Spork: Automatic Parallelism Management for Loops

ANONYMOUS AUTHOR(S)

Parallel loops are ubiquitous in high-level parallel programming languages. The semantics of parallel loops is reasonably straightforward and similar to their sequential counterparts. They are not hard to write, but they are extremely hard to "get right": unless the programmer carefully controls the overhead of parallelism exposed by a parallel loop, its performance will be dismal, so much so that it may be outperformed by its sequential counterpart. There has been some progress on automatic granularity control to reduce the burden of manual performance optimizations, but no existing approach performs well, especially for arbitrary loop bodies that may, for example, include arbitrary nesting, which is very common.

In this paper, we present automatic parallelism management techniques for parallel loops. These techniques aim to maximize the benefit of parallelism while minimizing its cost without restrictions on the expressiveness of the loops. To this end, we present two low-level primitives called **spork** and **spoin** that can be used to express loops that execute sequentially with little overhead while remaining "ready to go parallel" at any moment during execution. We formalize the semantics of these primitives and present compilation techniques for compiling high-level parallel loops into low-level codes that use **spork** and **spoin**. When coupled with a runtime system that judiciously decides when to actualize parallelism, these primitives allow the overheads of parallelism to be amortized against real, sequential work. We implement our techniques and perform an experimental evalation considering a range of benchmarks, including parallel codes that have been manually optimized over many years. The experiments show that our techniques perform well in practice, delivering good overheads and speedups, that are within 26% of manually optimized parallel codes, while requiring absolutely no human effort for performance optimization.

CCS Concepts: • **Software and its engineering → Parallel programming languages**.

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

1 Introduction

Parallelism has come a long way. In the 1980s, theoreticians noticed that it is possible to design efficient parallel algorithms just like sequential ones and did so for many problems [Jaja 1992]. The theoreticians of the day, however, worked on a model called PRAM (Parallel Random Access Machine) that was so out of touch with reality that by the 1990s, it crashed under its own weight, overtaken by a form of creative destruction that led, starting in the mid-2000s, to the development of multicore architectures. Compared to PRAM, multicore architectures were more asynchronous and also more tightly coupled, allowing faster access to memory. Ensuing advances in GPUs (Graphics Processing Units) and other specialized parallel architectures for tensor processing, and their applications to AI (and Large Language Models) have proved the staying power of parallelism, surpassing perhaps the wildest dreams of its early advocates of the yore.

In contrast to speed of advances in parallel architectures, the going has been rough for parallel software. In principal, it is not difficult to write parallel programs by using high-level parallelism constructs such as parallel loops supported by modern programming languages. But writing ***performant*** parallel programs, which compete with sequential codes on small numbers of cores while also scaling to larger numbers, remains a major challenge. For example, just as we could implement a simple sequential matrix multiplication with three nested loops, we could implement a parallel matrix multiplication with three nested "parallel-for" loops. But whoever runs this code is in for a rude awakening: on a single core, the parallel code will be an order of magnitude slower than its sequential counterpart and will struggle to catch up, even as we use more cores.

Why would such a simple parallel program perform so poorly? The problem is that parallelism is not free: parallel codes incur overhead to spawn, schedule, and synchronize parallel tasks. For example, every iteration of a parallel loop can spawn a task to execute the body of the loop in parallel. Such a spawn operation requires thousands of cycles even with the fastest implementations on modern hardware [Ghosh et al. 2020a]. Yet, the body of a loop can be relatively tiny, perhaps as few as a couple dozen cycles (as in the parallel matrix multiple example).

Today, we expect the programmer to control the cost-benefit ratio of parallelism by coarsening parallel loops. Specifically, the programmer splits the loop into chunks and spawns only one task per chunk, thereby amortizing the cumulative overhead of parallelism [Tzannes et al. 2014]. This "tuning" requires great care, because if the chunks are too coarse, then they will reduce parallelism and harm scalability; if the chunks are too fine, then the overheads will be large. But what exactly is "too coarse" and "too fine"? This depends on the architecture, the software stack, and even the actual input to the program, especially in modern workloads which tend to be data-dependent (e.g., sparse) and polymorphic. For example, the input to a parallel matrix multiplication routine can be a matrices of bits, floating point numbers, or an algebraic data structure; matrices may vary from dense to sparse, and anything in between. Thus even if the programmer somehow manages to coarsen perfectly, they end up overfitting the code to the architecture, to the software stack, and to the inputs considered, jeopardizing the portability of the program [Tzannes 2012]. Furthermore, the resulting code contains chunk size parameters which leak across module abstraction barriers, e.g., showing up as function arguments that allow adjusting the chunk sizes based on arguments at each call site.

Motivated by the challenges of manual programmer-driven granularity control, researchers have sought automation. Early work on oracle-guided scheduling provided the first provably efficient grainularity control technique but required programmer annotations [Acar et al. 2016a]. Subsequent work on heartbeat scheduling eliminated the need for annotations to provide a fully automatic technique [Acar et al. 2018; Rainey et al. 2021; Su et al. 2024]. Considering the Parallel ML language with fork-join parallelism, more recent work [Westrick et al. 2024] combined compiler, run-time techniques, and heartbeat scheduling to automate parallelism management entirely. Automatic parallelism management allows programmer to express all potential for parallelism without any worry about performance tuning.

In this paper, we extend automatic parallelism management to support ubiquitous parallel loops. The idea behind our approach is to compile parallel loops into a form which executes sequentially but can, at a moment's notice, be split into multiple parallel tasks. To realize this idea efficiently without restricting the loops, we propose two low-level control-flow constructs, called ***spork*** (**s**equential or **p**arallel f**ork**) and ***spoin*** (**s**equential or **p**arallel j**oin**).

At a high level, **spork** specifies a loop that runs sequentially by default but remains "parallel ready" to be parallelized dynamically; symetrically, **spoin** specifies the synchronization needed for such a loop based on whether it was parallelized or not. From an operational perspective, each **spork** registers an alternative code path for a parallel task, which can be represented implicitly in the call stack, making its sequential execution cost essentially zero. If the runtime decides to "go parallel" then it does so by creating a bona fide task from the implicit representation. Each **spork** has a matching **spoin** that decides whether to continue sequentially or to perform a parallel synchronization, depending on decision made by the runtime.

To support performant parallel loops, the compiler wraps every loop body with a **spork**-**spoin** pair, registering a parallel task to complete the remainder of the iterations, while also executing sequentially when the runtime deems parallelism unnecessary. The runtime has the choice to parallelize a loop at each iteration at a cost but this cost will be born almost entirely when parallelism is actualized. This in turn enables us to amortize the cost by using the heartbeat scheduling technique [Acar et al. 2018; Rainey et al. 2021] that ensures that parallelism is created only when its costs are amortized. In a nutshell, our approach moves all loop parallelization costs off the fast, sequential path, while ensuring that parallelism can be exploited without any restrictions.

We formalize the semantics of spork and spoin primitives and implement the semantics by extending the MaPLe compiler and run-time system. The implementation incorporates the primitives into an SSA intermediate representation (IR) and adapts existing optimizations and compilation passes appropriately. Our design of **spork** (and **spoin**) allow for certain optimizations, such as function inlining, which results in static nesting of **spork**-**spoin** pairs (corresponding to statically nested loops), which are key to efficiency. Even though **spork** and **spoin** are well suited to parallel loops, they can also encode other loop-like constructs, including for example, parallel reductions.

We evaluate our approach on over a dozen benchmarks from the Parallel ML Benchmark Suite [Arora et al. 2021, 2023; Westrick et al. 2024], covering a variety of problem domains, including graph analysis, computational geometry, sparse linear algebra, numerical algorithms, and text analysis. Compared to prior work that parallelizes loops by fork-join primitives and uses automatic parallelism management to handle them, our approach improves performance by a factor of 2x (on average) for both sequential and parallel runs. Perhaps most notably, we observe that (average) parallel overheads with respect to sequential runs is 2x with good scalability, leading to 28x average speedups on 80 cores over sequential (a 46x self speedup). Finally, our approach is less than 25% slower than manually optimized benchmarks for all core counts. These results show that automatic parallelism is not just a theoretical idea but can deliver us a future where parallelism can be managed automatically without any programmer involvement.

The specific contributions of the paper include the following.

- The design of ***spork*** (and ***spoin***), new control-flow primitives which are suitable for the implementation of heartbeat-driven parallel loops.
- Formal definitions of ***spork*** and ***spoin*** in terms of an SSA intermediate representation and an operational semantics.
- A compilation strategy for expressing `parfor` and `reduce` primitives in terms of spork and spoin, including important optimizations.
- A full implementation in the MaPLe compiler and run-time system.
- An empirical evaluation, demonstrating on over a dozen benchmarks that our approach is capable of guarantee low overhead and high scalability without any manual chunking/tuning of parallel loops.

2 Overview and Key Ideas

We consider an ML-like (higher-order, polymorphic, etc.) source language with support with parallel for-loops and parallel reductions in the form of the following two higher-order functions.[1]

$$\texttt{parfor} : \texttt{int} \times \texttt{int} \times (\texttt{int} \to \texttt{unit}) \to \texttt{unit}$$

$$\texttt{reduce} : (\alpha \times \alpha \to \alpha) \times \alpha \times \texttt{int} \times \texttt{int} \times (\texttt{int} \to \alpha) \to \alpha$$

The semantics of $\texttt{parfor}(i, j, f)$ is to execute all of $\{f(i), f(i+1), \ldots, f(j-1)\}$ in parallel. Similarly, the semantics of $\texttt{reduce}(c, z, i, j, f)$ is to compute the "sum" of $\{f(i), f(i+1), \ldots, f(j-1)\}$ with respect to the binary associative function c and corresponding "zero" element z. Throughout the paper, we will refer to both `parfor` and `reduce` as "parallel loops", where the function f in both cases is the "body" of the loop. These primitive parallel loops can be used to implement a wide variety of common parallel operations, such as `map`, `filter`, `scan` (prefix sums), `flatten`, and many others [Westrick et al. 2022b].

[1]In our actual implementation, implement `parfor` as a special case of `reduce`, using the trivial combining function over the type `unit`. This is optimized away by the compiler, producing an efficient implementation of `parfor`.

The primary contribution of this paper is a technique for compiling and executing parallel loops which guarantees low overhead while maintaining high scalability, regardless of what code appears within the loop body. This is difficult, because loops can contain 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 loops perform well, in all possible cases, with no need for programmer intervention.

Spork and spoin: control-flow primitives for splittable loops. To meet our goal, we propose two new control-flow primitives called ***spork*** and ***spoin*** which are used to encode *splittable loops*. A ***splittable loop*** executes sequentially by default, with nearly zero overhead relative to a sequential loop, but at any moment can be interrupted and split into two or more parallel tasks, exposing parallelism. Spork and spoin are used to delimit a splittable loop body, with **spork** appearing at the beginning and **spoin** appearing at the end. Arbitrary code may appear between the two, with the only restriction that every control-flow path leading out of the **spork** must eventually reach a **spoin**. The **spork** is used to register an alternative code path for a potential split of the loop. The **spoin** is ultimately compiled into a conditional, to check whether or not the split occurred. In the resulting executable, control-flow can be dynamically diverted in response to a split.

We can then amortize all of the overheads of splitting (including the cost of spawning, scheduling, and synchronizing tasks) by scheduling splits "infrequently", using a technique known as *heartbeat scheduling* [Acar et al. 2018; Rainey et al. 2021]. The idea is to interrupt the execution periodically by delivering a "heartbeat signal". Upon receiving the signal, every thread executing a loop performs a split. By spacing the heartbeats sparsely, this technique allows amortizing the overhead of splitting to the cost of the useful, sequential work done between each heartbeat.

Our approach seamlessly supports nested parallel loops with any amount of static or dynamic (e.g., recursive) nesting. For example, a **spork**-**spoin** pair may be statically nested within another; alternatively, an inner loop may execute within a function call in the body of an outer loop. In these cases, at each heartbeat, we have a choice of *which* loop to split. To ensure high scalability, we always split the *oldest* (i.e., outermost) loop. Splitting the oldest loop is critical for performance, ensuring that the critical path of the computation is stretched by at most a constant factor [Acar et al. 2018]. That is, by always splitting the outermost loop, we ensure that all theoretical parallelism of the source program is (asymptotically) preserved.

"Three-way" splits. With each **spork**-**spoin** pair, we have to specify exactly how the loop is to be split (if a split occurs). Our approach is to generate two new tasks at each split, each of which is responsible for half of all remaining iterations; the original task is left only to finish its current iteration. This is essentially a "three-way" split, where an original task responsible for the index range (i, j) splits into three tasks corresponding to the ranges $(i, i+1)$, $(i+1, m)$, and (m, j), respectively, where $m = \lfloor \frac{i+1+j}{2} \rfloor$ is the midpoint between $i+1$ and j. Note that this strategy is asymptotically optimal from a parallelism perspective, as it ensures at most a logarithmic number of splits along the critical path.

An example of the three-way splitting policy is shown in Figure 1 which illustrates the execution of `parfor(0,15,f)` for some function `f`. At each heartbeat, all remaining iterations are split off of the current iteration and split in half, creating two tasks. The two new tasks recursively execute instances of `parfor(...,f)` on subranges determined dynamically, at the time of each split. Each shaded box delimits a recursive `parfor` instance. Note the second heartbeat in the diagram, which is delivered to all active loop iterations, including `f(2)`, which was already previously split. In this case, any loop within `f(2)` would be split, adhering to the outermost-first splitting policy.

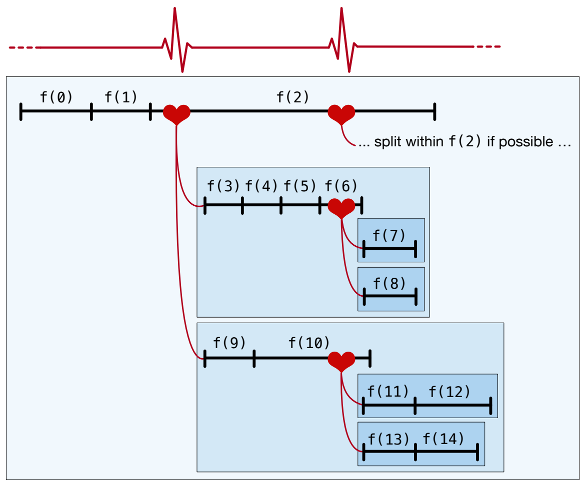

Fig. 1. Example execution of `parfor(0,15,f)` with our three-way splitting policy, driven by a regular heartbeat. At each heartbeat, all remaining iterations (if any) are split off of the current iteration and split in half, creating two tasks.

The cost of splitting. The traditional strategy for implementing a parallel loop is to split the loop range into subranges, which run recursively in parallel. Although it creates parallelism, this strategy incurs significant overheads compared to its sequential counterpart: computing the midpoint of the loop range requires a handful of instructions, and each recursive call has push and pop frames on the call-stack. These overheads amortize well if the loop body is itself large, but otherwise, they dominate and can harm performance especially in highly irregular workloads where it is difficult to predict the cost of each loop iteration. Concretely, we have measured that recursive splitting—even when executed sequentially—can be as much as 6x slower than a sequential loop. Our approach moves all of this overhead away from the "fast path", and instead incurs this overhead infrequently, at each heartbeat.

Low-level intuition. The utility of **spork** and **spoin** is that they can be expressed at a reasonably high level of abstraction, allowing them to be integrated into a compiler and subjected to standard compiler optimizations. Eventually, these control-flow primitives are lowered into executable code, and, to provide the reader with some intuition, we briefly describe how the final executable operates.

Our approach hinges on the ability to interrupt and split any loop that is currently in flight. We rely on standard signal handling mechanisms (specifically software polling [Basu et al. 2021; Feeley 1993b; Ghosh et al. 2020b]) to switch to a signal handler whenever a heartbeat signal arrives. The signal handler can then inspect the current call-stack and locate a frame corresponding to an in-flight loop. Here, we leverage a static classification of return addresses: every return address either returns to code within a loop body (statically delimited by a **spork**-**spoin** pair), or it returns to some non-loop code. This information is accessible at run-time via a static "frame info" lookup table, which we include in the compiled executable.

After locating an appropriate frame, the signal handler can then perform a split, which creates a new task and adjusts the behavior of the original task (to synchronize with the new task, instead of continuing the loop). We create new tasks by copying the frame and modifying the return address of the copy, causing it to return to a different code path when resumed; this alternative code path is statically encoded with the **spork**. To adjust the behavior of the original task, we write a pointer to the spawned task into a designated slot of the original stack frame. This designated slot is inspected

Program	P	$::= \mathbf{let}\ \bar{F}\ \mathbf{in}\ f_{\text{main}}$	*Expression*	e	$::= v \mid x \mid x + y \mid \ldots$
Function	F	$::= \mathbf{fun}\ f(\bar{x}) = \mathbf{let}\ \bar{B}\ \mathbf{in}\ b_{\text{entry}}$	*Value*	v	$::= () \mid \texttt{true} \mid \texttt{false} \mid \ldots$
Basic block	B	$::= \mathbf{block}\ b(\bar{x}) = C$	*Function name*	f, g	
Block code	C	$::= S; C \mid T$	*Block label*	b	
Statement	S	$::= x \leftarrow e$	*Temporary*	x, y	

$$\begin{array}{lll} \textit{Transfer} & T & ::= \mathbf{goto}\ b_{\text{next}}(\bar{x}) \mid \mathbf{if}(x, b_{\text{then}}, b_{\text{else}}) \mid \mathbf{call}\ f(\bar{x}) \triangleright b_{\text{ret}} \mid \mathbf{return}(\bar{x}) \\ & & \quad \mid \mathbf{spork}(b_{\text{body}}, b_{\text{spwn}}) \mid \mathbf{spoin}(b_{\text{unpr}}, b_{\text{prom}}) \mid \mathbf{retjoin}(x) \end{array}$$

Fig. 2. Syntax of SSA$^{\text{SP}}$, with the three new transfers highlighted: **spork**, **spoin**, and **retjoin**.

at each **spoin**, which then jumps to the appropriate code: either continuing the loop sequentially (the "fast path"), or synchronizing with the spawned task (the "slow path"). More implementation details are given in Section 4.

3 Spork: Sequential/Parallel Fork

We introduce a new intermediate representation language, SSA$^{\text{SP}}$, derived from static single assignment form (SSA), and how to lower source-level `reduce` calls into SSA$^{\text{SP}}$. SSA$^{\text{SP}}$ extends SSA with three additional basic block transfers for managing parallelism: **spork**, **spoin**, and **retjoin**.

3.1 The SSA$^{\text{SP}}$ Intermediate Representation

Figure 2 defines the syntax of SSA$^{\text{SP}}$. A program P is a list of first-order functions, one specially marked main. Each function has a name, list of parameters, and a list of basic blocks, including one marked as the function entry point.

Each basic block is a label, list of parameters, and a list of statements terminated by a transfer. Statements assign the value of an expressions to a temporary (e.g. $x \leftarrow y + z$), and transfers enable control flow across basic blocks (**goto**, **if**), functions (**call**, **return**), and in SSA$^{\text{SP}}$, across threads.

SSA$^{\text{SP}}$ extends SSA by introducing three new transfers, highlighted in Figure 2. The $\mathbf{spork}(b_{\text{body}}, b_{\text{spwn}})$ (**s**equential/**p**arallel f**ork**) transfer behaves as a $\mathbf{goto}\ b_{\text{body}}()$, but it additionally opens a scope in which b_{spwn} is a potential entry block for a new thread, should the program choose during execution to spawn a thread while inside the scope. The $\mathbf{spoin}(b_{\text{unpr}}, b_{\text{prom}})$ transfer closes this scope, and performs a conditional jump: b_{unpr} if the program never spawned a thread for b_{spwn}, and b_{prom} if it did. In the second (parallel) case, the spawned thread must terminate with the $\mathbf{retjoin}(x)$ transfer, which returns the value of the x temporary back to the parent thread and exits. Then, when the parent thread closes this scope with $\mathbf{spoin}(b_{\text{unpr}}, b_{\text{prom}})$, it synchronizes with the child thread and jumps to b_{prom}, a basic block that receives the value from the child thread's **retjoin** as an argument.

3.2 Operational Semantics

We present definitions for the operational semantics of SSA$^{\text{SP}}$ in Figure 3. Note, we use $\varnothing$ for an empty list and $a \cdot b$ is the concatenation of lists a and b. A thread pool $\mathcal{P}$ is a nonempty list of threads. Each thread consists of a call stack paired with the remaining code from the basic block the thread is executing. A call stack is a nonempty list of stack frames, each with three components: (1) a deque ρ of spawn block labels, one for each unpromoted

$$\begin{array}{lll} \textit{Thread pool} & \mathcal{P} ::= & \overline{\mathcal{T}} \\ \textit{Thread state} & \mathcal{T} ::= & \mathcal{K} \diamond C \\ \textit{Call stack} & \mathcal{K} ::= & \overline{k} \\ \textit{Stack frame} & k ::= & \langle \rho, \mathcal{X}, b_{\text{ret}}? \rangle \\ \textit{Spawn deque} & \rho ::= & \overline{b}_{\text{spwn}} \\ \textit{Value map} & \mathcal{X}, \mathcal{Y} \in & (\textit{temp}) \rightharpoonup (\textit{value}) \end{array}$$

Fig. 3. Definitions for SSA$^{\text{SP}}$ operational semantics

$$\frac{\mathcal{P} \mapsto \mathcal{P}'}{\mathcal{P}_1 \cdot \mathcal{P} \cdot \mathcal{P}_2 \mapsto \mathcal{P}_1 \cdot \mathcal{P}' \cdot \mathcal{P}_2}\ \textsc{step}$$

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

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

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

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

$$\frac{\textbf{block}\ b_{\text{body}}() = C}{\mathcal{K} \cdot \langle \rho, \mathcal{X} \rangle \diamond \textbf{spork}(b_{\text{body}}, b_{\text{spwn}}) \mapsto \mathcal{K} \cdot \left\langle \rho \cdot b_{\text{spwn}}, \mathcal{X} \right\rangle \diamond C}\ \textsc{spork}$$

$$\frac{\forall\ \langle \rho, \_, \_ \rangle \in \mathcal{K}.\, \rho = \varnothing \qquad \textbf{block}\ b_{\text{spwn}}() = C'}{\mathcal{K} \cdot \left\langle b_{\text{spwn}} \cdot \rho, \mathcal{X}, b_{\text{ret}} \right\rangle \cdot \mathcal{K}' \diamond C \mapsto \left(\mathcal{K} \cdot \langle \rho, \mathcal{X}, b_{\text{ret}} \rangle \cdot \mathcal{K}' \diamond C\right) \cdot \left(\langle \varnothing, \mathcal{X} \rangle \diamond C'\right)}\ \textsc{promote}$$

$$\frac{\textbf{block}\ b_{\text{unpr}}() = C}{\mathcal{K} \cdot \left\langle \rho \cdot b_{\text{spwn}}, \mathcal{X} \right\rangle \diamond \textbf{spoin}(b_{\text{unpr}}, \_) \mapsto \mathcal{K} \cdot \langle \rho, \mathcal{X} \rangle \diamond C}\ \textsc{spoin-unprom}$$

$$\frac{\textbf{block}\ b_{\text{prom}}(x) = C}{\left(\mathcal{K} \cdot \langle \varnothing, \mathcal{X} \rangle \diamond \textbf{spoin}(\_, b_{\text{prom}})\right) \cdot \left(\langle \varnothing, \mathcal{Y} \rangle \diamond \textbf{retjoin}(y)\right) \mapsto \mathcal{K} \cdot \langle \rho, \mathcal{X}[x \hookrightarrow \mathcal{Y}(y)] \rangle \diamond C}\ \textsc{spoin-prom}$$

Fig. 4. Selected rules from SSA$^{\text{SP}}$ operational semantics

spork-**spoin** scope we are inside (local to this stack frame, i.e. those entered while this was the current stack frame), (2) a mapping $\mathcal{X}$ that stores the value of each temporary in scope, and (3) an optional continuation block b_{ret} for returning to this stack frame after a return, present in all but the current stack frame.

We define the execution of SSA$^{\text{SP}}$ via the small-step operational semantics in Figure 4. Each rule is of the form $\mathcal{P} \mapsto \mathcal{P}$, modifying the pool of current threads:

- STEP allows arbitrary stepping of any thread (or slice) in the thread pool, regardless of the position it occurs in that pool.
- STMT executes a statement $x \leftarrow e$ by evaluating e and associating x with its value in the current frame's value mapping.
- GOTO jumps to a new block, assigning values to its parameters from the arguments provided.
- CALL saves which block to return to, pushes a new stack frame onto the call stack, and initializes it by mapping from function parameters to the values of the arguments.
- RETURN conversely pops the current stack frame and returns to the caller's, passing the returned value(s) as arguments to b_{ret}.
- SPORK allows its b_{spwn} block to be promoted into a thread later by pushing it onto the end of the current frame's spork deque, then continues with the body block.
- PROMOTE may happen nondeterministically at any point while this block is in the spork deque. It finds the oldest spawn block across all stack frames on the call stack (including the current frame), pops it, and creates a new thread running that block.
- SPOIN-PROM happens at a **spoin** when its associated **spork** was promoted. It requires that the spork deque is empty: because promotions happen in queue order, we know the associated **spork**'s spawn block was promoted only if there are no other blocks to promote. For similar reasons, we know the successive thread is the child to synchronize with. Once

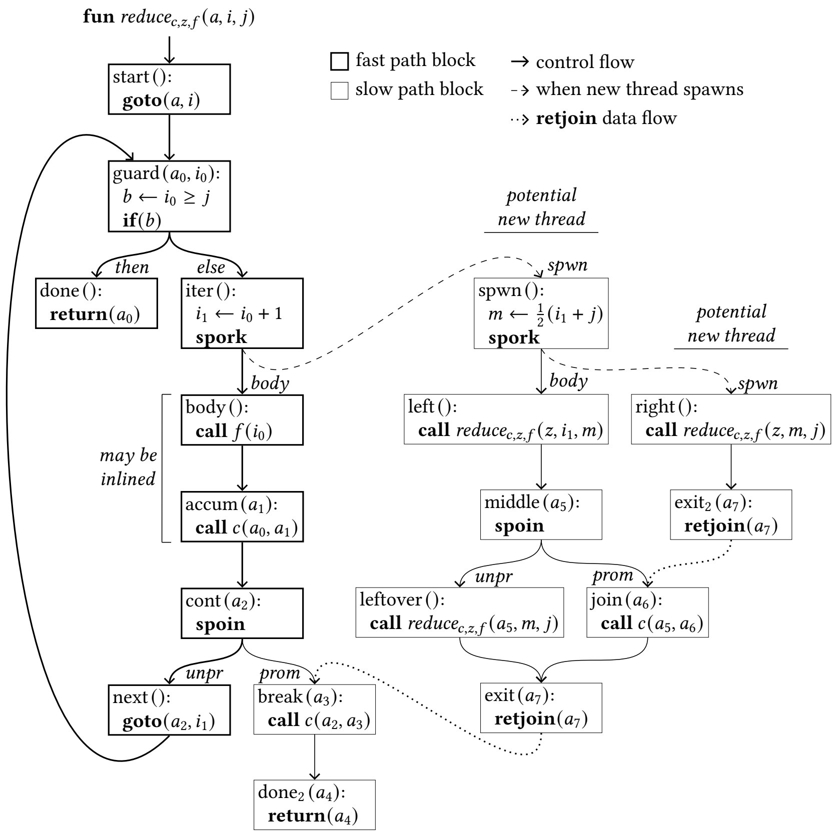

Fig. 5. Implementing parallel reduce in SSA$^{\text{SP}}$ for a particular c, z, f. For each call to the higher-order reduce(f, g, z, i, j), we generate a first-order $reduce_{c,z,f}(z, i, j)$ function unique to that call. The calls to f and c on the fast path may (and often will) be inlined.

the child ends with a **retjoin**(y), the original thread uses the value of y as an argument to the b_{prom} block.

- **SPOIN-UNPROM** happens at a **spoin** when its spork remained unpromoted, indicated by a nonempty spork deque. It closes the **spork**-**spoin** pair by popping from the end of the spork deque (which prevents that block from being promoted in the future), then jumps to the b_{unpr} block because the scope was unpromoted.

3.3 Implementing Parallel reduce with spork

Using **spork**, **spoin**, and **retjoin**, we can implement a parallel reduce in SSA$^{\text{SP}}$ for a particular c, z, and f as in Figure 5, which allows us to achieve low sequential overhead while maintaining good scalability on many cores. Aside from promotions (which are amortized by sequential work), this *reduce* is much like a sequential fold. The function's implementation starts with the *guard* block, which checks if the loop is complete (that is, $i \geq j$). If there is remaining work to do, it **spork**s: by default, the program continues to the *body* block, which calls f with the iteration index i_0 and then combines the result with the 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.

However, if the program wants to spawn a thread while evaluating the loop body and finds that this is the oldest unpromoted **spork**, it creates a new thread running *spwn*. Then, the original thread resumes its execution and, when finished, **spoin**s: since the **spork**'s potential parallelism was promoted, it waits for the newly spawned thread to exit with **retjoin** and passes that value as an argument to *break*. It then calls the combine operator c with the accumulated result of the iterations up to this point (a_1) and the result of the rest as computed on the spawned thread (a_2), finally returning that value.

Note that this implementation allows for every single loop iteration to become a task with its own thread if needed. Additionally, f and c can be inlined in the loop body, allowing arbitrary nesting of `reduce`. This is important for the performance of short, tight loops and nested parallel loops: the modest overhead of a function call for every loop iteration can be detrimental to the overall performance of the program. Our design allows for inlining to avoid this, as the fast path of `reduce` becomes entirely intraprocedural (having no function call) when f and c are inlined. In the case of a benchmark with nested loops (*sparse-mxv-csr*), we observe as much as +25% speedup compared to the same program but where the nested `reduce` call is not inlined.

3.4 Implementing par with spork

While we can write loop-level parallel programs in **MPL$^{\text{sp}}$** by using `reduce`, we additionally include a primitive higher-order function `par` to support divide-and-conquer style algorithms:

$$\texttt{par} : (\texttt{unit} \to \alpha) \times (\texttt{unit} \to \beta) \to \alpha \times \beta$$

which executes its arguments potentially in parallel and returns a tuple of their results. As is the case with `reduce`, by the time the program is lowered to SSA$^{\text{sp}}$, it has been subjected to transformations which change each call $\texttt{par}(f, g)$ into a call to a specialized first-order variant $par_{f,g}()$.

We implement each $par_{f,g}()$ as the SSA$^{\text{sp}}$ function shown in Figure 6. To begin, *par* immediately **spork**s, calling $f()$. If the program wants to promote something while evaluating $f()$ and there are no older unpromoted **spork**s, it spawns a new thread running $g()$. When the original thread finishes evaluating $f()$, it checks if a promotion occurred with **spoin**. If another thread did run $g()$, then it synchronizes with that thread and returns a tuple of their results. Otherwise, it runs $g()$ serially (in the original thread) and then returns the two results.

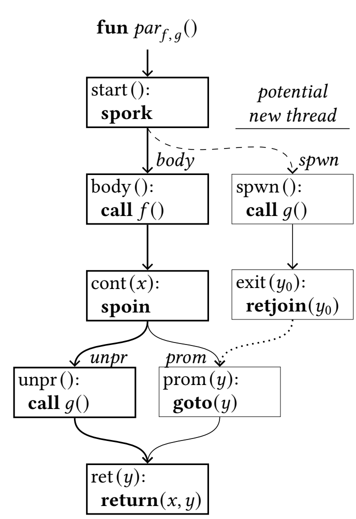

Fig. 6. Implementing `par` in SSA$^{\text{sp}}$.

4 Implementation

We have implemented SSA$^{\text{sp}}$ (Section 3), along with associated parallelism management infrastructure, in the context of a compiler and runtime system dubbed **MPL$^{\text{sp}}$** ("MaPLe with a **spork**"). **MPL$^{\text{sp}}$** is the latest version of MPL ("MaPLe"), which has been exploring efficient and scalable parallel functional programming by coupling thread scheduling and memory management for nested fork-join parallelism [Acar et al. 2015] through disentanglement [Arora et al. 2021; Westrick et al. 2022a, 2020] and hierarchical heaps [Guatto et al. 2018; Raghunathan et al. 2016]. **MPL$^{\text{sp}}$** is the second version of MPL that employs heartbeat scheduling for automatic parallelism management; it succeeds MPL$^{\text{s}}$ ("Sugar MaPLe") [Westrick et al. 2024], which used a *potentially parallel function call* (**pcall**) primitive to efficiently implement the coarse-grained two-way `par`, but could not efficiently implement the fine-grained parallel `reduce`. The various versions of MPL (collectively referred to as

MPL*) are themselves derived from MLton [MLton nd; Weeks 2006], a whole-program optimizing compiler for Standard ML. 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.

In this section, we present an overview of the important aspects of the $\mathbf{MPL^{sp}}$ implementation.

4.1 Separation of Compiler and Scheduler

Implementing the semantics of $\mathrm{SSA}^{\mathrm{SP}}$ in $\mathbf{MPL^{sp}}$ requires integration with the thread-scheduling components of MPL. In particular, the **PROMOTE** rule creates a new thread and the **SPOIN-PAR** 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 $\mathrm{SSA}^{\mathrm{SP}}$ would require the backend of the compiler to lower **spoin** and **setjoin** 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 $\mathrm{SSA}^{\mathrm{SP}}$ in source SML code and the runtime system and the control-flow aspects in the compiler. To faithfully model our implementation, we briefly describe a simple variant of $\mathrm{SSA}^{\mathrm{SP}}$. This variant changes the b_{spwn} block of **spork** to unary (rather than nullary), replaces the **retjoin** transfer with the combination of a **setjoin** binary expression and a **getjoin** unary expression, and changes the **spork** deque to contain either block labels or join tokens as elements. The **PROMOTE** rule, rather than popping a b_{spwn} label from the front of the **spork** deque, *replaces* the oldest b_{spwn} label in the **spork** deque with a fresh join token and spawns a new thread that executes b_{spwn} with the join token as its argument.[2] As before, if a b_{spwn} label can be popped from the back of the **spork** deque, then the **SPOIN-UNPROM** rule executes b_{unpr}. But, if a join token can be popped, then the **SPOIN-PROM** rule executes b_{prom} with the join token as an argument. A **JOIN** rule, that is agnostic to the mechanism by which threads are spawned, allows one thread executing a $\mathbf{setjoin}(j, v)$ expression (where j is a join token) to synchronize with another thread executing a $\mathbf{getjoin}(j)$ expression (with the same join token) and continues the first thread with a unit value and the second thread with v.

The compiler only "knows" about **spork** and **spoin** transfers, while the **setjoin** and **getjoin** operations are implemented in source SML code. It is a simple matter to ensure that the code corresponding to b_{spwn} ends with a **setjoin** followed by a thread exit and that the code corresponding to b_{prom} begins with a **getjoin** (see Section 4.6).

4.2 Back-End Changes: Using Frames to Implement spork, spoin, and Promotion

The most challenging aspect of the implementation is to efficiently realize the dynamic **spork** deque in a manner that both allows the runtime system to identify the oldest **spork** that can be promoted and admits an efficient implementation of **spoin**, particularly the determination of whether or not the last **spork** was promoted.

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 induce ***spork** scopes* that are *properly nested* (if,

---

[2] In this variant, elements are never popped from the front of the deque, so it might be better described as a ***spork** stack*.

due to inlining, there are multiple **spork**-**spoin** pairs in a function). Informally, a **spork**'s scope is a region of the control-flow graph that must be entered via the b_{body} of the **spork** and exited via the matching **spoin**.[3] Proper nesting means that, for any distinct pair of **spork**s in a function, their scopes are either disjoint or one is a proper subset of the other.

In fact, we do not require the correspondence between **spork**s and **spoin**s to be one-to-one; it suffices that each **spork** is matched by one or more **spoin**s and each **spoin** is the match of exactly one **spork**.[4] This allows the scope of a **spoin** to be exited via different matching **spoin**s along different control-flow paths, rather than requiring control flow to join in order to exit via a unique matching **spoin**. In Sections 4.6 and 4.6, we will discuss how this weaker notion is used to reduce the overhead on the fast path. The manner in which we expose **spork** and **spoin** in source SML code will guarantee that all functions will have properly-nested **spork** scopes.

Given the control-flow graph of a function with properly-nested **spork** scopes, we can perform a simple analysis to statically determine, at each control-flow point, the nesting of **spork** scopes that have been entered (by traversing the b_{body} edge of a **spork**) but not exited (by passing through a matching **spoin**). A static **spork** nesting is a sequence, where the first element is the **spork** of outermost (largest) scope and the last element is the **spork** innermost (smallest) scope; it is sometimes useful to consider a static **spork** as the sequence of b_{spwn} labels of the **spork**s. This static nesting of **spork** scopes is the key to an efficient implementation of **spork** and **spoin** transfers. The static **spork** nesting at a control-flow point approximates the dynamic **spork** deque of both the original SSA$^{\text{sp}}$ from Section 3 and the variant described above. Specifically, when the control-flow point is executed in the variant semantics, the top-frame's dynamic **spork** deque will have the same length as the static **spork** nesting and can be split into a prefix of join tokens and a suffix of b_{spwn} labels and the suffix of b_{spwn} labels is itself a suffix of the static **spork** nesting (and the suffix of b_{spwn} labels is exactly equal to the **spork** deque from the execution in the original semantics). Therefore, the maximum length of the static **spork** nestings of a function corresponds to the maximum length of the dynamic **spork** deque during any execution of that function. Also note that each **spork** occurs at the same index in each static **spork** nesting of which it is a member; this index can be associated with the **spork** and each of its matching **spoin**s.

Using these observations, we can give a realization of the **spork** deque and implementations of **spork** and **spoin** transfers and of promotion. During lowering, when the call stack is made explicit, the backend reserves ***spork** slots*: a contiguous sequence of slots in a function's stack frame equal in length to the maximum length of the static **spork** nestings of the function. At each control-flow point, the dynamic **spork** deque corresponds to the prefix of the **spork** slots with length equal to that of the static **spork** nesting associated with the control-flow point; these are the *active **spork** slots* at that control-flow point. Our invariant is that an inactive **spork** slots is `NULL` and that an active **spork** slot is `NULL` when it corresponds to an unpromoted element of the dynamic **spork** stack and is non-`NULL` when it corresponds to a promoted element. To establish the invariant, the backend 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.

A **spork** transfer is lowered to nothing more than a jump to b_{body}. From the operational semantics, it might appear that a **spork** transfer should be lowered to a write of b_{spwn} to the **spork** slot corresponding to the **spork**'s index (pushing b_{spwn} to the back of the dynamic **spork** deque). This would inform the promotion procedure of the b_{spwn} of the **spork** scope that has been entered. But, writing a (non-`NULL`) b_{spwn} would violate our invariant, since the **spork** slot is transitioning from

[3]This property can be formalized in terms of *dominators* and *post dominators*.

[4]In the compiler IRs, a **spork** is annotated with a unique identifier and each of its matching **spoin**s is annotated with that same identifier.

inactive to unpromoted active. Moreover, the **spork** scopes that have been entered but not exited is statically known for each control-flow point and there is no need to dynamically communicate that information to the promotion procedure.

A **spoin** transfer is lowered to a sequence that reads the **spork** slot corresponding to the **spoin**'s index, compares the read value with NULL, and conditionally branches when true to b_{unpr} and when false to a new block that writes NULL to the slot and jumps to b_{prom} with the read value as an argument. As described earlier, the non-NULL value that is passed to b_{prom} will be (a pointer to) a join token used to obtain the final value from the child thread, although the entire compiler is agnostic to the meaning of the non-NULL value. The write of NULL before jumping to b_{prom} maintains our invariant, since the **spork** slot is transitioning from promoted active to inactive.

Note that these lowerings yield an extremely efficient fast (sequential) path: a **spork** performs only a jump (which is likely to be eliminated by merging the b_{body} block) and a matching **spoin** performs only a read, a comparison, and a conditional branch (to b_{unpr}).

The promotion procedure, implemented in the runtime system, 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 temporaries 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 *frame information* 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 frame pointer with the stack-top pointer and iterates over each frame by reading the return address pointed to by the frame pointer and decrementing the frame pointer by the size recorded in the corresponding frame info until the frame pointer is equal to the stack-bottom pointer.

MPL$^{\mathbf{sp}}$ extends the frame info with the static **spork** nesting (as an array of b_{spwn} labels) of the control-flow point that corresponds to the return address. Based on the invariant for active **spork** slots, the promotion procedure must find the deepest (oldest) frame with NULL active **spork** slots and then find the NULL active **spork** slot with the lowest (oldest) index. In order to distinguish between active and inactive NULL **spork** slots, the promotion procedure uses the length of the frame's static **spork** nesting. Once the promotion procedure has found the correct frame and active **spork** slot, it obtains the b_{spwn} label from the static **spork** nesting at the index corresponding to the found active **spork** slot. The promotion procedure writes the (non-NULL) join token into the found active **spork** slot. Finally, the found frame (including the newly written non-NULL value) is copied to the bottom of a new call stack, b_{spwn} 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.

The lowering of the b_{spwn} block of a **spork** transfer is handled specially. In the variant semantics, the b_{spwn} block is unary and is expected to be executed with a join token as its argument. When lowered, a b_{spwn} block is treated as the return block of a **call** that returns no results. After performing the caller-side of the returning convention, a value is read from the **spork** slot corresponding to the **spork**'s index, NULL is written to that slot (inactivating it, since the b_{spwn} control-flow point is not in any **spork** scope), and execution continues with the read value as the b_{prom} argument.

4.3 Front-End and Closure-Conversion Changes

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

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

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

The *tag*, which must be a compile-time constant, is associated with the **spork** and included in the static **spork** nestings added to frame infos; it is used to communicate a policy that is used at promotion (see Section 4.4). The f_{body} and f_{spwn} functions correspond to the code for the homonymous edges of the introduced **spork**. Instead of a single matching **spoin**, the lowering of `prim_spork_spoin` introduces *two* matching **spoin**s; one **spoin**, with the f_{unprVal} and f_{promVal} functions corresponding to the code for the b_{unpr} and b_{prom} edges, is executed if f_{body} terminates with a value and the other **spoin**, with f_{unprExn} and f_{promExn} for b_{unpr} and b_{prom}, is executed if f_{body} terminates with an uncaught exception. If, during optimization, f_{body} and the functions it calls are inlined (as is often the case), the resulting control-flow graph will **goto** directly from the returning of a value to the value **spoin** and **goto** directly from the raising of an exception to the exception **spoin**. One motivation for this value/exception split is that it would be incorrect for control to leave the **spork** body via an uncaught exception (rather than via a matching **spoin**). We describe a second performance motivation in Section 4.6. The δ argument corresponds to the arbitrary data value stored in the **spork** slot when promoted. Although this data value will always be a join token used for synchronization, making the `prim_spork_spoin` polymorphic with respect to it emphasizes that the compiler makes no assumptions about it and treats it opaquely.

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

4.4 Parallelism Management

While the **MPL**$^{\mathbf{sp}}$ compiler is responsible for the low-level compilation that yields an efficient implementation of **spork** and **spoin** transfers, the thread-scheduling component of **MPL**$^{\mathbf{sp}}$, implemented in source SML code and the runtime system, is responsible for the promotion strategy. **MPL**$^{\mathbf{sp}}$ uses 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 **spork**s on the thread's call stack (with each promotion costing one token), but can be banked if the thread has no promotable **spork**s. Eager spending means that a thread must check for unspent tokens when entering a **spork** scope (and spend one immediately to promote this **spork**); we handle this aspect in Section 4.6. This algorithm guarantees work- and span-efficiency [Westrick et al. 2024]: if a program has work W and span S (excluding the costs of promotions) and a promotion costs τ, then the program will perform at most $\frac{C}{N}W$ promotions and have at most total work $(1 + \frac{C\cdot\tau}{N})W$ and total span $(\tau + N)S$ (including the costs of promotions).

Explicitly counting and checking steps of (non-promotion) work by each thread would be prohibitively expensive; a practical application of heartbeat scheduling approximates work done by the passage of (wall-clock) time. An interval timer delivers a `SIGALRM` to the program with period

N and a signal handler that 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 tokens at a promotion, it has the option of giving some of those tokens to the spawned child (without violating the efficiency guarantees). A **spork** is tagged with a token-sharing policy: either give half of the parent's excess tokens to the child or give all of them. The "inner" **spork** in `reduce` and the **spork** in `par` use the first policy, since the body and the (potential) child thread are typically of comparable work, while the "outer" **spork** in `reduce` uses the second policy, since the remaining loop iterations are expected to be significantly more work than the one current loop iteration. We consider the reverse (when a child with excess tokens joins with its parent) in the next section.

4.5 Work-Stealing Scheduler

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. With work-stealing, the **getjoin** operation first observes, by attempting to pop from the back of the scheduler deque, whether or not the child was stolen.[5] If it was, then a full synchronization with the corresponding **setjoin** must occur to obtain a value from the child. But, if it was not, then the parent can choose how to proceed. It could interpret this as though no promotion happened, in which case it jumps to the b_{unpr} code. This is the choice we make for the "inner" **spoin** in `reduce` and the **spoin** in `par`. But, for the "outer" **spoin** in `reduce`, we execute code similar to the "outer" **spork**'s b_{spwn}, except that it starts the left-half reduction with the accumulator from the **spork**'s b_{body} (i.e., with the accumulator from the now-finished loop iteration that was "interrupted" by the promotion) and terminates with **return** rather than a **setjoin**. Even though the child was not stolen, the fact that a promotion occurred prompts the loop split.

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

4.6 Integration via Source SML Code

A `spork_spoin` function finishes the implementation of the SSA$^{\text{SP}}$ semantics, by performing the necessary integration with the synchronization, parallelism management, and work-stealing components around a use of `prim_spork_spoin`. We must ensure that the f_{spwn} function seen by the primitive ends with a **setjoin** followed by a thread exit and that the f_{promVal} and f_{promExn} functions begin with a **getjoin** (Section 4.1). We must immediately trigger a promotion if the current thread has excess tokens (Section 4.4) and we safely expose the token-sharing policies (Section 4.4). We expose an additional possible code path to be used when a child is spawned by a promotion but is not stolen (Section 4.5). And, we reify (and later propagate) exceptions raised by the execution of a

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

```
fun promote t = runtime_promote (t, newJoin ())
datatype 'a result = Val of 'a | Exn of exception
fun extract res = case res of Val v => v | Exn exn => raise exn
datatype tokshr_policy = GIVE_NONE | GIVE_HALF | GIVE_ALL
fun spork_spoin (policy: tokshr_policy, body: unit -> 'a, spwn: unit -> 'b,
                 seq: 'a -> 'c, sync: 'a * 'b -> 'c, unstolen: 'a -> 'c): 'c =
 let
  fun body' () =
   let val _ = if tokens () > 0 then promote (Thread.current ()) else ()
   in  body ()  end
  fun spwn' (j: 'b join) = let val sr = Val (spwn ()) handle exn => Exn exn
                           in  setJoin (j, sr) ; Thread.exit ()  end
  fun seqVal' bv = seq bv
  fun seqExn' exn = raise exn
  fun syncVal' (bv, j: 'b join) = case getJoin j of
                                    NONE => unstolen bv
                                  | SOME sr => sync (bv, extract sr)
  fun syncExn' (exn, j: 'b join) = (getJoin j ; raise exn)
  val tag = encodePolicy policy
 in
  prim_spork_spoin (tag, body', spwn', seqVal', seqExn', syncVal', syncExn')
 end
```

Fig. 7. `spork_spoin` function that wraps a use of the `prim_spork_spoin` primitive

child thread (giving precedence to exceptions raised by the body). This well-behaved `spork_spoin` function (Figure 7) can be used to robustly implement higher-level parallel operations.

We focus again on the fast (sequential) path that excludes "user code": the (implicit, compiler-implemented) **spork**, execution of `body'` without an eager promotion and excluding `body`, the (implicit, compiler-implemented) **spoin**, execution of `seqVal'` excluding `seq`. Compared to the fast pass described at the end of Section 4.2, this adds only a read of the current thread's tokens (stored as thread-local metadata), a comparison, and a conditional branch.

We also provide a performance reason for `prim_spork_spoin` to handle exceptions. Suppose the lowering of `prim_spork_spoin` only introduced one matching **spoin**. `spork_spoin` would be responsible for ensuring that an exception raised by the **spork** body is propagated across the **spoin**, using reification as with the child thread. `body'` would end with `Val (body ()) handle exn => Exn exn`, which incurs an allocation, and, instead of both `seqVal'` and `seqExn'`, there would be a single `fun seq' br = seq (extract br)`, which incurs a case analysis. Although **MPL$^{\text{sp}}$** employs an efficient bump allocator, even this single allocation and case analysis can add significant overhead to an otherwise non-allocating loop that is executed many times; moreover, these allocations are extremely short lived and can induce additional garbage collections. Although we do not give a detailed evaluation along this dimension in Section 5, we observe that having this allocation and case analysis on the fast path is 1.14x slower on average on both single core and 80 cores.

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

5 Evaluation

We evaluate the performance of **MPL$^{\text{sp}}$** by comparing it against several systems with different implementations of parallel primitives, as shown in Figure 8:

- **MPL$^{\text{sp}}$** (our contribution): automatic parallelism management of both primitives `par` and `reduce`, as described in this paper. No parallelism grain control necessary.

```
fun par (f, g) = [prim]
fun reduce (c, z, i, j, f) = [prim]
```

(a) **MPL^{sp}** has primitives par and reduce implemented as in Figure 5 and 6.

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

(b) MPL^{s}'s binary-splitting implementation of reduce in terms its automatically managed par.

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

(c) MPL's implementation of reduce, which uses eager par_e and requires a manually-tuned grain size GR to be passed at every call-site.

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

(d) MLton's sequential implementations of par and reduce.

Fig. 8. Definitions of par and reduce for the implementations we evaluate in this section.

- MPL^{s} [Westrick et al. 2024]: automatic parallelism management of the primitive par.reduce is implemented by repeatedly splitting with par down to single iterations, as shown in Figure 8b. No parallelism grain control necessary.
- MPL [Arora et al. 2021, 2023]: "eager" primitive $\textbf{par}_e$, which always immediately spawns a new task; reduce is implemented by repeatedly splitting with $\textbf{par}_e$ and switching to sequential fold below a grain size, as shown in Figure 8c. The grain size is tuned manually at every call-site.
- MLton [MLton nd; Weeks 2006]: sequential compiler on which all MPL* versions are based on. The primitives par and reduce are replaced with fast sequential implementations, as shown in Figure 8d.

Except for the presence or absence of manually-tuned grains at each **reduce** call site, all benchmarks use the exact same code, with only the particular implementation used above changing.

In our evaluation, we study three parts:

(1) In Section 5.2, we show that **MPL^{sp}** achieves low overheads relative to sequential MLton on a single core, averaging **1.67x** slower. At the same time, **MPL^{sp}** maintains good parallel scalability, averaging 28x speedup on 80 cores relative to sequential MLton and 46x self-speedup on 80 cores.
(2) In Section 5.3, we demonstrate that compared to manually-tuned parallel code, **MPL^{sp}** needs no manual tuning yet introduces only **1.13x** and **1.26x** overheads on 1 and 80 cores.
(3) In Section 5.4, we find **MPL^{sp}** improves upon MPL^{s} by introducing a new primitive **reduce**, compiled using **spork**, **spoin**, and **setjoin**, getting **1.93x** and **1.61x** faster on 1 and 80 cores.

5.1 Experimental Setup and Benchmarks

Experiments are run on an 80-core machine equipped with two 2.30GHz Intel Xeon (40-core) Platinum 8380 CPUs and 256GB of memory, running Ubuntu 22.04.4 LTS and Linux kernel version 5.15.0-101-generic. We use MLton version 20210117 and MPL^{s}/MPL version 0.5. Benchmark timings are evaluated with a 5 second warmup and then by taking the average of 20 back-to-back runs. For more stable results, we disable hyperthreading and pin experiments to particular cores.

We consider 16 benchmarks from the Parallel ML Benchmark Suite [Arora et al. 2021, 2023; Westrick et al. 2024], covering a variety of problem domains such as graph analysis, computational

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

	MLton	MPL$^{\text{sp}}$		Overhead	Speedup
Benchmark	T_1	T_1	T_{80}	$\frac{T_1(\mathbf{MPL^{sp}})}{T_1(\text{MLton})}$	$\frac{T_1(\text{MLton})}{T_{80}(\mathbf{MPL^{sp}})}$
bfs	2.88	3.10	.092	**1.08**	**31.4**
bignum-add	.404	.857	.015	**2.12**	**26.2**
delaunay	4.91	7.56	.459	**1.54**	**10.7**
grep	1.73	2.38	.043	**1.38**	**40.1**
linefit	.330	1.27	.038	**3.85**	**8.71**
mandelbrot	1.83	2.66	.040	**1.46**	**46.0**
map-heavy	3.42	4.21	.055	**1.23**	**62.1**
map-light	.344	.986	.034	**2.87**	**10.2**
msort	3.42	6.14	.092	**1.79**	**37.2**
nearest-nbrs	.974	1.34	.027	**1.38**	**36.0**
nqueens	1.14	1.46	.022	**1.29**	**51.2**
primes	1.31	2.11	.057	**1.61**	**23.0**
sparse-mxv-csr	1.03	1.76	.037	**1.70**	**28.2**
suffix-array	2.31	2.77	.061	**1.20**	**37.6**
triangle-count	5.35	8.90	.149	**1.67**	**36.0**
wc	.489	1.11	.023	**2.27**	**21.6**
geomean				**1.67**	**27.6**

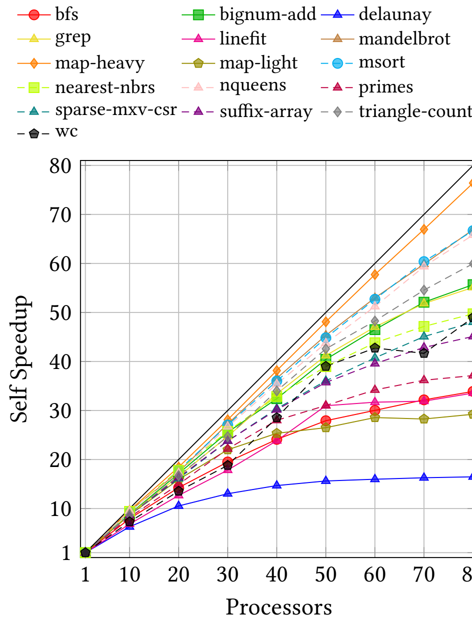

Fig. 9. Self scalability of **MPL$^{\text{sp}}$** to different processor counts. The gray line represents ideal speedup.

geometry, sparse linear algebra, numerical algorithms, and text analysis. In all our experiments, the code for the benchmarks is identical across systems except for the differences shown in Figure 8.

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

We evaluate against MLton to determine (a) the overheads of our approach in comparison to a fast sequential implementation, and (b) the scalability of our approach on increasing number of processors. Note that for this comparison, MLton uses entirely sequential implementations of `par` and `reduce` as shown in Figure 8d.

Table 1 shows our results on 1 and 80 cores (**MPL$^{\text{sp}}$**), alongside the corresponding sequential overheads vs MLton and parallel speedups. The column titled $\frac{T_1(\mathbf{MPL^{sp}})}{T_1(\text{MLton})}$ 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 12 of the 16 benchmarks, **MPL$^{\text{sp}}$** has less than 2x overhead. **MPL$^{\text{sp}}$** also maintains good parallel scalability, with 27.6x speedup on average in comparison to sequential MLton on 80 cores. In Figure 9, we also plot the self-speedup of **MPL$^{\text{sp}}$** across a variety of core counts and observe generally that performance improves as the number of cores increases, with 46x self-speedup on average on 80 cores relative to **MPL$^{\text{sp}}$**'s single-core time. These results demonstrate that our approach is able to maintain high scalability, even without any manual tuning or chunking of parallel loops.

The benchmarks *bignum-add*, *linefit*, *map-light*, and *wc* exhibit larger overheads. These benchmarks are dominated by an extremely tight loop with only a few instructions per iteration, which stresses our approach and magnifies any per-loop overhead. We inspected the code generated for *map-light* and observed that some of the overhead is due to inefficient register allocation, resulting in unnecessary stack spilling on the fast path, which could be avoided with further optimization effort. The primitives **spork** and **spoin** offer new opportunities for compiler optimizations, in

Table 2. Overheads of $\mathbf{MPL^{sp}}$ vs manually-tuned, eager MPL

Benchmark	MPL T_1	MPL T_{80}	Overhead $\frac{T_1(\mathbf{MPL^{sp}})}{T_1(\text{MPL})}$	Overhead $\frac{T_{80}(\mathbf{MPL^{sp}})}{T_{80}(\text{MPL})}$
bfs	3.14	.078	**.988**	**1.17**
bignum-add	.730	.012	**1.17**	**1.32**
delaunay	7.41	.267	**1.02**	**1.72**
grep	2.39	.036	**.992**	**1.18**
linefit	.557	.021	**2.28**	**1.83**
mandelbrot	1.91	.026	**1.40**	**1.53**
map-heavy	4.23	.056	**.996**	**.988**
map-light	1.11	.034	**.886**	**.994**
msort	4.52	.067	**1.36**	**1.37**
nearest-nbrs	1.31	.025	**1.03**	**1.09**
nqueens	1.60	.023	**.914**	**.958**
primes	2.00	.054	**1.06**	**1.05**
sparse-mxv-csr	1.75	.035	**1.00**	**1.05**
suffix-array	5.62	.096	**.493**	**.637**
triangle-count	4.23	.068	**2.10**	**2.17**
wc	.749	.011	**1.48**	**2.10**
geomean			**1.13**	**1.26**

Table 3. Improvement factors of $\mathbf{MPL^{sp}}$ (ours) over MPL^{s}.

MPL^{s} T_1	MPL^{s} T_{80}	Improvement $\frac{T_1(\text{MPL}^{s})}{T_1(\mathbf{MPL^{sp}})}$	Improvement $\frac{T_{80}(\text{MPL}^{s})}{T_{80}(\mathbf{MPL^{sp}})}$
5.93	.154	**1.91**	**1.68**
1.80	.029	**2.10**	**1.85**
7.91	.400	**1.05**	**.872**
5.77	.090	**2.43**	**2.10**
3.28	.057	**2.58**	**1.50**
3.83	.056	**1.44**	**1.41**
3.41	.045	**.810**	**.808**
7.15	.142	**7.26**	**4.20**
6.09	.097	**.993**	**1.06**
1.48	.029	**1.10**	**1.08**
2.31	.034	**1.58**	**1.53**
9.63	.184	**4.55**	**3.23**
6.22	.092	**3.53**	**2.51**
5.84	.111	**2.11**	**1.80**
10.2	.156	**1.14**	**1.05**
2.71	.042	**2.45**	**1.87**
		1.93	**1.61**

particular by identifying performance-sensitive loop bodies and explicitly distinguishing between fast and slow paths. We believe that this information could be exploited in future work to further close the gap between sequential and parallel implementations.

5.3 $\mathbf{MPL^{sp}}$ competes with manually-tuned parallelism

In this experiment, we compare against MPL which uses eager implementations of its primitives and therefore requires manual tuning to amortize the overheads of parallelism. In comparison, $\mathbf{MPL^{sp}}$ removes the need for manual tuning while averaging only **1.13x** and **1.26x** overheads on 1 and 80 cores, respectively. Each of the programs compiled with MPL requires a manually-tuned grain size at each `reduce` call site, specifying the number of loop iterations to allocate to each task for that operation. This is in contrast to the otherwise identical programs compiled with $\mathbf{MPL^{sp}}$, which needs no parallelism grain control and automatically manages task creation at heartbeats. Full results of this comparison are shown in Table 2.

5.4 $\mathbf{MPL^{sp}}$ outperforms par-based automatic parallelism management (MPL^{s})

In this section we compare against MPL^{s} as developed by Westrick et al. [2024], which (similar to our approach) features an automatically managed implementation of **par** based on heartbeat scheduling with low overhead. Their implementation, however, does not automatically manage loop splitting overhead, requiring instead that loops are implemented in terms of **par** as shown in Figure 8b. This incurs splitting overheads on the fast path. In contrast, our $\mathbf{MPL^{sp}}$ automatically manages not just task creation but also the cost of the splitting itself, ensuring that these splitting costs are amortized against heartbeats.

We observe that our $\mathbf{MPL^{sp}}$ is on average **1.93x** and **1.61x** faster than Westrick et al. [2024]'s MPL^{s} on 1 and 80 cores, respectively, as shown in Table 3. The biggest improvements are in the benchmarks that most heavily rely on parallel loops, particularly those with very tight and/or nested loops. For example, on *map-light*, our $\mathbf{MPL^{sp}}$ exhibits 7.26x improvement on a single core; this benchmark simply iterates over a large array and increments every element by 1. Both *primes* and *sparse-mxv-csr* utilize nested parallel loops with tight inner loops, and we observe 4.55x and 3.53x improvement on a single core. Improvements on 80 cores are similar but smaller, which is expected because the additional splitting costs incurred by $\mathsf{MPL}^{\mathsf{s}}$ are all local overheads which parallelize well. Of the 80-core benchmarks, $\mathbf{MPL}^{\mathbf{sp}}$ also outperforms $\mathsf{MPL}^{\mathsf{s}}$ in all but two cases, *map-heavy* and *delaunay*. We inspected *map-heavy* and found instances of inefficient register allocation resulting in unnecessary stack spilling on the fast path, which accounts for the discrepancy.

The *delaunay* benchmark is challenging because it has little theoretical parallelism. 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". While both $\mathbf{MPL}^{\mathbf{sp}}$ and $\mathsf{MPL}^{\mathsf{s}}$ use heartbeat scheduling, our automatically managed implementation of `reduce` in $\mathbf{MPL}^{\mathbf{sp}}$ can take approximately twice as many heartbeats to disperse computation across all processors, due to the implementation of the three-way split: the first promotion generates a new task, but this task then waits for a second promotion to split the remaining iterations in half. Existing work has shown that it is possible to increase the heartbeat rate on stock hardware [Rainey et al. 2021; Su et al. 2024], which if applied in this case would improve scalability by decreasing the delay between successive heartbeats. Nevertheless, even in the case of low parallelism in *delaunay*, $\mathbf{MPL}^{\mathbf{sp}}$ is only 13% slower than $\mathsf{MPL}^{\mathsf{s}}$ on 80 cores.

6 Related Work

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

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

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

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

Implementing heartbeat scheduling in practice requires a low-level pre-emption mechanism (such as software polling [Basu et al. 2021; Feeley 1993b; Ghosh et al. 2020b]) to respond to heartbeats in a timely manner, which can be challenging to incorporate automatically into compiler-generated code without sacrificing sequential efficiency. Early implementations of heartbeat scheduling had minimal compiler support and required significant manual rewriting to ensure efficiency in practice [Acar et al. 2018; Rainey 2023; Rainey et al. 2021]. Recently, Su et al. [2024] demonstrated that heartbeat scheduling is capable of outperforming manual tuning for data-dependent and/or irregular workloads. Their approach places some restrictions on loop bodies (e.g., they do not support nested loops hidden behind a function call) and more generally they do not consider higher-order functions and integration with automatic memory management and scheduling. Our approach is most similar to *automatic parallelism management* [Westrick et al. 2024] which guarantees efficiency and scalability in a high-level fork-join language. This prior work only supports two-way fork-join parallelism, which (as discussed in Section 2) is insufficient to guarantee low overhead in comparison to sequential loops, a limitation which we address in this paper.

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

7 Conclusion

In this paper we present an automatic parallelism management technique for parallel loops and reductions, leveraging heartbeat scheduling to automatically amortize the overheads of splitting a loop into parallel tasks. As a result, we remove the need to manually tune chunk sizes for loops, greatly simplifying code while only introducing mild overheads relative to sequential loops and maintaining high scalability. Our evaluation with a broad set of benchmarks show that the proposed approach extracts excellent performance from parallel codes that make absolutely no effort to control the overhead of parallelism, delivering performance within 25% of manually optimized code across all core counts. These results show that automatic parallelism management techniques may be able deliver a future where performant parallelism requires no programmer involvement.

References

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

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

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

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

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

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

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

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

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

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. https://doi.org/10.1145/3453483.3454107

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

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

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

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

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

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

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

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, 519–538.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Joseph Jaja. 1992. *An introduction to parallel algorithms.* Addison Wesley Longman Publishing Company.

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 249–265. https://doi.org/10.1145/3018743.3018758

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

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

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

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

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

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

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

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

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

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

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

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

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

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