Libfork: portable continuation-stealing with stackless coroutines

C.J. Williams, J.A. Elliott

***Abstract*—Fully-strict fork-join parallelism is a powerful model for shared-memory programming due to its optimal time-scaling and strong bounds on memory scaling. The latter is rarely achieved due to the difficulty of implementing continuation-stealing in traditional High Performance Computing (HPC) languages – where it is often impossible without modifying the compiler or resorting to non-portable techniques. We demonstrate how stackless-coroutines (a new feature in C++20) can enable fully-portable continuation stealing and present *libfork* a lock-free fine-grained parallelism library, combining coroutines with user-space, geometric segmented-stacks. We show our approach is able to achieve optimal time/memory scaling, both theoretically and empirically, across a variety of benchmarks. Compared to openMP (libomp), libfork is on average 7.2× faster and consumes 10× less memory. Similarly, compared to Intel's TBB, libfork is on average 2.7× faster and consumes 6.2× less memory. Additionally, we introduce non-uniform memory access (NUMA) optimizations for schedulers that demonstrate performance matching *busy-waiting* schedulers.**

I. Introduction

SHRINKING transistors are the historic driving-force behind the rapid increase in computing power. Moore's law forecasts an exponential growth in the number of transistors on an integrated circuit [1]; as we reach the physical limits of transistor size, further increases in compute power are achieved by increasing the number of logical cores on a single chip [2]. Programs must embrace parallelism to take advantage of this horizontal scaling.

Parallel computers commonly come in two flavours: shared-memory computers have a single address-space shared by all cores; distributed-memory computers have separate address spaces for each core (normally these are physically separated). Different programming paradigms are appropriate for each type of computer. In this paper we focus on shared-memory parallelism (SMP) which represents almost all modern multi-core computers. Furthermore, distributed-memory computers usually employ two-level parallelism, leveraging SMP within compute nodes and falling back to message-passing between nodes. Therefore, SMP is relevant even for many distributed-memory HPC systems.

Complexities, such as instruction/memory (re)ordering, cache coherency and synchronization overhead, make low-level SMP programming difficult to reason about and error prone. Hence, higher level abstractions – i.e. structured concurrency – have been developed to insulate the programmer from these complexities. Examples include: data parallelism [3], pipeline parallelism [4], task parallelism [5], the actor model [6], etc. Choosing a parallel programming model has a large impact on the performance and complexity of a program.

A parallelism framework expresses the available concurrency of a program, when and where synchronization is needed and, sometimes where code must be executed. It is the job of a scheduler to execute the program in parallel on the physical hardware. If the problem or hardware allow it, the execution can be scheduled offline/statically or even at compile-time, otherwise it must be scheduled online/dynamically at runtime. Scheduling on heterogenous computers almost always requires online scheduling, as does irregular/unpredictable workloads. Therefore, we focus on online scheduling as the more general paradigm.

The remainder of this paper is structured as follows: Section II provides the necessary background; Section III details our C++ library, libfork, and how the operations of continuation stealing can be efficiently mapped to stackless coroutines; in Section IV we evaluate the performance of libfork; finally, in Section V we draw conclusions and discuss future work.

II. Background

The background section is divided into four components: in Section II-A we introduce generic coroutines and C++20's stackless variation; Section II-B introduces the fork-join model of parallelism; in Section II-C we expand upon work-stealing and its associated data-structures finally, in Section II-D we introduce cactus stacks and the challenges they present.

A. Coroutines

Semantically, a coroutine is a function with the ability to pause execution (with the `suspend` operation), which may then be continued at a later time (with the `resume` operation). Coroutines are commonly used in asynchronous programming and cooperative (multi)tasking. They are available in many higher-level programming languages such as Python [7], Golang [8], Kotlin [9], etc.

A full taxonomy of coroutines is presented by Moura and Ierusalimschy [10]. In brief, coroutines can be either stackless or stackfull. A stackfull coroutine (also called a green thread or fibre) is a user space equivalent of an operating system (OS) thread, similarly equipped with its own stack. As stackfull coroutines are scheduled cooperatively, the cost of a context switch is orders of magnitude faster than an OS thread. In contrast, stackless coroutines do not have their own stack. Instead, variables that span a suspension point are stored in a *coroutine frame*. This parallels (but does not replace) the stack frame of a regular function. This means a stackless coroutine cannot `suspend` from within a nested (regular) function call. In general, a coroutine frame must be dynamically allocated/deallocated. The context switch between stackless coroutines can be almost as fast as a bare function call.

1) Coroutines in C++: C++20 introduces stackless coroutines (with both symmetric and asymmetric control transfer mechanisms) however, they expose their `suspend`/`resume` operations indirectly. For the purpose of exposition, a C++20 coroutine:

- Is a function which allocates a (compiler generated) coroutine frame (this allocation can be overridden by the user) at the point of invocation.
- Can `co_await` an *awaitable* object. Effectively, this calls `suspend` on the current coroutine's frame and passes the suspended frame to the awaitable. The awaitable is then free to transfer control to another coroutine (by calling `resume` on a frame) or `return` to the previous `resume` call.
- Terminates with a `co_return` statement which: optionally returns a value via its coroutine frame and then `co_await`(s) a user-defined final-awaitable.

Transferring control from within a suspended coroutine to another suspended coroutine (via a `resume` operation) can use *symmetric-transfer*, this is a guaranteed tail-call, thus consuming no OS stack space.

B. The fork-join model of parallelism

The fork-join (FJ) model [11, 12] is a popular form of structured task-based concurrency used in programming languages and libraries such as: cilk [13], openMP [14], taskflow [15], Intel's TBB [16], nowa [17], fibril [18] and many others.

Within the task-based concurrency paradigm, a task is an abstract unit of work that can be executed concurrently with other tasks. A task may have dependencies on other tasks. The FJ model augments a language with the keywords `fork`[1] (sometime called `spawn`) and `join` (sometime called `sync`). The `fork` keyword creates a new (child) task which may be executed concurrently with the parent task. The `join` keyword signals that all child tasks (children) must be completed before execution of the parent task is allowed to continue. The fully-strict FJ model (SFJ) further imposes that all children are complete before the parent task returns/completes [19]. This constraint appears restrictive, but enables simpler reasoning about the program and stronger bounds on time/memory scaling.

Algorithm 1 Pseudocode for the Fibonacci recurrence, Eq. (1), with fork-join annotations to support parallel execution.

```
1: function FIB(n)
2:     if n < 2 then
3:         return n
4:     x ← fork FIB(n − 1)
5:     y ← FIB(n − 2)
6:     join
7:     return x + y
```

The canonical example of the SFJ model is the Fibonacci function, presented in Algorithm 1, which computes the Fibonacci recurrence [20]:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases} \quad (1)$$

The second recursive call in Algorithm 1 does not use the `fork` keyword as it is immediately followed by a `join`. Hence, the continuation of the parent task would contain no work if a `fork` was added. Adding a `fork` here would not invalidate the program but would add unnecessary overhead.

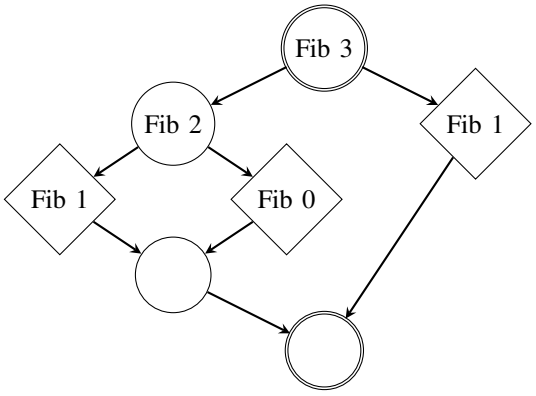


Fig. 1 The DAG representing the execution of the Fibonacci function from Algorithm 1 with argument $n = 3$. Diamond shaped nodes represent leaf tasks/function-calls while circular nodes represent non-leaf tasks, each with a matched join node. Edges represent dependencies i.e. parent-child relationships.

Each SFJ program maps to a directed acyclic graph (DAG) expressing the dependencies between parents and children. This model is useful for reasoning about the time and memory bounds of a program and is a common intermediate representation for schedulers. The DAG for Algorithm 1 is shown in Fig. 1. A scheduler is free to execute any tasks whose dependencies have been satisfied. Normally, the SFJ model is paired with a work stealing-scheduler (see Section II-C) which makes the decision when and where to execute tasks dynamically, as the DAG is built.

The *serial projection* of a SFJ program is a sequential program – generated by removing the `fork`/`join` keywords – which executes the tasks in a depth first traversal of the DAG. The execution time and (stack) memory-consumption of the serial projection are denoted T_s and M_s respectively.

C. Work stealing schedulers

A work stealing scheduler (WSS) maps a program's DAG to a set of cores on a physical machine. Each thread of execution (normally each thread is bound to a physical computation core), called a *worker*, is equipped with a work stealing queue (WSQ) (see Section II-C1 for details). A single worker begins the execution of a program with the *root* task. When a worker encounters a `fork` statement it creates a new task and pushes a task onto its own WSQ. Other workers can steal from this queue (or any other non-empty WSQ) if they have no tasks

[1]Not to confused with the Unix `fork(2)` system call.

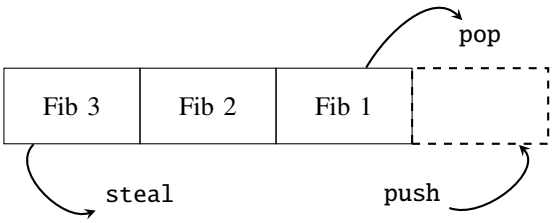


Fig. 2 A diagram of a work stealing queue, the queue contains handles to tasks during a moment of a depth first traversal of the DAG in Fig. 1.

to execute. A greedy WSS with P workers can execute a SFJ program in expected time [21]:

$$T_p \leq \frac{T_1}{P} + \mathcal{O}(T_\infty) \tag{2}$$

where T_1 is the time taken to execute the program with a single worker and T_∞ is the ideal runtime on a machine with an infinite number of workers. This expected time is optimal within a constant factor [21]. In the DAG model, T_∞ corresponds to the longest path through the DAG. An ideal WSS will use stack space [21]:

$$M_p \leq PM_1 \tag{3}$$

where M_1 is the stack space used by a single worker executing the program. This bound can be derived from the DAG, see Blumofe and Leiserson [21] for full details.

1) Work stealing queues: A work stealing queue is a semi-concurrent data structure, which efficiently supports the operations: `push`, `pop` and, `steal`. The `push` and `pop` operations, performed by a single owning worker/thread, add/remove elements of the queue in a first-in-last-out (FILO) order. The `steal` operation can be performed by any worker and removes an element from the queue in a first-in-first-out (FIFO) order. Figure 2 contains a schematic of a WSQ, the `steal` operation may be called concurrently with `push`, `pop` or other `steal` operations.

An efficient WSQ is the backbone of a work stealing runtime; the minimum overhead of a task (compared to a bare function call) corresponds to pushing then popping a task (or task handle) to and from a WSQ. Early implementations of WSQs include the THE [13] queue and the ABP [22] queue. Some papers have explored variations of WSQs that: enable stealing half the tasks a worker owns [23], have split public and private sections [24, 25, 26] and, can store the tasks inline as opposed to pointers/handles to them [27, 28]. These variations all have their merits however, we use a modern version [29, 30] of the Chase-Lev (CL) [31] queue which is fully lock-free, optimized for weak memory models and formally verified [29, 32].

2) Child vs continuation stealing: When a worker forks a task it can choose either to execute the child task and push the *continuation* of the parent task onto it's WSQ or it can push the child task onto its WSQ and continue execution of the parent. The former corresponds to a depth first traversal of the DAG and is called *continuation stealing*, while the latter corresponds to a breadth first traversal and is called *child stealing*. Child stealing is more common, as it is possible to implement as a library in most programming languages [33, 17]. However, child stealing breaks the memory bound, Eq. (3), as a task can have an unbounded number of children and each child will require some memory. Therefore, continuation stealing is the preferable strategy. Continuation stealing can also lead to better cache locality as the child task is likely to use data that the parent task has loaded into memory. Furthermore, continuation stealing preserves the order of execution between a programs's serial projection and its single-worker execution.

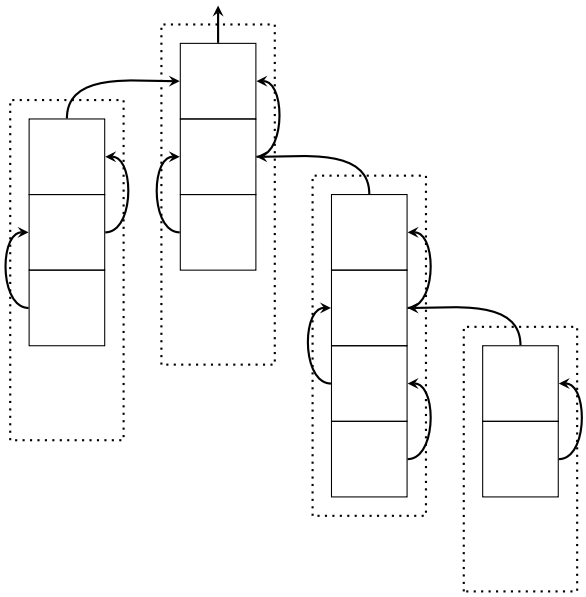


Fig. 3 A diagram of a cactus stack, sometimes called a spaghetti stack, which is an example of a parent pointer tree. Boxes represent (stack) frames. Arrows denote a parent-child relationship. The dotted lines represent regions that could be contiguous segments of memory, i.e. the first child of each parent can be placed on the parents (linear) stack.

D. Cactus stacks

In the context of the SFJ model, each task has an associated *frame* which contains variables local to the task. These are equivalent to traditional stack-frames in the serial projection of a program. A frame may contain pointers into itself hence, it is sensitive to its own location in memory and cannot generally be relocated. A given task may contain pointers to variables in any frame *above* it (i.e. its parent, grandparent, etc.) just like a traditional stack frame. As we are restricted to SFJ the lifetime of a child frame must strictly nest the lifetime of its parent.

For a continuation stealing worker, when no steals are occurring, each frame can be mapped onto a linear stack – as in the serial projection. However, after a worker steals a task, it must allocate the frame of any newly forked children (i.e. not the first child) on a new stack, this produces the branches in Fig. 3. The parent's stack cannot be used by the thief as it may still be in use by another worker, executing some other descendent of the parent. The new stack is linked to the parent stack. The resulting tree-of-stacks data structure is called a

cactus stack [34] and is sketched in Fig. 3. Designing a cactus stack that is simultaneously:

1) Interoperable with a legacy/linear stack
2) Supports a linear-scaling scheduler
3) Provides bounded and efficient memory use

is an open research problem [35, 18]. In Section III-A we present a partial-solution to this problem that forgoes Item 1 in favour of performance and compatibility with stackless coroutines.

III. Libfork

Libfork is pure C++20 library that implements lock-free, wait-free, continuation-stealing using stackless coroutines. The application programmer interface (API) of libfork is designed to mirror Cilk [13] and be as unsurprising as possible. The core API is summarized in Algorithm 2, which presents the fibonacci function from Algorithm 1 expressed with libfork's primitives.

Algorithm 2 The fibonacci function from Algorithm 1 written in C++ with the libfork library. Note: namespace qualifiers and optional decorators have been omitted for brevity. The first argument to a coroutine passes static information through the type-system (e.g. if the task is a root task, the invocation kind, the type of its return address, etc.) and acts as a y-combinator, allowing for recursion within anonymous functions.

```
auto fib = []
  (auto fib, int n) -> task<int> {

    if (n < 2) {
        co_return n;
    }

    int a, b;

    co_await fork[&a, fib](n - 1);
    co_await call[&b, fib](n - 2);

    co_await join;

    co_return a + b;
  };
```

Libfork differs from all other C++/C libraries of its kind because it is a fully-portable, library-only implementation with strong bounds on time and memory scaling, see Section III-E. Libfork utilizes segmented stacks to store tasks (detailed in Section III-A) which allow for unbounded task recursion, without the fear of stack overflows. Additionally, libfork's stacks are exposed to the user to allow for portable use of a (safer) `alloca(3)` equivalent.

Within libfork, a stackless coroutine corresponds to a task. The mapping between the operations of continuation stealing fork-join parallelism and the operations of stackless coroutines in combination with user-space stacks are detailed in section Section III-B.

Finally, libfork is fully NUMA aware and introduces a variation of the adaptive scheduler (see Section III-D) presented by Lin, Huang, and Wong [36] which is able to match the performance of busy-waiting schedulers on NUMA machines.

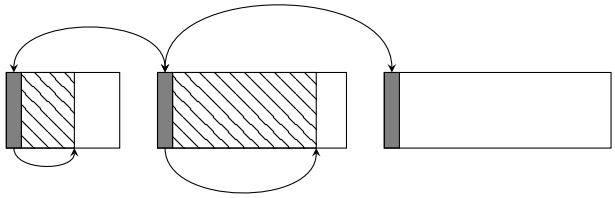


Fig. 4 Diagram (not to scale) of a segmented stack in libfork. The metadata region is filled in gray, hatched regions indicate allocated space, double ended arrows indicate doubly-linked list connections and, single ended arrows represent each stacklets' stack-pointer. This stack is composed of three stacklets, the middle stacklet is the top stacklet, i.e. contains the last allocation. The rightmost stacklet is a cached stacklet, each stack contains zero-or-one cached stacklets.

A. Segmented stacks

As libfork uses continuation stealing, each worker is either a *thief* with no tasks or *active* and there exists a chain of tasks from the root task to the worker's currently executing task. The coroutine frames along this chain, called a *strand*, are linked into a cactus stack.

Figure 3 gives an example of a cactus stack; the most straightforward implementation allocates heap memory for every coroutine frame. This upholds the memory bound and allocations/deallocations can be $\mathcal{O}(1)$ however, heap allocations are comparatively costly operations. Perhaps worse, the strand could be fragmented in memory, leading to inferior cache-locality compared to a traditional linear stack.

Another potential solution is to use a large linear stack, placing the first child of each parent on the parents linear stack and allocating new linear stacks as-required at branch points. This enables the fastest possible allocations/deallocations and the best possible cache locality along the fast-path. The size of the linear stacks would have to be $\mathcal{O}(M_1)$, in-case no branching occurred. Unfortunately, in the worst case, almost every coroutine frame could be placed on a different linear-stack. The memory bound would therefore approach:

$$M_p \leq PM_1^2 \tag{4}$$

As M_1 is $\mathcal{O}(10^6)$ this is much worse than the bound discussed in Section II-C. The memory bound would be restored if the wasted space in the linear stacks could be reclaimed. This is possible with segmented stacks [37] which grow/shrink on demand.

Libfork's utilizes geometric segmented-stacks. A stack is composed of segments of contiguous memory called *stacklets*. Each stacklet starts with 48B of metadata containing pointers: linking the stacklets into a doubly-linked list, tracking the position of the stacklets internal stack-pointer and, marking the region of memory available to the stacklet. If an allocation can fit on the current/top stacklet then an allocation is as fast as a pointer increment. Otherwise, a new stacklet, twice as large as the previous one (or large enough to fit the allocation, whichever is greater), is allocated from the heap. The time for

n consecutive allocations is:

$$nT_{\text{pointer}} + \mathcal{O}\left(\log_2 n\right) T_{\text{heap}} \tag{5}$$

hence, the amortized cost of a single allocation is $\mathcal{O}\left(T_{\text{pointer}}\right)$. If a stacklet becomes empty after a deallocation then it may be cached (if it is not more than twice as large as the previous stacklet). This guards against hot-splitting [37]. The allocation and deallocation hot paths are identical to a linear stack, the additional instruction cost is loading more pointers and a predictable branch.

B. Continuation stealing with stackless coroutines

1) Fork or call: The call to the `fork` function in Algorithm 2 creates a new child task, which will recursively call `fib` with a new argument and write the results to the variable a in the parent task. The `fork` function generates an awaitable who's actions are specified in Algorithm 3. The thread-local objects (the stack and WSQ) are stored in `thread_local` variables, such that they can be accessed without returning control to the scheduler. Only the push to the WSQ on Line 7 requires atomic/synchronized operations. The call-awaitable, generated by the `call` function in Algorithm 2, is identical to Algorithm 3 except the push to the WSQ is omitted. The tail-call is achieved through symmetric-transfer as the child, c, is constructed in the suspended state. In general, the child task *may* be allocated on a different frame than the parent.

Algorithm 3 Pseudocode for the fork-awaitable, return address handling omitted for brevity.

Require: $p \leftarrow$ the parent's suspended frame.
Require: $f \leftarrow$ a child function.
Require: $x \ldots \leftarrow$ arguments for f.
1: **function** AwaitFork(p, f, $x \ldots$)
2: $g \leftarrow$ the **thread-local** stack
3: $s \leftarrow$ allocate space for child on g
4: $c \leftarrow$ invoke f with $x \ldots$ (frame at s on g) ▷ Child
5: Set p as the parent of c
6: Set g as the stack of c
7: Push p onto the **thread-local** WSQ
8: **tailcall** Resume(c) ▷ Execute child

Algorithm 4 Pseudocode for the join-awaitable.

Require: $c \leftarrow$ a suspended coroutine frame.
1: **function** AwaitJoin(c)
2: **if** this task has not been stolen **then**
3: **tailcall** Resume(c) ▷ already own c's stack
4: **atomically**
5: Mark c as at-a-join-point
6: Decrease c's join counter
7: **if** this was the last worker to join c **then**
8: $g \leftarrow$ the **thread-local** stack
9: $g_c \leftarrow$ the stack of c
10: $g \leftarrow g_c$ ▷ take c's stack
11: **tailcall** Resume(c)
12: **return** ▷ to scheduler,

2) Join: The join-awaitables's actions are described in Algorithm 4. We use the split counter method of nowa [17] to implement the operations in the atomic block, Line 4, with no locks. If the worker is the last to join, it takes ownership of the child's stack on Line 10; before taking ownership the workers previous stack, g, is empty. We implement an additional shortcut/optimization before entering Algorithm 4 to prevent unnecessary suspension of the parent task when no steals have occurred or all children are already complete. All tail-calls are achieved through symmetric-transfer.

Algorithm 5 Pseudocode for the final-awaitable, return-value handling omitted for brevity.

Require: $c \leftarrow$ a suspended coroutine frame.
1: **function** AwaitReturn(c)
2: $p \leftarrow$ the parent of c
3: $g \leftarrow$ the **thread-local** stack
4: $g_p \leftarrow$ the stack of p
5: Deallocate c from g
6: **if** p is null **then** ▷ i.e. c is a root task
7: **return** ▷ to scheduler
8: **if** c was called **then**
9: **tailcall** Resume(p)
10: **if** try pop from thread local WSQ **then**
11: **tailcall** Resume(p) ▷ hot-path
12: **atomically** ▷ implicit join
13: Check if p is at-a-join-point
14: Decrease p's join counter
15: **if** p was at-a-join-point **then**
16: **if** this was the last worker to join p **then**
17: **if** $g \neq g_p$ **then**
18: $g \leftarrow g_p$ ▷ take p's stack
19: **tailcall** Resume(p)
20: **if** $g = g_p$ **then**
21: $g \leftarrow$ new empty stack ▷ release p's stack
22: **return** ▷ to scheduler

3) Cooperatively returning: The final-awaitables's actions are described in Algorithm 4. The first and second `if`-statements can be resolved at compile time (using the static information discussed in Algorithm 2) and thus have no runtime-overhead. Again, the split counter method [17] is used to implement the atomic block with no locks. If the worker fails to pop the parent from its WSQ, it must perform an *implicit join*, potentially transferring execution back to the parent. If the implicit-join succeeds and the worker does not already own the parents stack, the worker takes ownership of the parents stack on Line 18. Before taking ownership, the workers previous stack, g, is empty. Otherwise, if the implicit-join fails, the worker may need a new stack. If the worker does need a new stack, the old one is released on Line 21 (some worker will eventually take ownership of it when resuming the parent). Once again, all tail-calls are achieved through symmetric-transfer.

C. Stack allocation API

A fork-join scope is the region between the first `fork` and its corresponding `join`. Outside a fork-join scope, a worker always "owns" the stack a coroutine lives on. Hence, they can allocate/deallocate from it – as long as they preserve FILO ordering and strictly nest the lifetime of the allocations within the coroutine's lifetime. This is useful for allocating temporary memory for storing the return values used in a fork-join scope, e.g. a buffer for a parallel reductions partial sums. This is a portable alternative to `alloca(3)`, which is not available on all platforms and, due to the use of segmented stacks, will never overflow the stack.

D. Scheduling and NUMA

Libfork's workers are NUMA aware. They use hwloc [38] to determine the NUMA topology of a machine, which is represented as a tree with physical CPU cores at its leaves. The topological distance between two leaves/cores is the maximum of the distances between each leaf and their common ancestor. Cores that are (topologically) closer have faster access to common shared memory. In libfork, each worker is pinned to a core. When a worker attempts to steal a task from a *victim*, the victim is selected with a probability proportional to:

$$w_{ij} = \frac{1}{n_{ij} r_{ij}^2} \tag{6}$$

where $r_{ij} \in \mathbb{Z}$ is the topological distance between cores i and j and, n_{ij} is the number of cores separated by r_{ij}.

Libfork supplies two default schedulers[2], both of which are greedy [21]. The *busy* scheduler's workers continuously attempt randomized stealing, selecting their victims according to Eq. (6). This minimizes latency at the cost of high CPU usage, even when workers have no tasks. The *lazy* scheduler is a simple variation of the adaptive scheduler presented by Lin, Huang, and Wong [36]. We separate workers into groups as determined by the NUMA node they are pinned to. When at-least one worker is active globally then within each group – as opposed to globally as suggest by Lin, Huang, and Wong [36] – at least one worker is kept awake and attempts randomized stealing. The remaining workers in each group can sleep when/if they find no work. All the workers select their victim according to Eq. (6). This trades potentially-higher latency for lower CPU usage when the available parallelism is low. Keeping one worker awake per NUMA node reduces cross-node stealing.

1) Explicit scheduling: Libfork's workers each maintain a WSQ and a lock-free single-consumer multi-producer *submission queue*. This means there is no global submission-queue – which is often employed to submit root-tasks to the pool of workers. As each task in libfork is a coroutine, workers can use these submission queues to perform *explicit-scheduling*; if a task requests to be run on a particular worker then it can be suspended and ownership transferred by pushing a handle onto the requested workers submission queue. This is desirable for certain runtimes (e.g. MPI [39]) which may require a specific thread to interact with them.

[2]Libfork's extension-API supports user customization of schedulers.

E. Theoretical bounds

Definition 1. A stacklet has a stack of size k and a metadata region of size c. All "sizes" are in bytes unless specified otherwise.

Theorem 1 (Segmented stack overhead)**.** *A segmented stack storing $M \geq 1$ bytes has a worst-case size of $\mathcal{O}(c) + c\log_2(M) + 4M$.*

Proof. Each allocation must be at-least one byte. There exists $n+1$ stacklets and n can always be made greater than zero by adding a cached (empty) stacklet. The first n stacklets must contain at least one allocation each. The wasted space on each of the first n stacklets is at most one-less than the size of the requested (stack) allocation that triggered the (heap) allocation of the next stacklet. Hence, the total wasted stack-space on the first n stacklets is at most M. Furthermore, the total used stack-space on the first n stacklets is at most M if the last stack is empty. Therefore, the total stack size of the first n stacklets is at most $2M$. Summing over the minimal stack-size of the first n stacklets:

$$1 + 2 + 4 + \ldots + 2^{n-1} \leq 2M \tag{7}$$

hence, the maximum value of n is:

$$n \leq \lfloor \log_2(2M+1) \rfloor \tag{8}$$

The last stacklet can be at-most twice as large as the previous stacklet hence, its worst case size is $2M$. Therefore, including the metadata overhead, the total memory used by the segmented stack, M', is at most:

$$\begin{aligned} M' &\leq (n+1)c + 2M + 2M \\ &\leq +c\log_2(2M+1) + 4M \\ &\leq \mathcal{O}(c) + c\log_2(M) + 4M \end{aligned} \tag{9}$$

QED

Lemma 1. *Given a path through a DAG composed of n tasks each consuming memory m_i, the total memory consumed by a strand along that path is at-worst $M \leq 2nc + 3(m_1 + \ldots + m_n)$.*

Proof. Tasks along a strand are stored on a chain of segmented stacks. The gradient of the result of Theorem 1 with respect to M is monotonically decreasing hence, the largest memory overhead occurs when each task is stored on it's own stack. Therefore, each task requires at-worst:

$$(c + m_i) + (c + 2m_i) = 2c + 3m_i \tag{10}$$

bytes of memory for a three stacklet stack. Therefore, the memory consumed by the strand along this path is at-worst:

$$\begin{aligned} M_{\text{path}} &\leq (2c + 3m_1) + \ldots + (2c + 3m_n) \\ &\leq 2nc + 3(m_1 + \ldots + m_n) \end{aligned} \tag{11}$$

QED

Definition 2. Let M_1 be the sum of the task sizes along the path through the DAG that maximizes such a sum, i.e. M_1 is

the maximum (stack) memory usage of a program executed by a single worker on a linear stack with no overhead.

Lemma 2. *The longest path through a programs's DAG contains at-most $N \leq M_1$ tasks.*

Proof. The smallest task must consume at-least 1 byte of memory. The longest path must consume less memory than the path that uses M_1 memory, hence:

$$\sum_{i=1}^{N} 1 \leq M_1 \tag{12}$$

QED

Theorem 2 (Stack memory bound)**.** *Libfork will use at most $M_p \leq (2c+3)\,PM_1$ stack memory executing a program with P workers.*

Proof. Due to symmetric transfer, each worker uses no OS-stack space when transferring control between tasks hence, a constant amount of OS-stack space per worker is used. In the worst-case, the path through the DAG that uses M_1 memory is also the longest path in the DAG hence, applying Lemma 1 and Lemma 2, all paths use less than:

$$\begin{aligned} M_{\max} &\leq 2nc + 3(m_1 + \ldots + m_n) \\ &\leq 2Nc + 3M_1 \\ &\leq (2c+3)\,M_1 \end{aligned} \tag{13}$$

bytes of memory. Libfork maintains the *busy-leaves property* i.e. "at every time step, every living [task] that has no living descendants has a processor working on it" [21]. Therefore, every worker consumes less memory than the path through the DAG that consumes the most memory. Hence, the worst-case memory usage of a program with P workers is:

$$\begin{aligned} M_p &\leq PM_{\max} \\ &\leq (2c+3)\,PM_1 \end{aligned} \tag{14}$$

QED

Our final memory bound is very loose, the constant multiplier $(2c+3)$ is dominated by the $2c$ contribution from the metadata overhead. In practice, this contribution is negligible as the average task size is a few hundred bytes and most stacklets store several tasks.

IV. Experimental Evaluation

We evaluated libfork with two sets of benchmarks. The first is a representative subset of the classical FJ/work-stealing benchmarks [13, 18, 17] and the second is the unbalanced tree search (UTS) benchmark family [40]. The benchmarking parameters are detailed in Table I.

We compared libfork to Intel's TBB [16] and taskflow [15] which – like libfork – are both pure, portable library-implementation of FJ parallelism. We also made comparisons with openMP [14] (specifically the libomp implementation from the LLVM project [41]); as a custom compiler front-end, openMP could access global-optimizations not accessible to the other libraries hence, it is not fully comparable. However, as it has been integrated into the major C++ compilers (GCC, Clang, ICC and, MSVC), we decided to include it for reference.

TABLE I Summary of benchmark parameters. For the UTS benchmarks, a *geometric* tree has $t = 1$ and $a = 3$ while a *binomial* tree has $t = 0$ and $b = 2000$. Unless otherwise specified, all other UTS parameters are defaulted.

Name	Description	Parameters
fib	Recursive Fibonacci	$n = 42$
integrate	Numerical integration	$n = 10^4, \epsilon = 10^{-9}$
matmul	Matrix × Matrix	$n = 8192$
nqueens	N-queens problem	$n = 14$
T1	Small geometric tree	$d = 10, b = 4, r = 19$
T1L	Large geometric tree	$d = 13, b = 4, r = 29$
T1XXL	Huge geometric tree	$d = 15, b = 4, r = 19$
T3	Small binomial tree	$q = 0.124875, m = 8, r = 42$
T3L	Large binomial tree	$q = 0.200014, m = 5, r = 7$
T3XXL	Huge binomial tree	$q = 0.499995, m = 2, r = 316$

A. Benchmark methodology

All benchmarks were run on a NUMA machine with two Intel(R) Xeon(R) Platinum 8480+ CPUs at 2.00 GHz with boost enabled up to 3.8 GHz. Each CPU socket had its own NUMA node and 56 cores, totaling 112 cores. The total available RAM was 500 GiB split over both sockets. The benchmarking code was compiled using Clang version 18.0.0[3] with the highest optimization setting and runtime asserts disabled. We used the libomp distribution bundled with Clang, libfork version 3.5.0[4], taskflow version 3.6.0 and, TBB version 2021.10.0.

Timings were performed using the Google benchmark library. All benchmarks were repeated until a minimum time had elapsed. Measurements of the maximum resident set size (MRSS) during the execution of a benchmark were performed with the GNU `time` utility. These are quantized to 4 KiB increments. Unless otherwise stated, all benchmark were run 5 times. We report the median and standard deviation of these measurements.

B. Results

1) Execution time: The results of the execution time for the benchmarks are presented in Fig. 5 and Fig. 6. The parallel speedup of a program is defined as:

$$\text{Speedup} = \frac{T_s}{T_p} \tag{15}$$

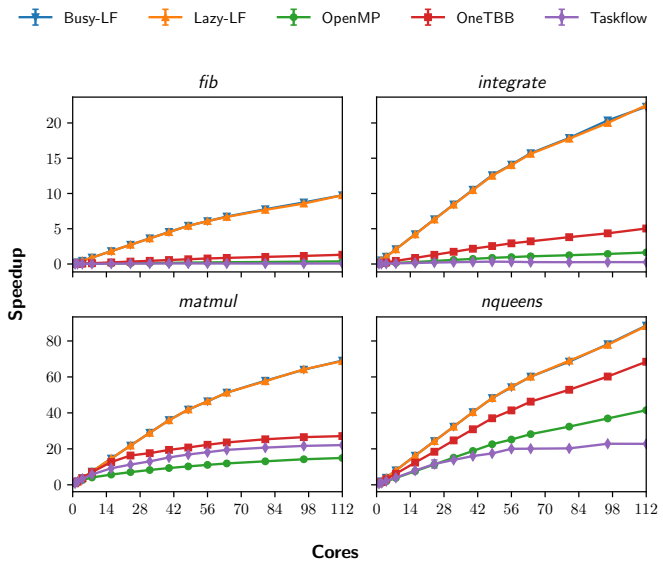
Following Eq. (2), for a program with sufficient parallelism (i.e. $T_1 \gg T_\infty$) the speedup is expected to be linear. Additionally, the parallel efficiency is defined as:

$$\text{Efficiency} = \frac{\text{Speedup}}{P} \tag{16}$$

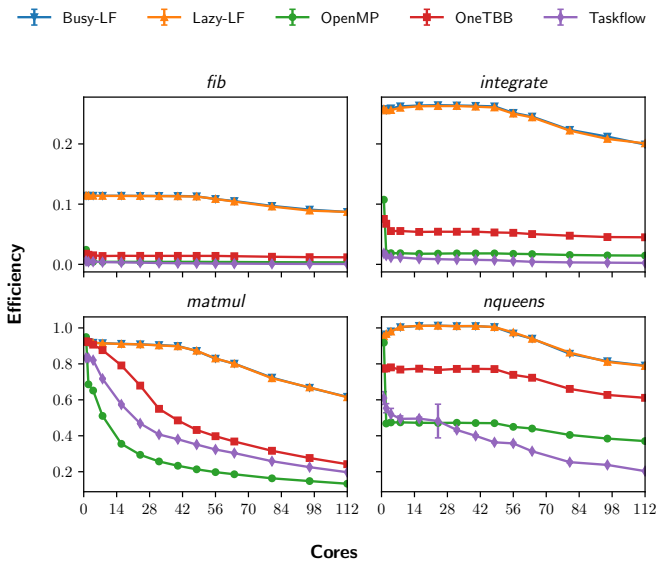
which, for a linear scaling framework, should approach one as T_1 approaches T_s. At $P = 1$, the efficiency is equal to T_s/T_1,

[3]Using commit `c4b795df8075b111fc14cb5409f7138c32313a9b`, from the source available at https://github.com/llvm/llvm-project.git

[4]Using commit `8e4ab70ee1c8b200a3e89fd467b7acadbb7c6d9e`, from the source available at https://github.com/ConorWilliams/libfork.git



(a) Parallel speedup – Eq. (15).



(b) Parallel efficiency – Eq. (16).

Fig. 5 Classical benchmarks, here *Busy-LF* and *Lazy-LF* refer to libfork's busy and lazy schedulers respectively.

which directly measures the overhead of a framework in the absence of communication interference, e.g. stealing, cache-thrashing, lock-contention, etc.

2) Memory consuption: The results of the memory consumption during the benchmarks are presented in Fig. 7. We fit the data to the power law:

$$\text{MRSS} \approx a + bM_1P^n \tag{17}$$

with fitting parameters a, b and n. The fitted exponents are presented in Table II.

C. Discussion

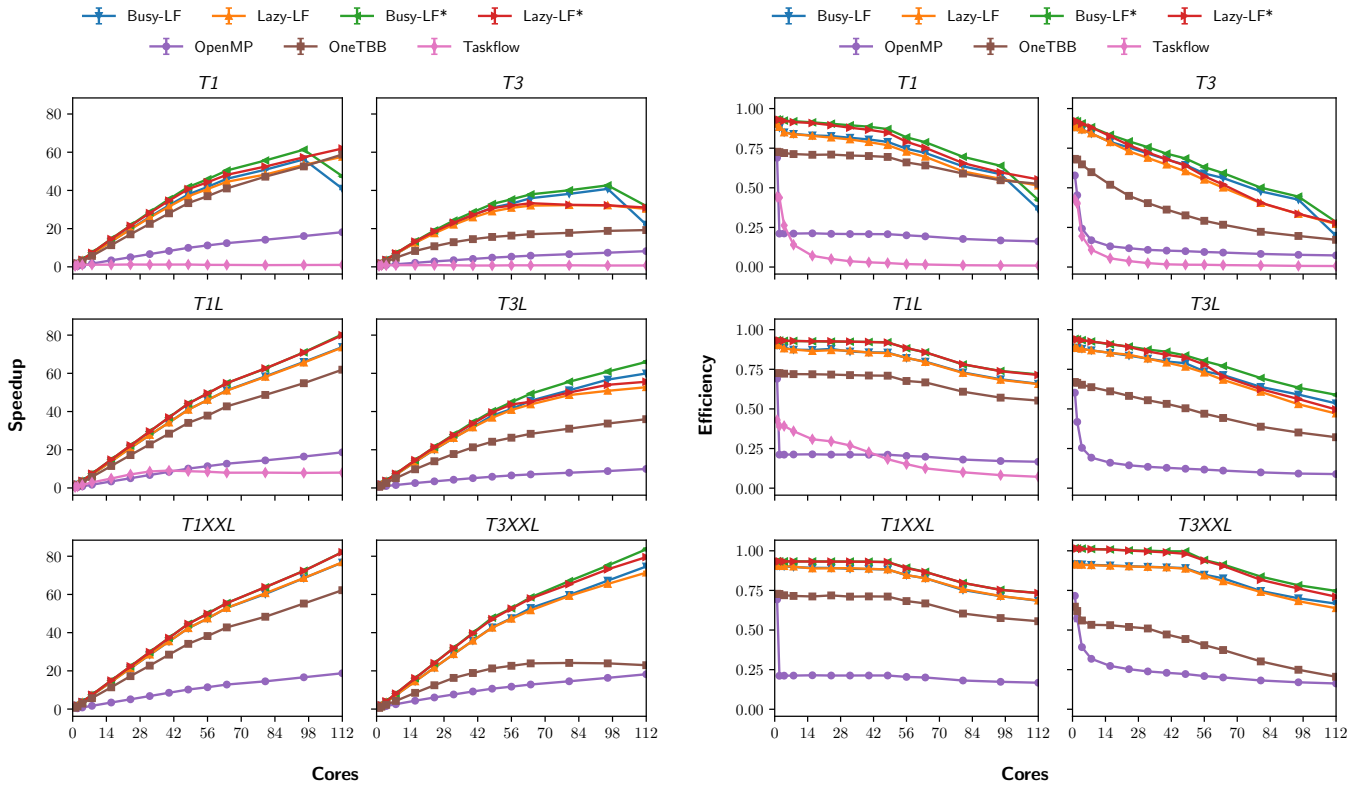
All computations are fully-strict and libfork is a greedy scheduler that maintains the busy-leaves property [21] hence, libfork's execution time should scale as Eq. (2). This linear scaling is apparent in almost all the benchmarks up to around 56 cores (half the total). After this point a second linear scaling continues but with a shallower gradient. This is probably due to the CPU clock-boost throttling back; as more of the cores become active the CPU is no-longer able to maintain the boosted clock speeds due to the additional thermal load.

1) Classic benchmarks: Across all the classic benchmarks, libfork's lazy and busy schedulers have almost identical performance. This is to be expected as the classic benchmarks spawn a large number of tasks. These tasks are easily distributed across the available cores. Their recursive structure means a thief always steals the largest task a victim has available, hence, a lazy worker is unlikely to ever sleep.

a) Fibonacci and integration: We see libfork significantly outperforms the other libraries in the Fibonacci and integration benchmarks. This is highlighted at 112 cores where libfork is 7.5× and 4.5× faster then TBB and 24× and 14× faster than openMP. These are very fine-grained benchmarks that predominantly-test scheduling overhead. In particular, for Fibonacci (which has only a few instructions per-task), the overheads, T_1/T_S, where 8.8, 41, 57 and, 180 for libfork, openMP, TBB, and, taskflow respectively. This cannot be due to libfork's WSQ implementation as the same one is used in taskflow [36]. Instead, the overhead of creating a libfork task is much lower; this could be dominated by heap allocations in the other libraries.

TABLE II Fitted exponential, n, from Eq. (17) and the data in Fig. 7; errors are estimated from the fit's covariance matrix.

Benchmark	Lazy-LF	Busy-Lf	TBB	OpenMP	Taskflow
Recursive Fibonacci	0.86 ± 0.08	0.93 ± 0.06	1.06 ± 0.03	1.20 ± 0.10	0.00 ± 0.03
Numerical integration	0.96 ± 0.10	0.94 ± 0.06	1.04 ± 0.03	1.07 ± 0.09	0.00 ± 0.03
Matrix × Matrix	0.88 ± 0.08	1.09 ± 0.11	1.07 ± 0.03	1.04 ± 0.13	0.00 ± 0.08
N-queens problem	0.94 ± 0.03	1.05 ± 0.07	1.11 ± 0.03	1.30 ± 0.07	0.00 ± 0.37
T1	1.06 ± 0.05	1.04 ± 0.03	0.78 ± 0.02	1.18 ± 0.06	0.46 ± 0.14
T1L	1.08 ± 0.05	0.99 ± 0.06	0.93 ± 0.06	1.23 ± 0.08	0.99 ± 0.02
T1XXL	0.86 ± 0.10	0.90 ± 0.03	0.92 ± 0.03	1.08 ± 0.07	-
T3	0.83 ± 0.05	0.87 ± 0.05	0.67 ± 0.03	0.90 ± 0.03	0.64 ± 0.15
T3L	0.43 ± 0.03	0.52 ± 0.03	1.00 ± 0.03	1.01 ± 0.01	-
T3XXL	0.38 ± 0.01	0.36 ± 0.01	0.86 ± 0.01	1.01 ± 0.01	-



(a) Parallel speedup – Eq. (15).

(b) Parallel efficiency – Eq. (16).

Fig. 6 UTS benchmarks, here *Busy-LF* and *Lazy-LF* refer to libfork's busy and lazy schedulers respectively. The libfork benchmarks marked with a '*' used a modified algorithm utilizing libfork's stack allocation API (see Section III-C) instead of heap-allocating space for return values. The taskflow benchmarks for T3L, T1XXL and T3XXL exhausted the available memory (500 GiB) and could not run to completion.

b) Matrix multiplication: The matrix multiplication benchmark computes the product of two matrices whose size was $\mathcal{O}$(GiB) (much larger than the CPU caches) hence, cache locality and NUMA are important considerations. As the only continuation stealer, libfork scales much better than the other candidates. This is particularly apparent in Fig. 5b, where none of the other candidates manages a horizontal efficiency.

c) N-queens problem: The n-queens benchmark is perhaps the easiest to schedule as each task contains a substantial amount of work while the cache locality concerns are much less than the matrix multiplication benchmark. We see most of the frameworks manage linear scaling but, libfork still outperforms them, probably due to its reduced scheduling overhead.

d) Memory scaling : The memory profiles (Fig. 7) for the Fibonacci, integration and n-queens benchmarks all follow a similar pattern. Strikingly, taskflow consistently consumes 2–4 orders-of-magnitude more memory than libfork. This could be due to taskflows's internal caching of the (exponentially) many tasks produced by the recursive benchmarks. Furthermore, looking at the fitted exponents in Table II, we see taskflow seems to allocate all these tasks regardless of the amount of parallelism. OpenMP and TBB perform better but still consume up to 12× and 3.1× more memory than libfork. This is probably due to a combination of libfork's segmented stacks reducing wasted memory and potentially a smaller metadata overhead per-task. The fitted exponents for libfork are all less than 1, confirming that libfork's memory scaling is better than its theoretical bound. The sub-linear scaling is probably due to some stack space being shared between cores. Contrastingly, TBB consistently scales with an exponent just above 1 and openMP has exponents as high as 1.3. Some of this could be due to the `malloc` implementation fragmenting the task allocations but, this is likely a consequence of child-stealing. Libfork's coefficient b in Eq. (17) never exceeds 0.2, this is much less than the theoretical bound, $2c+3 = 99$, of Theorem 2, which further reinforces the looseness of the bound.

2) UTS benchmarks: The geometric UTS benchmarks (i.e. the T1 family) form a similar tree structure to the classic benchmarks; that is, the expected size of a subtree increases with proximity of the subtree's root to the root. The binomial UTS benchmark (i.e. the T3 family) has irregular, self-similar subtrees and is "an optimal adversary for load balancing strategies, since [...] the expected work at all nodes is identical" [40].

a) Small trees: The T1 and T3 benchmarks are relatively small benchmarks, taking only 10ms and 13ms to execute with libfork at 112 cores. This explains libfork's sub-linear – sometimes negative – time scaling and low efficiency in Fig. 6;

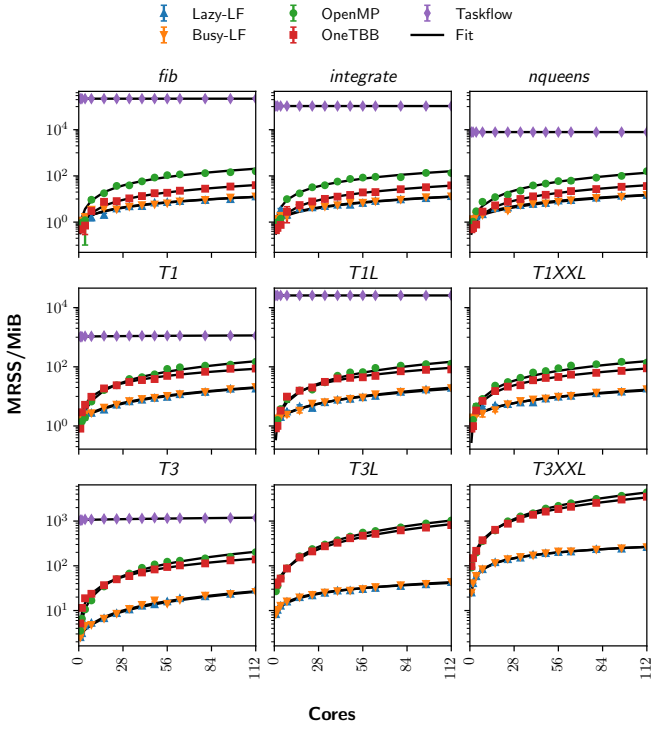


Fig. 7 Peak memory consumption during the benchmarks, fit to Eq. (17). The taskflow benchmarks for T3L, T1XXL and T3XXL exhausted the available memory (500 GiB) and could not run to completion. The matrix multiplication benchmark has been excluded as the MRSS is dominated by the allocation of the input/output matrices.

cores run out of work and attempt spurious stealing of small tasks, which adds considerable overhead. Interestingly, the lazy scheduler mitigates this – by sleeping the workers that cannot find work – and does not exhibit the negative scaling. Notably, TBB handles this work-starved environment well.

b) Geometric trees: For the remainder of the T1 family, libfork demonstrates similar linear scaling as in the classical benchmarks because, the underlying program's DAGs have a similar structure. As the recursion depth is similar across the geometric trees, the memory consumption remains roughly constant for libfork, openMP and TBB. Again, taskflow continues to allocate enough memory for all the tasks ever-spawned, this eventually exhausts the system's memory for T1XXL and terminates the benchmark.

c) Binomial trees: Despite the increased difficulty in load balancing, libfork demonstrates linear time-scaling for the remainder of the binomial family. The T3L benchmark continues to trigger frequent steals, incurring overhead, and resulting in a lower efficiency. The binomial trees have a larger recursion depth hence, requiring more memory. As with the T1 family, taskflow exhausts the available memory and fails to complete the benchmarks. Libfork does a much better job at minimizing allocations; in the T3XXL benchmark, libfork requires 13× less memory than TBB and 17× less memory than openMP, despite delivering at least a 3.6× speedup over both. In Table II, we see the fitted exponents dropping to much less than one. This is because the depth of each leaf node in the DAG is binomially-distributed hence, the average worker will be using much less than the maximum stack space.

d) Stack allocation API: The measurements annotated with a '*' in Fig. 6 used a modified algorithm, utilizing libfork's stack allocation API (see Section III-C). This brings a small performance enhancement in all the UTS benchmarks, as it reduces the number of heap allocations and increases cache locality.

V. Conclusions

We have shown how the operations of continuation-stealing fork-join parallelism can be mapped to the primitives of stackless coroutines and developed libfork: a lock-free, wait-free fine grained, NUMA aware, fully decentralized (no global queues), weak-memory-model optimized, parallelism library. By utilizing C++20's coroutines libfork is, to the best of the author's knowledge, the only fully-portable continuation-stealing C++ tasking library. Libfork achieves linear time-scaling, performing up to: 7.5× faster than Intel's TBB, 24× faster than openMP (libomp) and, 100× faster than taskflow. Similarly, libfork achieves linear memory-scaling, through its integration of user-space segmented stacks to store coroutine frames, consuming up to: 19× less memory than Intel's TBB, 24× less memory than openMP (libomp) and, several orders-of-magnitude less memory than taskflow.

Libfork is an ongoing development; coroutines are a relatively new addition to the C++ language. Hopefully, future language additions, such as asynchronous resource-acquisition-is-initialization (RAII), will allow for usability and performance enhancements. Future compiler optimizations, particularly work on the heap-allocation-elision-optimization (HALO) which could completely elide allocations for non-recursive coroutines, may bring library-based solutions on-par with language level solutions. For now, libfork enables programmers to expose finer-grained parallelism than previously possible, a necessary step to fully utilizing available hardware.

Artifacts

The source code for libfork and its benchmark suite are available online: https://github.com/ConorWilliams/libfork

Thanks

We would like to thank Patrick R.L. Welche for reviewing the manuscript and providing stimulating discussions.

Acknowledgements

We gratefully acknowledge the funding received from the EPSRC via the CDT in Computational Methods for Materials Science (Grant number EP/L015552/1). We also acknowledge Rolls-Royce Plc for the provision of funding. All information and foreground intellectual property generated by this research work is the property of Rolls-Royce Plc.

References

[1] Gordon E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. ISSN: 1098-4232. DOI: 10.1109/n-ssc.2006.4785860. URL: http://dx.doi.org/10.1109/N-SSC.2006.4785860.

[2] Laszlo B Kish. "End of Moore's law: thermal (noise) death of integration in micro and nano electronics". In: *Physics Letters A* 305.3–4 (Dec. 2002), pp. 144–149. ISSN: 0375-9601. DOI: 10.1016/s0375-9601(02)01365-8. URL: http://dx.doi.org/10.1016/S0375-9601(02)01365-8.

[3] W. Daniel Hillis and Guy L. Steele. "Data parallel algorithms". In: *Communications of the ACM* 29.12 (Dec. 1986), pp. 1170–1183. ISSN: 1557-7317. DOI: 10.1145/7902.7903. URL: http://dx.doi.org/10.1145/7902.7903.

[4] I-Ting Angelina Lee et al. "On-the-Fly Pipeline Parallelism". In: *ACM Transactions on Parallel Computing* 2.3 (Sept. 2015), pp. 1–42. ISSN: 2329-4957. DOI: 10.1145/2809808. URL: http://dx.doi.org/10.1145/2809808.

[5] Peter Thoman et al. "A taxonomy of task-based parallel programming technologies for high-performance computing". In: *The Journal of Supercomputing* 74.4 (Jan. 2018), pp. 1422–1434. ISSN: 1573-0484. DOI: 10.1007/s11227-018-2238-4. URL: http://dx.doi.org/10.1007/s11227-018-2238-4.

[6] Gul Agha and Carl Hewitt. "Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming". In: *Research Directions in Object-Oriented Programming*. Cambridge, MA, USA: MIT Press, 1987, pp. 49–74. ISBN: 0262192640.

[7] Michel F. Sanner. "Python: a programming language for software integration and development." In: *Journal of molecular graphics & modelling* 17 1 (1999), pp. 57–61. URL: https://api.semanticscholar.org/CorpusID:12160699.

[8] Rob Pike. "Go at Google". In: *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. SPLASH '12. ACM, Oct. 2012. DOI: 10.1145/2384716.2384720. URL: http://dx.doi.org/10.1145/2384716.2384720.

[9] Roman Elizarov et al. "Kotlin coroutines: design and implementation". In: *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. SPLASH '21. ACM, Oct. 2021. DOI: 10.1145/3486607.3486751. URL: http://dx.doi.org/10.1145/3486607.3486751.

[10] Ana Lúcia De Moura and Roberto Ierusalimschy. "Revisiting coroutines". In: *ACM Transactions on Programming Languages and Systems* 31.2 (Feb. 2009), pp. 1–31. ISSN: 1558-4593. DOI: 10.1145/1462166.1462167. URL: http://dx.doi.org/10.1145/1462166.1462167.

[11] Melvin E. Conway. "A multiprocessor system design". In: *Proceedings of the November 12-14, 1963, fall joint computer conference on XX - AFIPS '63 (Fall)*. AFIPS '63 (Fall). ACM Press, 1963. DOI: 10.1145/1463822.1463838. URL: http://dx.doi.org/10.1145/1463822.1463838.

[12] Linus Nyman and Mikael Laakso. "Notes on the History of Fork and Join". In: *IEEE Annals of the History of Computing* 38.3 (July 2016), pp. 84–87. ISSN: 1058-6180. DOI: 10.1109/mahc.2016.34. URL: http://dx.doi.org/10.1109/MAHC.2016.34.

[13] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. "The implementation of the Cilk-5 multithreaded language". In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. PLDI98. ACM, May 1998. DOI: 10.1145/277650.277725. URL: http://dx.doi.org/10.1145/277650.277725.

[14] E. Ayguade et al. "The Design of OpenMP Tasks". In: *IEEE Transactions on Parallel and Distributed Systems* 20.3 (Mar. 2009), pp. 404–418. ISSN: 1045-9219. DOI: 10.1109/tpds.2008.105. URL: http://dx.doi.org/10.1109/TPDS.2008.105.

[15] Tsung-Wei Huang et al. "Cpp-Taskflow: Fast Task-Based Parallel Programming Using Modern C++". In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2019. DOI: 10.1109/ipdps.2019.00105. URL: http://dx.doi.org/10.1109/IPDPS.2019.00105.

[16] Alexey Kukanov. "The Foundations for Scalable Multicore Software in Intel Threading Building Blocks". In: *Intel Technology Journal* 11.04 (Nov. 2007). ISSN: 1535-864X. DOI: 10.1535/itj.1104.05. URL: http://dx.doi.org/10.1535/itj.1104.05.

[17] Florian Schmaus et al. "Nowa: A Wait-Free Continuation-Stealing Concurrency Platform". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, May 2021. DOI: 10.1109/ipdps49936.2021.00044. URL: http://dx.doi.org/10.1109/IPDPS49936.2021.00044.

[18] Chaoran Yang and John Mellor-Crummey. "A Practical Solution to the Cactus Stack Problem". In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '16. ACM, July 2016. DOI: 10.1145/2935764.2935787. URL: http://dx.doi.org/10.1145/2935764.2935787.

[19] Pablo Halpern. "Strict fork-join parallelism". In: *WG21 paper N* 3409 (2012).

[20] Edouard Lucas. *Théorie des nombres*. Vol. 1. Gauthier-Villars, 1891.

[21] Robert D. Blumofe and Charles E. Leiserson. "Scheduling multithreaded computations by work stealing". In: *Journal of the ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 1557-735X. DOI: 10.1145/324133.324234. URL: http://dx.doi.org/10.1145/324133.324234.

[22] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. "Thread Scheduling for Multiprogrammed Multiprocessors". In: *Theory of Computing Systems* 34.2 (Apr. 2001), pp. 115–144. ISSN: 1433-0490. DOI: 10.1007/s00224-001-0004-z. URL: http://dx.doi.org/10.1007/s00224-001-0004-z.

[23] Danny Hendler and Nir Shavit. "Non-blocking steal-half work queues". In: *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. PODC02. ACM, July 2002. DOI: 10.1145/571825.571876. URL: http://dx.doi.org/10.1145/571825.571876.

[24] Tom van Dijk and Jaco C. van de Pol. "Lace: Non-blocking Split Deque for Work-Stealing". In: *Euro-Par 2014: Parallel Processing Workshops*. Springer International Publishing, 2014, pp. 206–217. ISBN: 9783319143132. DOI: 10.1007/978-3-319-14313-2_18. URL: http://dx.doi.org/10.1007/978-3-319-14313-2_18.

[25] James Dinan et al. "Scalable work stealing". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. ACM, Nov. 2009. DOI: 10.1145/1654059.1654113. URL: http://dx.doi.org/10.1145/1654059.1654113.

[26] Hannah Cartier, James Dinan, and D. Brian Larkins. "Optimizing Work Stealing Communication with Structured Atomic Operations". In: *50th International Conference on Parallel Processing*. ICPP 2021. ACM, Aug. 2021. DOI: 10.1145/3472456.3472522. URL: http://dx.doi.org/10.1145/3472456.3472522.

[27] Karl-Filip Faxén. "Wool-A work stealing library". In: *ACM SIGARCH Computer Architecture News* 36.5 (Dec. 2008), pp. 93–100. ISSN: 0163-5964. DOI: 10.1145/1556444.1556457. URL: http://dx.doi.org/10.1145/1556444.1556457.

[28] Karl-Filip Faxen. "Efficient Work Stealing for Fine Grained Parallelism". In: *2010 39th International Conference on Parallel Processing*. IEEE, Sept. 2010. DOI: 10.1109/icpp.2010.39. URL: http://dx.doi.org/10.1109/ICPP.2010.39.

[29] Nhat Minh Lê et al. "Correct and efficient work-stealing for weak memory models". In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '13. ACM, Feb. 2013. DOI: 10.1145/2442516.2442524. URL: http://dx.doi.org/10.1145/2442516.2442524.

[30] Brian Norris and Brian Demsky. "CDSchecker: checking concurrent data structures written with C/C++ atomics". In: *ACM SIGPLAN Notices* 48.10 (Oct. 2013), pp. 131–150. ISSN: 1558-1160. DOI: 10.1145/2544173.2509514. URL: http://dx.doi.org/10.1145/2544173.2509514.

[31] David Chase and Yossi Lev. "Dynamic circular work-stealing deque". In: *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. SPAA05. ACM, July 2005. DOI: 10.1145/1073970.1073974. URL: http://dx.doi.org/10.1145/1073970.1073974.

[32] Jaemin Choi. "Formal Verification of Chase-Lev Deque in Concurrent Separation Logic". In: *ArXiv* abs/2309.03642 (2023). URL: https://api.semanticscholar.org/CorpusID:261582315.

[33] Shumpei Shiina and Kenjiro Taura. "Distributed Continuation Stealing is More Scalable than You Might Think". In: *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Sept. 2022. DOI: 10.1109/cluster51413.2022.00027. URL: http://dx.doi.org/10.1109/CLUSTER51413.2022.00027.

[34] Will Clinger, Anne Hartheimer, and Eric Ost. "Implementation strategies for continuations". In: *Proceedings of the 1988 ACM conference on LISP and functional programming*. LISP88. ACM, Jan. 1988. DOI: 10.1145/62678.62692. URL: http://dx.doi.org/10.1145/62678.62692.

[35] I-Ting Angelina Lee et al. "Using memory mapping to support cactus stacks in work-stealing runtime systems". In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. PACT '10. ACM, Sept. 2010. DOI: 10.1145/1854273.1854324. URL: http://dx.doi.org/10.1145/1854273.1854324.

[36] Chun-Xun Lin, Tsung-Wei Huang, and Martin D. F. Wong. "An Efficient Work-Stealing Scheduler for Task Dependency Graph". In: *2020 IEEE 26th International Conference on Parallel and Distributed*

Systems (ICPADS). IEEE, Dec. 2020. DOI: 10.1109/icpads51040.2020.00018. URL: http://dx.doi.org/10.1109/ICPADS51040.2020.00018.

[37] Zhiyao Ma and Lin Zhong. "Bringing Segmented Stacks to Embedded Systems". In: *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*. HotMobile '23. ACM, Feb. 2023. DOI: 10.1145/3572864.3580344. URL: http://dx.doi.org/10.1145/3572864.3580344.

[38] Franois Broquedis et al. "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications". In: *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, Feb. 2010. DOI: 10.1109/pdp.2010.67. URL: http://dx.doi.org/10.1109/PDP.2010.67.

[39] David W Walker and Jack J Dongarra. "MPI: a standard message passing interface". In: *Supercomputer* 12 (1996), pp. 56–68.

[40] Stephen Olivier et al. "UTS: An unbalanced tree search benchmark". In: *Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 235–250.

[41] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. DOI: 10.1109/cgo.2004.1281665. URL: http://dx.doi.org/10.1109/CGO.2004.1281665.