Heartbeat Scheduling

Provable Efficiency for Nested Parallelism

Umut Acar  
Carnegie Mellon University and Inria

Arthur Charguéraud  
Inria & University of Strasbourg, ICube

Adrien Guatto  
Inria

Mike Rainey  
Inria & Indiana University

Filip Sieczkowski  
Inria
Motivation: make it easier to write high-level and efficient fork-join parallel code

Running example:
(using notation of the Cilk language extensions for C/C++)

Applies function f to iterates in the range [lo, hi)

```c
void map(lo, hi, f)
    if lo <= hi
        return
    else if lo + 1 == hi
        f(lo)
        return
    int mid = (lo + hi) / 2
    spawn map(lo, mid, f)
    map(mid, hi, f)
    sync
```

Fork point enables calls to go in parallel

Join point blocks until both calls return
Fibers and their overheads

- We consider languages with support for fork join, on a multicore system.
- Every fork point potentially creates a fiber.
- Each fiber creation imposes a noticeable cost at runtime.
- The total cost can range from a few percent to a large enough to negate parallelism.

Can we design a runtime technique that ensures, for any fork-join program, bounded overheads on the overall cost of fiber creation?
Related work & contribution

Main approaches to taming fiber-creation overheads
Related work & contribution

Main approaches to taming fiber-creation overheads

Reduce the cost of each fiber creation
(useful, but not sufficient)
Related work & contribution

Main approaches to taming fiber-creation overheads

Reduce the cost of each fiber creation
(useful, but not sufficient)

Reduce the number of fibers created
(i.e., prune excess parallelism)
Related work & contribution

Main approaches to taming fiber-creation overheads

- Reduce the cost of each fiber creation (useful, but not sufficient)
- Reduce the number of fibers created (i.e., prune excess parallelism)
- Granularity control: Prediction of running time to throttle fiber creation (depends on predicting execution time, requires additional information, not always available)
**Main approaches to taming fiber-creation overheads**

- **Reduce the cost of each fiber creation**
  
  *useful, but not sufficient*

- **Reduce the number of fibers created**
  
  (i.e., *prune* excess parallelism)

  - **Granularity control:**
    Prediction of running time to throttle fiber creation
    
    *(depends on predicting execution time, requires additional information, not always available)*

  - **Lazy Scheduling:**
    Delay creating a fiber until it’s needed to realize parallelism
    
    *(no formal guarantees; known adversarial inputs)*
Related work & contribution

Main approaches to taming fiber-creation overheads

Reduce the cost of each fiber creation (useful, but not sufficient)

Reduce the number of fibers created (i.e., prune excess parallelism)

Granularity control: Prediction of running time to throttle fiber creation
(dependents on predicting execution time, requires additional information, not always available)

Lazy Scheduling: Delay creating a fiber until it’s needed to realize parallelism
(no formal guarantees; known adversarial inputs)

Heartbeat Scheduling: a runtime technique that, for any fork join program and any input, ensures provably small overheads and good utilization.
void map(lo, hi, f)
    if lo <= hi
        return
    else if lo + 1 == hi
        f(lo)
        return
    int mid = (lo + hi) / 2
    spawn map(lo, mid, f)
    map(mid, hi, f)
sync

Ready fibers
(what the scheduler sees at any instant)

Compiler

Runtime
(where the scheduling occurs)
Decision to be made by the runtime for each fork point

Enable latent parallelism

Sequentialize latent parallelism

Delay latent parallelism

Delay creating a fiber, in case the fork point ends up being excess parallelism
The problem with manual granularity control

An acceptable setting of $\text{grain}$ depends on:

(Tzannes et al 2014)

- The calling context
  - e.g., function $f$ might perform little to a lot of work, might perform a call to map
- The execution environment
  - Vagaries of chip architecture
  - Number of cores
  - Operating system / software environment

Manual granularity control degrades code quality and is not performance portable.
Heartbeat scheduling

Key idea: **amortize** fiber-creation overhead against past work

- At runtime, each core keeps track of how long it’s been since the previous fiber creation.

- When it’s been long enough, the core inspects the call stack of its current running fiber.

- If there’s some latent parallel call in the call stack, the core promotes the parallel call into a new fiber.
How heartbeat scheduling works

Heartbeat (alarm clock fires every $h$ cycles)

```c
void main()
    map(0,4,f)
    return

void map(lo, hi, f)
    if lo <= hi
        return
    else if lo + 1 == hi
        f(lo)
        return
    int mid = (lo + hi) / 2
    spawn map(lo, mid, f)
    map(mid, hi, f)
    sync
```
How heartbeat scheduling works

void main()
    map(0, 4, f)
    return

void map(lo, hi, f)
    if lo <= hi
        return
    else if lo + 1 == hi
        f(lo)
        return
    int mid = (lo + hi) / 2
    spawn map(lo, mid, f)
    spawn map(mid, hi, f)
    sync

Promotion

Snapshot of the call stack, just after second recursive call to map.

The stack grows down.

Heartbeat (alarm clock fires every $h$ cycles)
How heartbeat scheduling works

```c
void main()
  map(0,4,f)
return

void map(lo, hi, f)
  if lo <= hi
    return
  else if lo + 1 == hi
    f(lo)
    return
  int mid = (lo + hi) / 2
  spawn map(lo, mid, f)
  spawn map(mid, hi, f)
sync
```

Heartbeat (alarm clock fires every $h$ cycles)

SnapShot of the call stack, just after second recursive call to map.

The stack grows down.
How heartbeat scheduling works

Heartbeat (alarm clock fires every \( h \) cycles)

```c
void main()
    map(0,4,f)
return
```

```c
void map(lo, hi, f)
if lo <= hi
    return
else if lo + 1 == hi
    f(lo)
    return
int mid = (lo + hi) / 2
spawn map(lo, mid, f)
    map(mid, hi, f)
sync
```

**Snapshot of the call stack, just after second recursive call to map.**

**Create a new fiber**

**Promotion**

**Split the stack**

The stack grows down.
How heartbeat scheduling works

Heartbeat (alarm clock fires every $h$ cycles)

```c
void main()
    map(0,4,f)
    return

void map(lo, hi, f)
    if lo <= hi
        return
    else if lo + 1 == hi
        f(lo)
        return
    int mid = (lo + hi) / 2
    spawn map(lo, mid, f)
    map(mid, hi, f)
    sync
```

Snapshot of the call stack, just after second recursive call to `map`.

The stack grows down.

Create a new fiber

Promotion

Register dependency edges

Split the stack
**Cost model and time bound**

**Work**

\[ w = \text{total # of vertices} \]

**Span**

\[ s = \text{length of critical path} \]

- Work \( w = 21 \)
- Span \( s = 10 \)

**Critical path**
Cost model and time bound

Work
$w = \text{total \# of vertices}$

Span
$s = \text{length of critical path}$

Work-stealing bound:
(Blumofe & Leiserson)

For any fork-join program:
$E[t_p] \leq \frac{w}{p} + O(s)$

The bound accounts for the cost of load balancing fibers, but assigns to each scheduling operation a unit cost.
Time bound for heartbeat scheduling

Definitions:

\( W \)  Work (total # vertices)

\( S \)  Span (critical-path length)

\( t_p \)  Running time of the program on \( p \) cores

Work stealing:  \( E[t_p] \leq w/p + O(s) \)
Time bound for heartbeat scheduling

Definitions:

- $W$: Work (total # vertices)
- $S$: Span (critical-path length)
- $t_p$: Running time of the program on $p$ cores

Work stealing: $E[t_p] \leq w/p + \mathcal{O}(s)$

- $\tau$: Cost of creating a fiber
- $h$: Heartbeat duration

(Per fiber overhead = $\tau/h$.)
Definitions:

\( W \)  Work (total # vertices)

\( S \)  Span (critical-path length)

\( t_p \)  Running time of the program on \( p \) cores

\( \tau \)  Cost of creating a fiber

\( h \)  Heartbeat duration

\( h = k\tau \)  (Per fiber overhead = \( \tau/h \).)

Work stealing:

\[ E[t_p] \leq \frac{w}{p} + O(s) \]

We can pick \( h \) to be a multiple \( k \) of \( \tau \).
Time bound for heartbeat scheduling

Definitions:

\( W \)  Work (total \# vertices)

\( S \)  Span (critical-path length)

\( t_p \)  Running time of the program on \( p \) cores

\( \tau \)  Cost of creating a fiber

\( h \)  Heartbeat duration

(Per fiber overhead = \( \tau/h \).)

Work stealing:

\[ E[t_p] \leq \frac{w}{p} + O(s) \]

Work stealing with heartbeat, accounting for sched. overheads:

\[ E[t_p] \leq \frac{w}{p} + \left(\frac{1}{k} \times \frac{w}{p}\right) + O(k \times s) \]

1. Bounded increase in overheads (e.g., 5% of work, if \( k = 20 \))

2. Bounded increase in the span

\( h = k\tau \)

We can pick \( h \) to be a multiple \( k \) of \( \tau \).
Prototype implementation

Heartbeat mechanism

Need to wake up and try to promote $\approx 20-50\mu s$.

The heartbeat can be realized by software polling or hardware interrupts.

Native support for parallel loops

Should avoid introducing a new stack frame for each parallel loop invocation.

Our solution: extend frame representation to expose loop descriptor.

Cactus stack

( + heartbeat acceleration structure)

For calling convention: we use the classic cactus-stack representation.

Bookkeeping needed because we need $O(1)$ access to topmost promotable frame.

Promotable frames are linked together by a doubly linked list.
Experimental results

Baseline = original authors’ Cilk code

Ran on machine with 1TB RAM, using all 40 cores
Experimental results

Heartbeat is almost always faster than the baseline, sometimes a little slower.

Baseline = original authors’ Cilk code

Ran on machine with 1TB RAM, using all 40 cores.
Experimental results

Heartbeat is almost always faster than the baseline, sometimes a little slower.

Code simplification: the baseline codes use several manual granularity-control techniques.
Heartbeat uses none!

Baseline = original authors’ Cilk code
Ran on machine with 1TB RAM, using all 40 cores
Related work

**Formal bounds for scheduling fork join**
Brent ’74, Arora et al ’98, Blumofe & Leiserson ’99, Agarwal et al ’07, Acar et al ’11

**Lazy-scheduling methods**
Mohr et al ’91, Feeley ’93, Goldstein et al ’96, Frigo et al ’98, Imam et al ’14, Tzannes et al ’14

**Prediction-based methods**
Weening ’89, Pehoushek et al ’90, Lopez et al ’96, Duran et al ’08, Acar et al ’16, Iwasaki et al ’16, Shintaro et al ’16

Heartbeat is the first to show analytical bounds on scheduling overheads for all fork join programs.

Heartbeat is the first in this class of approaches to have a state-of-the-art implementation and be backed by end-to-end bounds.

Heartbeat offers similar but stronger guarantees than Oracle-Guided Granularity Control, and delivers state-of-the-art in performance.
Conclusion

• Heartbeat scheduling supports really lightweight nested parallelism:
  • It simplifies code: no need for manual granularity control.
  • It is protected by formal bounds from adversary programs.
  • It can, on ten benchmarks, achieve comparable or better performance to Cilk, a carefully engineered implementation.

• Future work:
  • Optimized compiler implementation
  • Generalizing beyond fork join (e.g., futures)

• Thanks for your attention!