scope in which the code beginning at b_{spwn} 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 b_{spwn}, or 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 the values of $\bar{x}$ as arguments.

2.2 Operational Semantics

$$\begin{array}{llll}
\textit{Thread pool} & \mathcal{R} & ::= & \mathcal{T} \mid \mathcal{R}_p \curlywedge \mathcal{R}_c \\
\textit{Thread state} & \mathcal{T} & ::= & \mathcal{K} \diamond C \\
\textit{Call stack} & \mathcal{K} & ::= & \overline{k} \\
\textit{Stack frame} & k & ::= & \langle f, \sigma, \mathcal{X}, b_{\text{ret}}? \rangle \\
\textit{Spawn deque} & \sigma & ::= & \bar{\pi} \,\$\, \bar{\upsilon} \\
\textit{Spawn block} & \pi, \upsilon & ::= & b_{\text{spwn}} \\
\textit{Value map} & \mathcal{X}, \mathcal{Y} & \in & \textit{(temp)} \rightharpoonup \textit{(value)}
\end{array}$$

Fig. 5. Definitions for Spork IR operational semantics

Formalizing this notion, we present definitions for the operational semantics of Spork IR in Figure 5 and the semantics themselves in Figure 6. 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 $\mathcal{R}_c$.[1] 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 σ of spawn blocks, one for each spork scope we are inside (local to this stack frame, i.e., those entered while this was the active stack frame) with "\$" delimiting promoted ones $\bar{\pi}$ from unpromoted $\bar{\upsilon}$, (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 active stack frame.

We define the execution of Spork IR via the small-step operational semantics in Figure 6. 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 v in the active frame's value mapping $\mathcal{X}$.
- GOTO jumps to block b_{next}, assigning to its parameters $\bar{y}$ the values of the arguments $\bar{x}$.
- IF-THEN and IF-ELSE conditionally jump: b_{then} if e evaluates to nonzero, otherwise b_{else}.

[1]This structure aids the well-formedness rules; see Appendix A for more details.

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

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

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

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

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

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

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

$$\frac{\mathcal{K}\ \text{fully promoted (i.e.,}\ \forall\ \langle \_, \_\,\$\,\bar{v}, \_, \_\rangle \in \mathcal{K}.\ \bar{v} = \varnothing) \quad \textbf{block}\ b_{\text{spwn}}\colon C' \in f}{\mathcal{K}\cdot\big\langle f, \bar{\pi}\,\$\,b_{\text{spwn}}\cdot\bar{v}, \mathcal{X}, b_{\text{ret}}?\big\rangle\cdot\mathcal{K}' \diamond C \mapsto \mathcal{K}\cdot\big\langle f, \bar{\pi}\cdot b_{\text{spwn}}\,\$\,\bar{v}, \mathcal{X}, b_{\text{ret}}?\big\rangle\cdot\mathcal{K}' \diamond C \curlywedge \langle f, \varnothing, \mathcal{X}\rangle \diamond C'}\,\textsc{promote}$$

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

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

Fig. 6. Spork IR small-step operational semantics, highlighting changes in state.

- **CALL** stores b_{ret} as the return address, pushes a new stack frame onto the call stack, and initializes it by mapping from function parameters $\bar{y}$ to the values of the arguments $\bar{x}$.
- **RETURN**, conversely, pops the active stack frame and returns to the caller's, passing the values of the returned arguments $\bar{y}$ as arguments to b_{ret}.
- **SPORK** always jumps to b_{body}, but additionally pushes b_{spwn} to the end of the active frame's spawn deque $\bar{\pi}\,\$\,\bar{v}$, allowing it to be promoted to a thread later.
- **PROMOTE** is nondeterministic and may happen at any time when at least one stack frame has unpromoted spawn(s). It finds the oldest stack frame with an unpromoted spawn block b_{spwn} by ensuring all frames before it ($\mathcal{K}$) are entirely promoted, marks b_{spwn} as promoted, and creates a new thread executing that block. **This ensures spawn blocks are promoted in order of oldest to newest**.
- **SPOIN-PROM** happens at a spoin when its associated spork was promoted, popping its spawn block b_{spwn} from the end of the (fully promoted) spawn deque. No unpromoted spawns remain, since promotions happen oldest-first and, conversely, spoin closes the most recent spork scope. Once the child thread ends with a **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 b_{spwn} from being promoted in the future), then jumps to the b_{unpr} block.

2.3 Well-Formedness

Spork IR enforces several constraints in order to facilitate the lowering of code to efficient assembly. Chiefly, these constraints are that sporks and spoins come in matching pairs, with no other control flow into, or out of, the code between, and that functions can only return after all sporks have been closed by a spoin. This first restriction results in every basic block having a statically inferable spork nesting, and the second ensures that any potentially necessary synchronization local to a stack frame has happened before destroying that frame. These properties prove absolutely crucial for lowering Spork IR to efficient assembly code, as they impose a static bound on the size of the spawn deque in every function and allow the compiler to know what the (local) spawn deque is for every program point. For precise well-formedness rules and a detailed discussion, see Appendix A.

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

Theorem 2.1 (Soundness). *For a well-formed program with* **fun** main(){**block** entry: $C, \ldots$} *and thread pool* $\mathcal{R}$ *such that* $\langle \text{main}, \emptyset, \emptyset, \text{entry} \rangle \diamond C \mapsto^* \mathcal{R}$*, either*

(1) $\mathcal{R} = \langle \text{main}, \emptyset, \mathcal{X}, b \rangle \diamond \mathbf{return}(\bar{x})$ is a final leaf thread returning from its last stack frame, or
(2) $\mathcal{R} \mapsto \mathcal{R}'$ for some thread pool $\mathcal{R}'$

Theorem 2.2 (Inlining). *Inlining any function call preserves well-formedness.*

Theorem 2.1 (Soundness) demonstrates the safety of Spork IR's operational semantics, 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.

Function call inlining in Spork IR can be performed in the typical way. For example, $\mathbf{call}\ f(\bar{x}) \triangleright b_{\text{ret}}$ would be inlined by adding the basic blocks of f to the current function's, changing the call transfer to $\mathbf{goto}\ \text{entry}_f(\bar{x})$, and replacing each $\mathbf{return}(\bar{y})$ in the newly added blocks with $\mathbf{goto}\ b_{\text{ret}}(\bar{y})$ (except those returns on the spawn side of a spork). Theorem 2.2 (inlining) provides a certain sense of composability, whereby well-formed code may be nested inside other well-formed code and remain well-formed. This is important in the following section, as it guarantees our encoding of parallel loops can be arbitrarily nested *without needing any change to its use of parallelism primitives.*

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`.[2] Therefore, 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

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

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 "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 $\textsc{par}_{f,g}()$. Once lowered to Spork IR, we introduce definitions for each of these PAR functions as shown in Figure 7.

$\textsc{par}_{f,g}()$ 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 which begins running the *spwn* block in parallel, calling $g()$. When the original thread finishes evaluating $f()$, it returns to execution at *check* with x storing the result of $f()$. *check* spoins, checking if the promotion occurred; if so, another thread already began running $g()$, so it synchronizes with that thread by waiting for it to finish (if it is still running) and then jumping to *join* with y_1 storing the result of that thread (i.e., the result of $g()$). If the spork was not unpromoted, spoin jumps to *evalG*. In this case it still needs to execute $g()$, doing so in the same thread and then returning the two results.

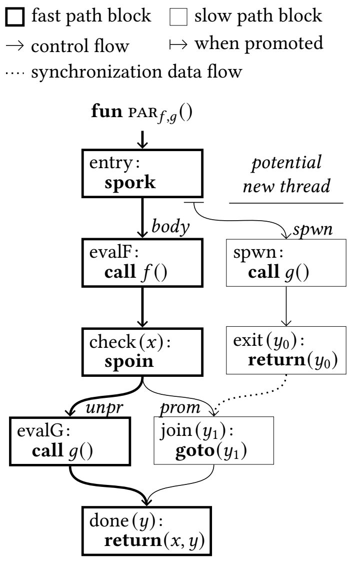


Fig. 7. Implementing `par` in Spork IR for a particular f, g.

The common case of $\textsc{par}_{f,g}()$ 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 $\textsc{par}_{f,g}()$ 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 $\textsc{par}_{f,g}()$ 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 work-first principle of [Frigo et al. 1998]: because the common case is sequential, optimizing for it results in each parallel thread performing its own (sequential) work more efficiently, resulting in a faster program even in a parallel setting.

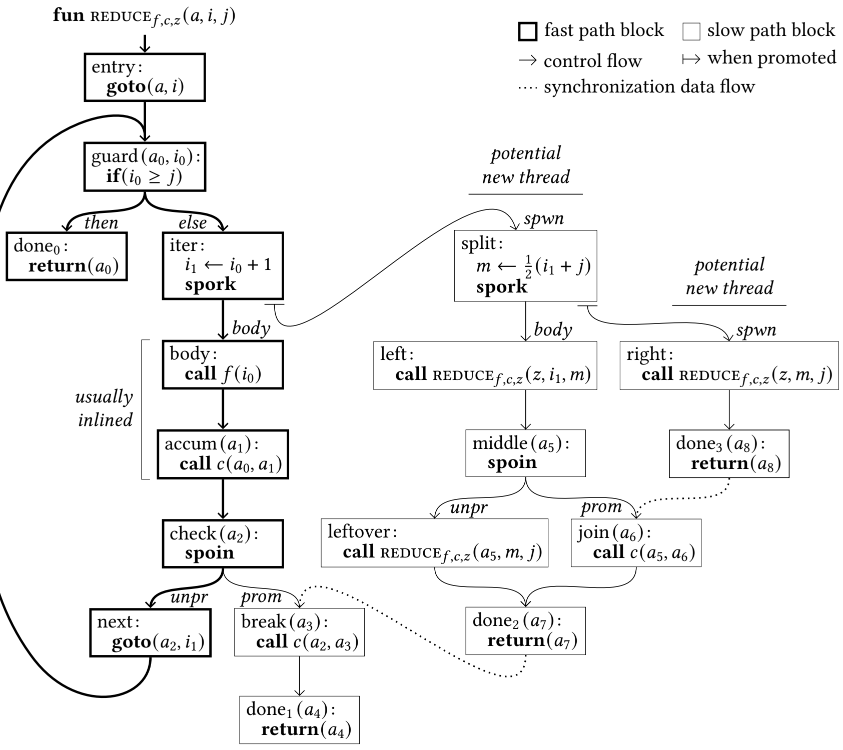

fun REDUCE$_{f,c,z}(a, i, j)$
fast path block
slow path block
→ control flow
↦ when promoted
···· synchronization data flow
entry: goto(a, i)
guard (a_0, i_0): if$(i_0 \geq j)$
then
else
done$_0$: return(a_0)
iter: $i_1 \leftarrow i_0 + 1$ spork
body
body: call $f(i_0)$
usually inlined
accum (a_1): call $c(a_0, a_1)$
check (a_2): spoin
unpr
prom
next: goto(a_2, i_1)
break (a_3): call $c(a_2, a_3)$
done$_1$ (a_4): return(a_4)
potential new thread
spwn
split: $m \leftarrow \frac{1}{2}(i_1 + j)$ spork
body
left: call REDUCE$_{f,c,z}(z, i_1, m)$
potential new thread
spwn
right: call REDUCE$_{f,c,z}(z, m, j)$
middle (a_5): spoin
done$_3$ (a_8): return(a_8)
unpr
prom
leftover: call REDUCE$_{f,c,z}(a_5, m, j)$
join (a_6): call $c(a_5, a_6)$
done$_2$ (a_7): return(a_7)

Fig. 8. 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 a specialized first-order function REDUCE$_{f,c,z}(a, i, j)$ unique to that call. The calls to f and c on the fast path may (and often will) be inlined.

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

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 8. 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 complete (that is, $i \geq j$). 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 during the loop body and this is the oldest unpromoted spork, it creates a new thread which begins running at *split*. The original thread continues executing the loop body and, when finished, spoins: since the spork's potential parallelism was promoted, it waits for the newly spawned thread to finish, receiving its returned result in the *break* argument a_3. It calls the combine operator c with the 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 *split*, it aptly splits the remaining range in half and executes each half potentially in parallel, in a very similar way to PAR. 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`s. 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$^{\mathbf{sp}}$: A Compiler for Automatically Managed Parallel Loops

We present **MPL$^{\mathbf{sp}}$**, a compiler that produces efficient parallel loops by implementing Spork IR. It employs a runtime system that uses heartbeat scheduling to schedule promotions and work-stealing scheduling to load-balance promoted tasks. **MPL$^{\mathbf{sp}}$** is derived from MPL, which itself extends MLton [MLton nd; Weeks 2006], a whole-program optimizing compiler for Standard ML (SML), 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*, an 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 it 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$^{\mathbf{sp}}$**. In Section 4.1, we describe how Spork IR is extended with explicit handling of join tokens and how their use 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), 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. 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 and spoin; we wrap up this section by describing these interactions in Section 4.4, respectively.

4.1 Join Tokens and the Separation of Compiler and Scheduler

Implementing the semantics of Spork IR in **MPL$^{\mathbf{sp}}$** 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 9), where the compiler only "knows" about spork and spoin transfers. The parent and child synchronize explicity via a *join token*[3] that is created at the time of a promotion. This 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. As in [Westrick et al. 2024], SETJOIN and GETJOIN are functions implemented in source SML code with calls to MPL runtime system functions. The fast path through the b_{body} and b_{unpr} blocks is unchanged.

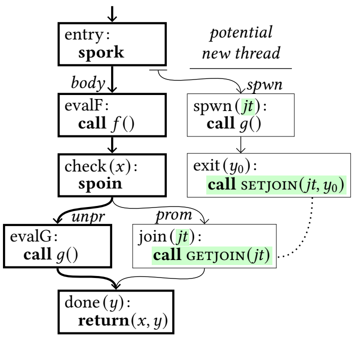


Fig. 9. Revised implementation of par, differences highlighted in green.

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

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

$\textbf{spork}\langle k\rangle(b_{\text{body}} \,\|\, b_{\text{spwn}}) \Rightarrow$ `jmp` b_{body} // †

$\textbf{block}\ b_{\text{spwn}}(jt)$: $\Rightarrow$ b_{spwn}:
...
(caller-side returning code)
`ld fp[`k`],` jt
`st NULL, fp[`k`]`
...

$\textbf{spoin}\langle k\rangle(b_{\text{unpr}}, b_{\text{prom}}) \Rightarrow$ `ld fp[`k`],` jt
`cmp` jt`, NULL`
`jne` b_{prom}
`jmp` b_{unpr} // †

$\textbf{block}\ b_{\text{prom}}(jt)$: $\Rightarrow$ b_{prom}:
...
`st NULL, fp[`k`]`
...

Fig. 10. Lowering of **spork** and **spoin**. Marked `jmp` instructions are likely eliminated by basic-block ordering.

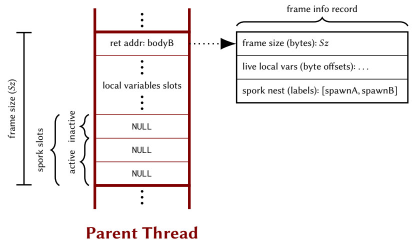
be entered via the b_{body} of the spork and exited via the matching spoin. Proper nesting means that, for any distinct pair of sporks in a function, their scopes are either disjoint or one is a proper subset of the other. This property is asserted for Spork IR by the well-formedness rules discussed in Section 2.3 and Appendix A.

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. Note that each spork scope occurs at the same index $\langle k\rangle$ in each spork nesting of which it is a member; this index can be associated with the scope's delimiting spork and spoin.

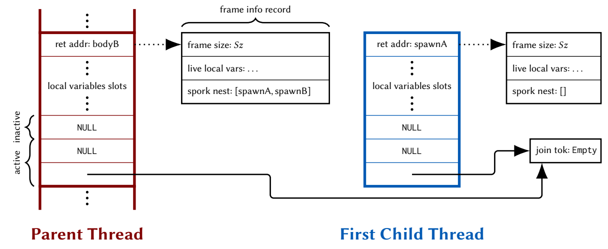
Lowering spork and spoin. Using these observations, we can give a realization of the spawn deque and implementations of spork and spoin transfers and of promotion. During lowering, when the call stack is made explicit, the back-end reserves *spork slots*: a contiguous sequence of N slots at the bottom 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*. 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 10 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 at index $\langle k\rangle$ is lowered to a sequence that reads from its spork slot by loading from an offset of the frame pointer `fp`, compares the read value with `NULL`, conditionally branches 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, transitioning from promoted active to inactive. The b_{prom} block argument jt is the non-`NULL` value already 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 compiler is agnostic to the meaning of the non-`NULL` value.

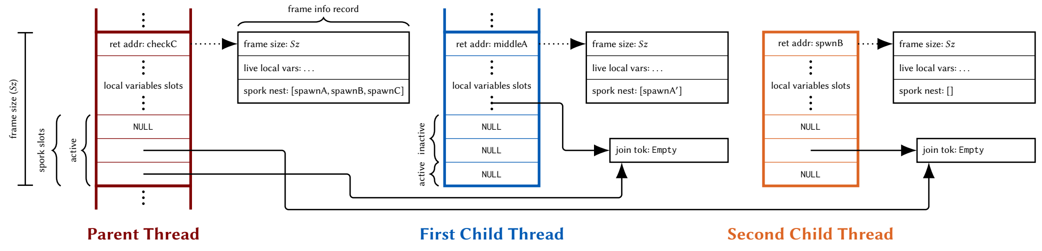
Note that these lowerings yield an extremely efficient fast (sequential) path: a spork performs only a jump and its matching spoin performs only a read, a comparison, a (failing) conditional branch, and a jump; moreover, the jumps will typically be eliminated by basic-block ordering.



(a) Before first promotion; parent thread is executing a **call** that returns to bodyB.



(b) After first promotion; note parent is unchanged except the first spork slot now points to a freshly allocated join token.



(c) After second promotion; parent thread has proceeded and is now executing a **call** that returns to checkC; first child thread has started and is now executing a **call** that returns to middleA. Note the state of the first child thread has no influence on the second promotion.

Fig. 11. Call stacks of parent and child threads during promotions. Note that stacks grows upwards.

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 to stack-top 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 it is reaches stack-bottom. **MPL**$^{\mathbf{sp}}$, 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 corresponding to the return address.

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 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 11 illustrates the effects of two promotions on a parent thread.

The lowering of the **block** $b_{\text{spwn}}(jt)$ of a spork transfer is handled specially (see Figure 10). It is treated as the return block of a void function call and, after performing the caller-side of the returning convention (e.g., adjusting the frame pointer `fp`), jt is assigned the value of the spork slot `fp[`k`]`. Then, `NULL` is written to that slot to inactivate it—since b_{spwn} is not inside the body of any spork scope—and execution continues. For example, in Figure 11, when the first child thread starts executing, the join token passed via a spork slot (Figure 11b) is moved to a local variable slot (Figure 11c), 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 SML is a higher-order language, the non-trivial control-flow of spork and spoin is easily exposed as a higher-order primitive. The earliest phase of the compiler that required changes was the closure-conversion phase, 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

$$\texttt{prim\_spork\_spoin}\,(f_{\text{body}} : \mathsf{unit} \to \alpha, f_{\text{spwn}} : \delta \to \zeta, f_{\text{unpr}} : \alpha \to \gamma, f_{\text{prom}} : \alpha \times \delta \to \gamma, \mathit{tag} : \mathsf{int}) : \gamma$$

The f_{body}, f_{spwn}, f_{unpr}, and f_{prom} functions correspond to the code for the homonymous edges of the introduced spork and spoin. 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 *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 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 matching spoin. Importantly, this translation of `prim_spork_spoin` guarantees 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 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, spending one if possible to immediately promote it; we handle this aspect in Section 4.4. 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).

```
fun promote (thr: Thread.t) = Scheduler.enqueue (runtime_promote (thr, newJoinToken ()))
fun tryEagerPromote () = if tryConsumeHBToken () then promote (Thread.current ()) else ()
fun setJoin (jt: β jointok, b: β) = ... (* store result b in jt, sync with parent, end thread *)
fun getJoin (jt: β jointok): β = ... (* synchronize with child, then retrieve result from jt *)

fun spork_spoin (body: unit -> α, spwn: unit -> β, unpr: α -> γ, prom: α*β -> γ, tag: int): γ =
  let fun body' () = (tryEagerPromote (); body ())
      fun spwn' jt = let val a = spwn () in setJoin (jt, a) end
      fun prom' (a, jt) = let val b = if Scheduler.tryDequeue jt then spwn () else getJoin jt
                          in prom (a, b) end
  in prim_spork_spoin (body', spwn', unpr, prom', tag) end
```

Fig. 12. spork_spoin function wrapping a use of the prim_spork_spoin primitive in source SML.

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$^{\mathbf{sp}}$** 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 heartbeat 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 heartbeat tokens to the child or give all of them. The sporks in PAR and in the *split* block of REDUCE use the first policy (see Figures 7 and 8), since the body and the (potential) child thread are typically of comparable work, while the *iter* block of REDUCE uses the second policy, since the remaining loop iterations are expected to be significantly more work than the one current loop iteration.

To execute threads on processors, **MPL$^{\mathbf{sp}}$** 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. On the promoted side of a spoin, we first check whether or not the child was stolen by attempting to pop from the back of the scheduler deque.[4] If it was, then full synchronization using GETJOIN and the corresponding SETJOIN must occur to obtain a value from the child. But if not, rather than wait for another thread to steal and execute the child, the current thread regains the child's heartbeat tokens, executes the child's code without any synchronization, and then jumps to the b_{prom} block; this is similar to the clone optimization of Frigo et al. [1998]. This avoids synchronization overheads when all workers are busy, as it no longer needs to synchronize via GETJOIN and SETJOIN.

4.4 Integration via Source SML Code

We finish the implementation of the Spork IR semantics with the wrapper function spork_spoin shown in Figure 12, which performs the necessary integration with the synchronization, parallelism management, and work-stealing components around a use of prim_spork_spoin. It ensures that the f_{body} function seen by the primitive immediately triggers a promotion if the current thread has accumulated any tokens (Section 4.3), that the f_{spwn} function ends with a SETJOIN (Section 4.1), and that the f_{prom} function checks the scheduler deque and either executes the child's code itself or synchronizes with a GETJOIN (Section 4.3). Compared to the fast path described in Section 4.2.1, this adds only the check for eager promotion: a read of the current thread's tokens (stored as thread-local metadata), a comparison, and a (failing) conditional branch.

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

Using `spork_spoin`, we implement `par` and `reduce` entirely in source SML code. The combination of monomorphisation, defunctionalization, inlining, and SSA IR optimizations specializes each use of `par` and `reduce` to their call-sites, yielding the control-flow graphs from Figures 7 and 8.

5 Evaluation

The goal of automatic parallelism management is to eliminate the burden of manually tuning parallelism grains, automatically achieving good performance across any number of cores. In this paper, we aim to additionally amortize the cost of splitting a parallel loop by moving it off the fast, sequential path. We evaluate the effectiveness of our approach with **MPL$^{\text{sp}}$** towards this goal on a benchmark suite of parallel programs. To better isolate the performance of our techniques vs. existing approaches, we directly compare against other SML compilers likewise based on MLton. In our evaluation, we study three parts:

(1) In Section 5.2, we compare our approach against prior work on automatic parallelism management [Westrick et al. 2024], showing up to 12.1x improvement on a single core and 6.10x on 80 cores and averaging 2.09x and 1.74x faster, respectively. The improvement is due to explicit support for parallel loops and guaranteed amortization of loop splitting costs while preserving all parallelism.
(2) In Section 5.3, we show that **MPL$^{\text{sp}}$** automatically amortizes the cost of splitting loops, reducing the number of splits in one benchmark by over four orders of magnitude.
(3) In Section 5.4, we show that our technique achieves low overheads on a single core relative to sequential elision, averaging 1.67x slower. At the same time, it maintains good parallel scalability, delivering 46.1x self-speedup on 80 cores (a 27.6x speedup over sequential).

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 22.04.5 LTS and Linux kernel version 5.15.0-144-generic. We use MLton version 20241230. Benchmark timings are evaluated with 5 warmup runs and then by averaging the next 20 runs. For more stable results, we disable hyperthreading and pin experiments to fixed 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. Many of these implementations make use of parallel loops with explicit, manually tuned grains. These grains were tuned by experts. For our evaluation, we remove these grains, replacing the coarsened loops with our spork-based loops. In all our experiments, the code for the benchmarks is identical across systems except for the implementation strategy of `par` and `reduce` and the presence (or absence) of manually tuned parallelism grains throughout.

5.2 MPL$^{\text{sp}}$ Achieves Low Overheads, Improving Over Westrick et al. [2024]

To evaluate our approach (**MPL$^{\text{sp}}$**), we compare against Westrick et al. [2024]'s implementation of automatic parallelism management, which we call MPL$^{\text{pc}}$, and a manually tuned baseline with call-site specific parallelism grains [Arora et al. 2023]. MPL$^{\text{pc}}$ only supports the `par` primitive, which requires all parallelism to be expressed in a divide-and-conquer style. Specifically, it compiles `par` into an SSA-level control-flow primitive called PCall (hence, "MPL$^{\text{pc}}$") which couples the creation of a potentially parallel task with a non-tail call. This is well-suited for recursive divide-and-conquer parallel algorithms, where recursive (non-tail) calls coincide with opportunities for parallelism.

Table 1. Overheads of automatic parallelism management using our approach ($\mathbf{MPL^{sp}}$) vs. those of $\mathrm{MPL^{pc}}$ (lower is better); both are relative to the same manually tuned baseline, which relies on call site-specific parallelism grains. Improvement factors of $\mathbf{MPL^{sp}}$ over $\mathrm{MPL^{pc}}$ (higher is better). T_1 and T_{80} report single-core and 80-core times, measured in seconds.

	T_1			*Overhead*		*Improvement*	T_{80}			*Overhead*		*Improvement*
Benchmark	Baseline	$\mathbf{MPL^{sp}}$	$\mathrm{MPL^{pc}}$	$\frac{\mathbf{MPL^{sp}}}{\text{Baseline}}$	$\frac{\mathrm{MPL^{pc}}}{\text{Baseline}}$	$\frac{\mathrm{MPL^{pc}}}{\mathbf{MPL^{sp}}}$	Baseline	$\mathbf{MPL^{sp}}$	$\mathrm{MPL^{pc}}$	$\frac{\mathbf{MPL^{sp}}}{\text{Baseline}}$	$\frac{\mathrm{MPL^{pc}}}{\text{Baseline}}$	$\frac{\mathrm{MPL^{pc}}}{\mathbf{MPL^{sp}}}$
bfs	2.99	3.09	5.99	**1.03**	**2.00**	**1.94**	.070	.094	.163	**1.34**	**2.33**	**1.75**
bignum-add	.723	.780	1.83	**1.08**	**2.53**	**2.34**	.010	.014	.029	**1.39**	**2.88**	**2.07**
delaunay	7.20	7.56	7.91	**1.05**	**1.10**	**1.05**	.225	.476	.417	**2.12**	**1.85**	**.876**
grep	2.37	2.41	5.57	**1.02**	**2.35**	**2.31**	.034	.044	.094	**1.30**	**2.79**	**2.14**
line-fit	.385	1.28	3.42	**3.32**	**8.89**	**2.68**	.020	.035	.060	**1.80**	**3.03**	**1.68**
mandelbrot	1.85	2.62	3.89	**1.42**	**2.10**	**1.48**	.024	.039	.058	**1.65**	**2.43**	**1.48**
map-heavy	4.22	4.22	3.41	**1.00**	**.809**	**.809**	.054	.055	.045	**1.03**	**.834**	**.808**
map-light	1.05	.987	12.0	**.939**	**11.4**	**12.1**	.031	.033	.203	**1.06**	**6.46**	**6.10**
merge-sort	4.47	5.67	6.44	**1.27**	**1.44**	**1.14**	.063	.089	.103	**1.41**	**1.62**	**1.15**
nearest-nbrs	1.32	1.35	1.48	**1.02**	**1.13**	**1.10**	.022	.027	.029	**1.22**	**1.33**	**1.09**
nqueens	1.61	1.48	2.31	**.916**	**1.43**	**1.56**	.022	.022	.035	**1.01**	**1.56**	**1.54**
primes	1.91	2.10	14.8	**1.10**	**7.75**	**7.05**	.052	.059	.258	**1.13**	**4.92**	**4.36**
sparse-mxv	1.73	1.75	6.62	**1.01**	**3.83**	**3.79**	.033	.036	.098	**1.10**	**2.97**	**2.69**
suffix-array	5.46	2.83	5.36	**.518**	**.982**	**1.90**	.093	.063	.109	**.677**	**1.17**	**1.73**
triangle-count	4.22	8.97	10.3	**2.12**	**2.43**	**1.14**	.063	.150	.163	**2.39**	**2.60**	**1.09**
word-count	.727	1.11	2.96	**1.52**	**4.07**	**2.67**	.010	.022	.044	**2.29**	**4.64**	**2.03**
min				**.518**	**.809**	**.809**				**.677**	**.834**	**.808**
geomean				**1.17**	**2.43**	**2.09**				**1.36**	**2.36**	**1.74**
max				**3.32**	**11.4**	**12.1**				**2.39**	**6.46**	**6.10**

Intuitively, PCall is somewhat similar to spork and spoin, but it conflates the creation of potentially parallel tasks with the creation of a stack frame, therefore incurring unavoidable overhead. Spork IR decouples these two concerns: the management of potentially parallel tasks is governed by spork and spoin, and the management of stack frames is governed by standard call and return instructions. With function inlining possible between spork and spoin, interprocedural overheads can be completely avoided, often yielding significant improvements.

To compare against their implementation, we replace all parallel loops and reductions with the appropriate divide-and-conquer code: parallel for loops use exactly the encoding shown in Figure 1, and the encoding of reductions is very similar. In this way, we have three versions of every benchmark: (1) the original, manually coarsened implementations as a baseline, (2) our implementation, labeled $\mathbf{MPL^{sp}}$, where every loop and reduction is replaced with the fully parallel spork-based encoding shown in Figure 8, and (3) Westrick et al. [2024]'s $\mathrm{MPL^{pc}}$, where every loop and reduction is replaced with the fully parallel divide-and-conquer encoding.

The results of this comparison are shown in Table 1. We find that our approach with $\mathbf{MPL^{sp}}$ significantly lowers overheads relative to the manually tuned baseline compared to $\mathrm{MPL^{pc}}$, improving on all core counts for 14 of the 16 benchmarks. On average, we improve over $\mathrm{MPL^{pc}}$ by reducing its +143% overhead to our +17% overhead on a single core, and from +136% to +36% on 80 cores (we found that most of this improvement comes from our novel encoding of `reduce` outperforming divide-and-conquer; see Appendix B for details). The benchmarks *map-light* and *primes* improve substantially over $\mathrm{MPL^{pc}}$, getting 12x and 7x faster, respectively. Both make use of tight loops, demonstrating this advantage; we examine *map-light* in more detail in Section 5.3.

In the case of *map-heavy*, we match the baseline within 3% (we suspect both $\mathbf{MPL^{sp}}$ and the baseline miss an optimization opportunity that $\mathrm{MPL^{pc}}$ finds). The *delaunay* benchmark is challenging because it has little theoretical parallelism. Figure 14 shows that *delaunay* parallelizes less than any of the other benchmarks, gaining only 15.9x 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 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 begin 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 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* and 80 cores, **MPL$^{\text{sp}}$**'s reduce is only 12% slower than divide-and-conquer.

5.3 MPL$^{\text{sp}}$ Automatically Amortizes the Costs of Splitting Loops

Overall, we observe significant improvements in comparison to the divide-and-conquer approach implemented in MPL$^{\text{pc}}$, especially for loop-heavy code. To better understand these improvements, we consider map-light in more detail, the benchmark where we observe our biggest improvements: 12.1x faster on 1 core and 6.1x on 80 cores. From the perspective of automatic parallelism management, this benchmark is a difficult case in the sense that it stresses the nearly zero-cost task representation. map-light consists of two back-to-back parallel loops of 2×10^8 iterations each (for a total of 4×10^8), both of which have loop bodies consisting of just a few instructions: iteration i performs a handful of arithmetic instructions and one memory write to an array at index i. Per iteration of such a loop, MPL$^{\text{pc}}$ requires at least one PCall, incurring the cost of one non-tail function call. Westrick et al. [2024]'s implementation therefore performs 4×10^8 non-tail calls to execute both loops in the benchmark. Concretely, the cost of one non-tail call is the cost of pushing and popping one stack frame; the 12x slowdown in Westrick et al. [2024]'s approach is almost entirely due to the overhead of pushing and popping these frames.

To confirm this, we implemented a sequential version of map-light using the divide-and-conquer approach for loops, but with this procedure's recursive calls executed in sequence instead of in parallel. We compiled this code using MLton for the best sequential performance and found that while it is faster than the single-core performance of MPL$^{\text{pc}}$ (4.7s vs. 12.0s), it is still *14x* slower than when the same benchmark uses an iterative, sequential loop (0.34s).

In contrast, our approach is capable of inlining the loop body between spork and spoin, resulting in no function calls except for at (infrequent) promotions; in this particular benchmark, we measure 1.7×10^4 promotions on average, reducing the number of splits (and therefore function calls) by four orders of magnitude. In other words, our approach automatically amortizes the cost of splitting loops by doing so only at promotions, remaining in fast, intraprocedural control flow otherwise.

5.4 MPL$^{\text{sp}}$ Has Low Sequential Overhead and Good Parallel Scalability

We evaluate each parallel **MPL$^{\text{sp}}$** 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 those shown in Figure 13 and compiled with MLton.

Table 2 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$^{\text{sp}}$** instead of purely sequential code even when only one core is available, with an average of 1.67x overhead. In 13 of the 16 benchmarks, **MPL$^{\text{sp}}$** has less than 2x overhead. **MPL$^{\text{sp}}$** also maintains good parallel scalability, with 27.6x average speedup in comparison to sequential on 80 cores. In Figure 14, we also plot the self-speedup of **MPL$^{\text{sp}}$** across a variety of core counts and

```
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. 13. Sequential implementations of par and reduce, compiled with MLton.

Table 2. 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	2.36	3.09	.094	**1.31**	**25.3**
bignum-add	.405	.780	.014	**1.92**	**28.5**
delaunay	4.89	7.56	.476	**1.55**	**10.3**
grep	1.65	2.41	.044	**1.46**	**37.7**
line-fit	.327	1.28	.035	**3.91**	**9.22**
mandelbrot	1.79	2.62	.039	**1.47**	**45.7**
map-heavy	3.68	4.22	.055	**1.15**	**66.4**
map-light	.343	.987	.033	**2.88**	**10.3**
merge-sort	4.02	5.67	.089	**1.41**	**44.9**
nearest-nbrs	.965	1.35	.027	**1.39**	**36.0**
nqueens	1.15	1.48	.022	**1.29**	**50.9**
primes	1.23	2.10	.059	**1.70**	**20.9**
sparse-mxv	1.05	1.75	.036	**1.67**	**28.7**
suffix-array	2.27	2.83	.063	**1.25**	**36.1**
triangle-count	5.27	8.97	.150	**1.70**	**35.1**
word-count	.488	1.11	.022	**2.27**	**22.3**
min				**1.15**	**9.22**
geomean				**1.67**	**27.6**
max				**3.91**	**66.4**

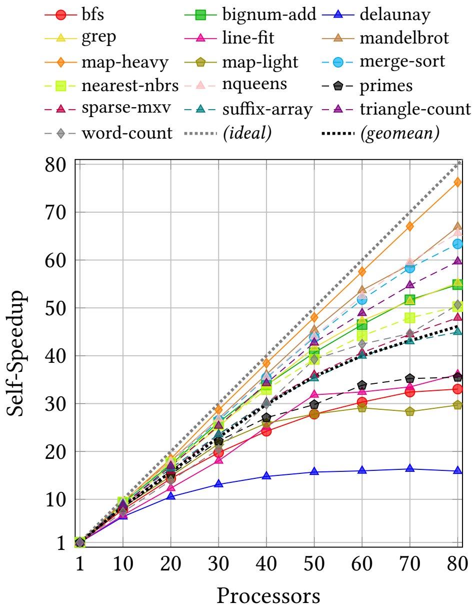


Fig. 14. Self-speedup of **MPL^{sp}** at different processor counts.

observe generally that performance improves as the number of cores increases, with 46.1x average self-speedup on 80 cores relative to **MPL^{sp}**'s single-core time T_1. These results demonstrate that our approach maintains high scalability without any manual tuning or chunking of parallel loops.

6 Related Work

The most closely related work is Westrick et al. [2024]'s *automatic parallelism management* (APM), which we discuss in Sections 1 and 5. APM deeply integrates a technique called *heartbeat scheduling* [Acar et al. 2018] into the compiler and runtime system of a high-level language to provably amortize all overheads of parallelism automatically and without any manual coarsening by the programmer. We extend APM to support parallel loops, significantly improving its efficiency.

MaPLe. 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].

Heartbeat Scheduling. Acar et al. [2018] proposed *heartbeat scheduling*, where tasks are created lazily according to a regular periodic pulse (a "heartbeat"). 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. The proof of efficiency critically relies upon an **oldest-first** spawning policy an every heartbeat.

Implementing heartbeat scheduling requires a low-level mechanism to respond to heartbeats in a timely manner. Our approach uses software polling [Basu et al. 2021; Feeley 1993b; Ghosh et al. 2020], specifically relying upon standard signal handling mechanisms, to redirect control-flow at compiler-inserted polling points. Rainey et al. [2021] refer to these points as *promotion-ready program points* (**prppt**s), and design a custom assembly language, TPAL, where basic blocks can be annotated as **prppt**s to register alternative code paths for promotions. TPAL expresses intraprocedural promotion by redirecting control-flow at **prppt**s to custom promotion handlers specialized to each **prppt**. However, TPAL only considered intraprocedural control-flow and has no explicit support for a call stack, leaving open how to implement an oldest-first promotion policy across nested function calls (where a suspended caller might contain the oldest promotable task). Recently, Su et al. [2024] developed a compilation strategy for heartbeat scheduling that can outperform manual coarsening for data-dependent and/or irregular loops. They target a low-level representation similar to TPAL, with the same limitation of only considering intraprocedural loops.

Our Spork IR offers a solution to this intra- vs. interprocedural control-flow problem. The spork-spoin mechanism differs from TPAL's **prppt** essentially by implicitly marking *every* basic block between the spork-spoin pair as a possible promotion point, recording this information in the call stack at a predetermined location within the frame. We then forgo custom promotion handlers in favor of a uniform promotion handler which can be invoked at any moment. Although subtle, this is critical for enabling interprocedural promotions: our uniform promotion handler is implemented inside the runtime system and can be applied to any stack frame at a possible promotion point. (By inspecting a frame, the runtime system can determine whether or not the frame is promotable, and all promotable frames in our approach are promoted in the same manner.)

Lazy task creation and lazy scheduling. Our approach relies on a nearly zero-cost task representation (implicitly encoded in stack frames) which bears resemblence to lazy task creation (LTC), introduced by Mohr et al. [1991] to mitigate task overheads. The idea of LTC is to spawn tasks lazily, in response to system load imbalance. To implement LTC, various nearly zero-cost task representations have been developed, often by leveraging data implicitly stored in the call stack [Feeley 1992, 1993a; Goldstein et al. 1996]. LTC can require a careful protocol/handshake when integrated with work-stealing scheduling [Arora et al. 2001; Blumofe and Leiserson 1999], in particular, to coordinate the stealer with a search for an appropriate implicit task buried within the call stack [Handleman et al. 2025; Hiraishi et al. 2009; Kumar et al. 2012]. One advantage of the automatic parallelism management approach is that it avoids such a handshake by scheduling promotions locally, rather than in response to a steal request. Because LTC techniques rely on load (im)balance feedback, it can be beneficial to integrate LTC closely with the system scheduler [Bergstrom et al. 2012; Tzannes 2012; Tzannes et al. 2010, 2014], a technique known as lazy scheduling (LS). LTC/LS both help reduce the overheads of task creation as long as the system remains saturated, which works well on small core/processor counts. However, in contrast to heartbeat-based approaches, LTC/LS are unable to guarantee low overhead for task creation in all cases. Our approach takes inspiration from the long line of work on LTC/LS, especially the lazy task representation techniques.

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, 2016]. This approach requires the user to supply cost functions for parallel code, which is sometimes difficult and in general not always possible.

7 Conclusion

This paper presents Spork IR, an intermediate representation for parallelism management that can express parallel loops, parallel reductions, and fork-join parallelism. We formalize the semantics of Spork IR, establishes its key soundness properties in the Lean theorem prover, and present an implementation of the IR by extending the MPL compiler for Parallel ML. Our evaluation with a broad set of benchmarks shows that our techniques deliver excellent performance while requiring no effort to control the overhead of parallelism. Notably, the implementation delivers performance within 36% on average of manually optimized benchmarks across all core counts. These results show that automatic parallelism management extended with parallel loops can enable programs written with high-level parallel constructs to achieve fast performance without any manual coarsening by the programmer, thus making progress on the long-standing challenge of bridging the benefits of high-level parallelism abstractions with high performance.

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, Jatin Arora, Matthew Fluet, Ram Raghunathan, Sam Westrick, and Rohan Yadav. 2020. MPL: A High-Performance Compiler for Parallel ML. https://github.com/MPLLang/mpl.

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, 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. 2016. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming (JFP)* 26 (2016), e23.

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.

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

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.

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.

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.

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.

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. 2020. 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.

Aaron Handleman, Kyle Singer, Tao B. Schardl, and I-Ting Angelina Lee. 2025. Towards Zero Spawn Overhead: Work Stealing Without Deques. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2025, Portland, OR, USA, 28 July 2025 - 1 August 2025*. ACM, 75–88. doi:10.1145/3694906.3743349

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.

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.

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

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.

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.

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.

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, 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.

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.

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 Well-Formedness Rules of Spork IR

The *well-formedness* rules of Spork IR require 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 rules are judgments of the form d WF ("well-formed"), where d can be a program or any of its constituent parts (expression, basic block, 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: $\mathbf{fun}_r\, f(\bar{x})$ is a function that returns r values, and $\mathbf{block}_{\Gamma;r;\sigma}\, b(\bar{x})$ is a block with local temporary scope Γ which eventually leads to a return of r values and is statically nested under σ sporks. 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.

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

$$\frac{\cdots}{\Gamma \vdash e\ \mathrm{WF}_{\mathrm{expr}}} \qquad \frac{\Gamma \vdash e\ \mathrm{WF}_{\mathrm{expr}} \quad P;\bar{B};\Gamma \cup \{x\};r;\sigma \vdash C\ \mathrm{WF}_{\mathrm{code}} \quad x \notin \Gamma}{P;\bar{B};\Gamma;r;\sigma \vdash x \leftarrow e; C\ \mathrm{WF}_{\mathrm{code}}}$$

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

$$\frac{\begin{array}{c}\mathbf{fun}_n\ g(\bar{y})\{\_\} \in P \quad |\bar{x}| = |\bar{y}| \quad \bar{x} \subseteq \Gamma\\ \bar{B};\Gamma;r;\sigma \vdash b_{\mathrm{ret}}(\cdot^n)\ \mathrm{WF}_{\mathrm{cont}}\end{array}}{P;\bar{B};\Gamma;r;\sigma \vdash \mathbf{call}\ g(\bar{x}) \triangleright b_{\mathrm{ret}}\ \mathrm{WF}_{\mathrm{code}}} \qquad \frac{r = |\bar{x}| \quad \bar{x} \subseteq \Gamma}{P;\bar{B};\Gamma;r;\varnothing \vdash \mathbf{return}(\bar{x})\ \mathrm{WF}_{\mathrm{code}}}$$

$$\frac{\begin{array}{l}\bar{B};\Gamma;r;\sigma\cdot n \vdash b_{\mathrm{body}}(\cdot^0)\ \mathrm{WF}_{\mathrm{cont}}\\ \bar{B};\Gamma;n;\varnothing \quad\ \vdash b_{\mathrm{spwn}}(\cdot^0)\ \mathrm{WF}_{\mathrm{cont}}\end{array}}{P;\bar{B};\Gamma;r;\sigma \vdash \mathbf{spork}(b_{\mathrm{body}} \parallel b_{\mathrm{spwn}})\ \mathrm{WF}_{\mathrm{code}}} \qquad \frac{\begin{array}{c}\bar{B};\Gamma;r;\sigma \vdash b_{\mathrm{unpr}}(\cdot^0)\ \mathrm{WF}_{\mathrm{cont}}\\ \bar{B};\Gamma;r;\sigma \vdash b_{\mathrm{prom}}(\cdot^n)\ \mathrm{WF}_{\mathrm{cont}}\end{array}}{P;\bar{B};\Gamma;r;\sigma\cdot n \vdash \mathbf{spoin}(b_{\mathrm{unpr}}, b_{\mathrm{prom}})\ \mathrm{WF}_{\mathrm{code}}}$$

$$\frac{\mathbf{block}_{\Gamma';r;\sigma}\ b(\bar{x})\colon \_ \in \bar{B} \quad \Gamma' \subseteq \Gamma \quad |\bar{x}| = n}{\bar{B};\Gamma;r;\sigma \vdash b(\cdot^n)\ \mathrm{WF}_{\mathrm{cont}}}$$

Fig. 15. Typing rules of well-formed Spork IR programs

Figure 15 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 $P; f; \Gamma; r; \sigma \vdash C$ $\mathrm{WF}_{\mathrm{code}}$ 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 deque σ (which we also call its *spork nesting*), and every control flow path (barring the b_{spwn} continuation of a spork) out of this point must return r values. The auxiliary $f; \Gamma; r; \sigma \vdash b(\cdot^n)$ WF_{cont} rule ("continuation") stipulates that b is a block in f with arity n, temporary scope Γ, eventually returning r values, and spork nesting σ. Note, these rules do not distinguish between promoted and unpromoted spawn blocks 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 of σ in every function and allow the compiler to know what the (local) spawn deque is for every program point.

Our Lean formalization follows the definitions outlined in this paper very closely, but with one nontrivial difference which aids mechanization: rather than by names, our proofs refer to temporaries by their indices in the local scope. For example, $\textbf{block}_{\Gamma;r;\sigma}\, b(x, y)\colon z \leftarrow x{+}y; \textbf{goto}\ b'(z)$ for some $|\Gamma| = 2$ would instead be written $\textbf{block}_{r;\sigma}\, b(4)\colon \Gamma[2] + \Gamma[3]; \textbf{goto}\ b'(\Gamma[4])$. 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.

B Improvement Breakdown

Table 3. Breakdown of improvement factors in automatic parallelism management. $\frac{\text{MPL}^{\text{pc}}}{\textbf{MPL}^{\textbf{sp}}}$ reports our ($\textbf{MPL}^{\textbf{sp}}$) overall improvement over MPL^{pc}. $\frac{\text{D\&C}_{\text{spork}}}{\textbf{MPL}^{\textbf{sp}}}$ shows how much of this is due to our novel reduce encoding (Figure 8) in place of divide-and-conquer parallel loops, and $\frac{\text{MPL}^{\text{pc}}}{\text{D\&C}_{\text{spork}}}$ shows the remaining portion from our underlying use of spork and spoin to implement divide-and-conquer loops rather than PCall.

	T_1	Improvement					T_{80}	Improvement				
Benchmark	$\text{D\&C}_{\text{spork}}$	$\frac{\text{MPL}^{\text{pc}}}{\textbf{MPL}^{\textbf{sp}}}$		$\frac{\text{D\&C}_{\text{spork}}}{\textbf{MPL}^{\textbf{sp}}}$		$\frac{\text{MPL}^{\text{pc}}}{\text{D\&C}_{\text{spork}}}$	$\text{D\&C}_{\text{spork}}$	$\frac{\text{MPL}^{\text{pc}}}{\textbf{MPL}^{\textbf{sp}}}$		$\frac{\text{D\&C}_{\text{spork}}}{\textbf{MPL}^{\textbf{sp}}}$		$\frac{\text{MPL}^{\text{pc}}}{\text{D\&C}_{\text{spork}}}$
bfs	6.13	**1.94**	=	**1.98**	×	**.978**	.161	**1.75**	=	**1.72**	×	**1.02**
bignum-add	1.77	**2.34**	=	**2.28**	×	**1.03**	.028	**2.07**	=	**1.96**	×	**1.05**
delaunay	7.98	**1.05**	=	**1.05**	×	**.991**	.411	**.876**	=	**.864**	×	**1.01**
grep	4.90	**2.31**	=	**2.03**	×	**1.14**	.076	**2.14**	=	**1.75**	×	**1.23**
line-fit	2.60	**2.68**	=	**2.03**	×	**1.32**	.040	**1.68**	=	**1.13**	×	**1.49**
mandelbrot	4.17	**1.48**	=	**1.59**	×	**.932**	.058	**1.48**	=	**1.49**	×	**.990**
map-heavy	4.25	**.809**	=	**1.01**	×	**.803**	.055	**.808**	=	**.999**	×	**.808**
map-light	4.87	**12.1**	=	**4.94**	×	**2.45**	.107	**6.10**	=	**3.23**	×	**1.89**
merge-sort	6.16	**1.14**	=	**1.09**	×	**1.04**	.100	**1.15**	=	**1.12**	×	**1.02**
nearest-nbrs	1.45	**1.10**	=	**1.08**	×	**1.02**	.028	**1.09**	=	**1.03**	×	**1.05**
nqueens	2.11	**1.56**	=	**1.43**	×	**1.09**	.030	**1.54**	=	**1.34**	×	**1.14**
primes	6.83	**7.05**	=	**3.26**	×	**2.17**	.155	**4.36**	=	**2.63**	×	**1.66**
sparse-mxv	5.84	**3.79**	=	**3.35**	×	**1.13**	.082	**2.69**	=	**2.25**	×	**1.19**
suffix-array	4.77	**1.90**	=	**1.69**	×	**1.13**	.102	**1.73**	=	**1.62**	×	**1.07**
triangle-count	9.85	**1.14**	=	**1.10**	×	**1.04**	.156	**1.09**	=	**1.03**	×	**1.05**
word-count	1.97	**2.67**	=	**1.78**	×	**1.50**	.028	**2.03**	=	**1.27**	×	**1.59**
min		**.809**		**1.01**		**.803**		**.808**		**.864**		**.808**
geomean		**2.09**	=	**1.77**	×	**1.18**		**1.74**	=	**1.48**	×	**1.17**
max		**12.1**		**4.94**		**2.45**		**6.10**		**3.23**		**1.89**