Heartbeat Scheduling: Provable Efficiency for Nested Parallelism

Umut A. Acar
Carnegie Mellon University and Inria
USA
umut@cs.cmu.edu

Arthur Charguéraud
Inria and Univ. of Strasbourg, ICube
France
arthur.chargueraud@inria.fr

Adrien Guatto
Inria
France
adrien@guatto.org

Mike Rainey
Inria and Center for Research in
Extreme Scale Technologies (CREST)
USA
me@mike-rainey.site

Filip Sieczkowski
Inria
France
filip.sieczkowski@inria.fr

Abstract

A classic problem in parallel computing is to take a high-level parallel program written, for example, in nested-parallel style with fork-join constructs and run it efficiently on a real machine. The problem could be considered solved in theory, but not in practice, because the overheads of creating and managing parallel threads can overwhelm their benefits. Developing efficient parallel codes therefore usually requires extensive tuning and optimizations to reduce parallelism just to a point where the overheads become acceptable.

In this paper, we present a scheduling technique that delivers provably efficient results for arbitrary nested-parallel programs, without the tuning needed for controlling parallelism overheads. The basic idea behind our technique is to create threads only at a beat (which we refer to as the "heartbeat") and make sure to do useful work in between. We specify our heartbeat scheduler using an abstract-machine semantics and provide mechanized proofs that the scheduler guarantees low overheads for all nested parallel programs. We present a prototype C++ implementation and an evaluation that shows that Heartbeat competes well with manually optimized Cilk Plus codes, without requiring manual tuning.

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

Keywords parallel programming languages, granularity control

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192391

ACM Reference Format:
Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3192366.3192391

1 Introduction

A longstanding goal of parallel computing is to build systems that enable programmers to write a high-level codes using just simple parallelism annotations, such as fork-join, parallel for-loops, etc, and to then derive from the code an executable that can perform well on small numbers of cores as well as large. Over the past decade, there has been significant progress on developing programming language support for high level parallelism. Many programming languages and systems have been developed specifically for this purpose. Examples include OpenMP [46], Cilk [26], Fork/Join Java [38], Habanero Java [35], TPL [41], TBB [36], X10 [16], parallel ML [24, 25, 30, 48, 51], and parallel Haskell [43].

These systems have the desirable feature that the user expresses parallelism at an abstract level, without directly specifying how to map lightweight threads (just threads, from hereon) onto processors. A *scheduler* is then responsible for the placement of threads. The scheduler does not require that the thread structure is known ahead of time, and therefore operates online as part of the runtime system. Many scheduling algorithms have been developed, taking into account a variety of asymptotic cost factors including execution time, space consumption, and locality [1–3, 5, 9–13, 15, 18, 29, 31, 45].

Most scheduling algorithms that come with a formal analysis establish asymptotic bounds in a simplified model in which spawning a thread has unit cost. Correspondingly, the job of achieving low constant factors for scheduling operations is usually treated as a purely empirical question, and approached as such. Yet, in practice, depending on the implementation, the cost of creating a thread, scheduling it,

then destroying it, may amount to thousands of cycles. Thus, in spite of the guarantees provided by the asymptotic analysis, poor management of overheads can result in massive slowdowns in practice—sometimes as much as 50x, in our experience. On the other extreme, it is also easy to overcompensate by reducing overheads too much, thereby creating too few threads to keep cores well fed. The key is to find solutions that strike a balance between the two extremes, and do so consistently.

Dozens of papers have been published on various techniques for taming parallelism overheads. To our knowledge, they fall in either of two categories: *granularity control* approaches [4, 19, 34, 37, 42, 47, 57], and *lazy splitting* approaches [26–28, 33, 44, 55]. Granularity control aims at coarsening the leaf-level subcomputations to avoid the creation of threads involving little work. Lazy splitting aims at creating threads on demand, postponing the creation of threads until other workers are in need. Although many of these techniques have entered into production systems and have led to improved performance overall, significant challenges remain.

In the face of nested parallelism, that is, when parallel constructs are nested, many of these approaches have been shown by the literature to fail to control overheads for certain classes of programs [55]. Parallelism overheads continue to be a challenge for practitioners. Expertly implemented research codes are not immune from such issues either: codes from a popular benchmarking suite that is implemented using Cilk Plus sometimes resort to using a variety of heuristics, such as manual loop grains, that, while controlling overheads well in many cases, may still perform poorly on certain classes of input [8]. Even though good results might be achieved on the benchmark considered, there is no guarantee that good performance will be achieved by the resulting program on different input data, or on a different hardware system [4, 53]. Given such sensitivity to input data, hardware details, program structure, etc., we believe that an approach based on a formally verified guarantee may be useful and could lead to greater reliability in the face of uncertainty in the execution environment.

Yet, among all the proposals that we are aware of, we know of only one that provides a formal efficiency bound for nested-parallel programs such that that bound takes into account the overheads of thread creation, namely *oracle-guided scheduling* [4]. In that approach, the programmer annotates every parallel call with an expression for computing its asymptotic cost and the run-time system decides whether a given call should be performed sequentially or in parallel. One limitation of this approach is that it breaks down when confronted with parallel functions whose complexity cannot be predicted reasonably precisely, e.g., functions whose complexity is data dependent (e.g., a string comparison function) and search algorithms in which cost estimates are difficult to make.

In this paper, we propose techniques for controlling thread-creation costs in all fork-join (nested-parallel) parallel programs. The basic idea behind these techniques is to amortize the cost of thread creation by ensuring that threads are created at a "heartbeat", i.e., periodically at intervals, each of which include sufficiently large amount of sequential work. This technique applies to all fork-join programs—we make no additional assumptions on the kinds of programs—and in particular, permits arbitrary nesting of parallel calls.

Our contribution can be summarized as follows.

- We present heartbeat scheduling, a technique for promoting parallel-call frames into threads.
- We formalize heartbeat scheduling using abstract machines and present formal bounds on the work and span, verified in Coq.
- We present empirical evidence that our approach leads in practice to bounded overheads and competitive performance, by evaluating a number of state-of-the-art benchmarks originally designed for Cilk Plus.

In Section 2, we give an overview of our approach, the bounds that we prove, and the implementation. In Section 3, we describe our approach formally by means of an abstract machine, and establish the theoretical cost bounds. In Section 4, we discuss the concrete implementation of our approach, including a cactus-stack data structure that is crucial to the implementation, the optimized treatment of parallel loops, and the heartbeat interpreter used for the proof-of-concept evaluation. In Section 5, we present benchmark results to confirm that our theoretical results on heartbeat scheduling translate into practical benefits.

2 Overview

In heartbeat scheduling, the runtime takes, at every N steps, the top-most stack frame corresponding to a parallel call and *promotes* it into a proper thread that may be subject to load balancing. The heartbeat period, N, is a system-dependent parameter that is chosen once for the system. The idea is to evaluate parallel function calls using conventional stack frames, essentially sequentially, and to promote these frames into proper threads when sufficient amount of sequential work has been performed since the last promotion.

It might be tempting to view heartbeat scheduling idea as a *lazy scheduling* technique [55], because heartbeat scheduling delays thread creation until its costs can be amortized. This view is partially justifiable, but not completely. In prior approaches to lazy scheduling [55], promotion is triggered, either directly or indirectly, by steal requests. Such a technique is unlikely to be (provably) effective for all programs. For example, numerous steal requests may lead to a large number of promotions, potentially resulting is large overheads. Or, a burst of steal requests might all have to wait for a few processors to create parallelism (and hence blow past span bounds) because the system has been "too lazy" and

did not create sufficiently many parallel threads. In contrast, heartbeat scheduling is independent from the load of the workers in the system. It relies solely on a processor-local decision, based on the number of cycles elapsed since the previous promotion, and is thus never too lazy nor too eager in creating parallelism.

We present bounds on the work and span of programs executed using heartbeat scheduling. The total work, including thread creation overheads, is bounded as

$$W \leq (1 + \frac{\tau}{N}) \cdot w,$$

where w denotes the *raw work* (total sequential execution time, excluding overheads), and τ denotes the cost of promoting one frame and scheduling the resulting thread. The total span, including thread creation overheads, is bounded as:

$$S \leq (1 + \frac{N}{\tau}) \cdot s,$$

where s denotes the span (the length of the critical path) of a fully parallel execution, that is, an execution in which all parallel calls are directly represented as threads. The fact that these bounds hold is not immediate and, in particular, rests upon promoting the oldest possible frame. Taken together, the work and span bounds show that the overheads can be tamed down to a small fraction of the run time, while nevertheless preserving the asymptotic amount of parallelism inherent to the program. In particular, unlike most scheduling techniques based on heuristics, hearbeat scheduling cannot be defeated by adversarial programs.

Perhaps surprisingly, heartbeat scheduling is agnostic to the specific load-balancing algorithm used for ensuring good work distribution. For example, we may combine heartbeat scheduling with work stealing, which guarantees an execution time of $T \leq \frac{W}{P} + O(S)$, where P denotes the number of processors. Thus by using work stealing for load balancing, we obtain the bound on P-processor run time of:

$$T \leq \left(1 + \frac{\tau}{N}\right) \cdot \frac{w}{P} + O\left(\frac{N}{\tau} \cdot s\right).$$

To limit overheads, we can choose N as a multiple of τ, i.e., $N = k\tau$, for some fixed k. The bound can then be written as:

$$T \leq \left(1 + \frac{1}{k}\right) \cdot \frac{w}{P} + O(ks).$$

The above bound shows that heartbeat scheduling achieves bounded overheads (e.g., 5% for $k = 20$), while increasing the span by k. Thus, for all programs with sufficient *parallel slackness* [56], where $\frac{w}{s} \gg kP$, performance should be close to optimal. In practice, this assumption is met by all algorithms featuring a logarithmic (or polylogarithmic) span.

To implement heartbeat scheduling, we rely on a *cactus stack* [32]. This representation essentially consists of pieces of the call stack organized in a tree structure. The cactus stack supports parallel computations without prohibitive stack-space usage in the worst case. To support constant-time access to the next *promotable* frame in the stack, we extend the cactus stack with a doubly-linked list between the promotable frames, i.e., the frames associated with parallel calls or parallel loops.

We empirically validate our approach by presenting a proof-of-concept implementation in C++. Our implementation uses an interpreter whereby benchmark programs are represented by values of an AST (Abstract Syntax Tree). Our interpreter explicitly allocates and deallocates frames from the cactus stack and implements frame promotion following the heartbeat strategy. Our implementation depends on regular polling to realize the heartbeat.

Thanks to the fact that sequential blocks are compiled and optimized by a conventional compiler, the resulting interpreter is efficient enough to enable meaningful comparison with the performance of state-of-the-art benchmarks compiled using Cilk Plus. We perform this comparison using parallel programs from the Problem Based Benchmark Suite (PBBS). PBBS consists of nontrivial algorithms exhibiting irregular parallelism [50]. The PBBS programs are implemented in Cilk Plus using a careful combination of techniques for controlling granularity.

Our benchmark results show that PBBS programs can incur significant overheads, sometimes over 25% of the execution time. With heartbeat scheduling, overheads are always less than 5%, because significantly fewer threads are created. This result matches our theoretical bounds for the setting of N that we consider. Despite the interpretive overheads of our proof-of-concept implementation, our code is generally able to match or beat the performance of PBBS programs on 40 cores. Our results also show that heartbeat scheduling achieves utilization similar to original PBBS programs, despite creating significantly fewer threads.

3 Semantics and Analysis

We introduce heartbeat scheduling, as an idealized nested-parallel language, in the setting of an untyped λ-calculus equipped with parallel pairs. This language allows us to present and analyze the key ideas of heartbeat scheduling, abstracting over implementation details. In the implementation section, we will show that the basic transitions, in particular the operation for splitting the stack on promotion operations, can indeed be implemented in constant time.

We give three semantics to that language: a *fully-sequential* semantics, a *fully-parallel* semantics, and our *heartbeat scheduling* semantics. The definition is two-fold.

First, we define a variant of the CEK machine of Felleisen and Friedman [23], which implements sequential call-by-value evaluation. This abstract machine makes the stack explicit and is thus well-suited for describing the stack surgery operations involved in heartbeat scheduling.

Second, we use the abstract machine to define the three aforementioned semantics in big-step style. This big-step

$$
\begin{array}{rcl}
\text{Cost graph } g & ::= & \mathbf{0} \mid \mathbf{1} \mid (g \cdot g) \mid (g \,||\, g) \\
\\
\mathit{work}(\mathbf{0}) & \triangleq & 0 \\
\mathit{work}(\mathbf{1}) & \triangleq & 1 \\
\mathit{work}(g_1 \cdot g_2) & \triangleq & \mathit{work}(g_1) + \mathit{work}(g_2) \\
\mathit{work}(g_1 \,||\, g_2) & \triangleq & \tau + \mathit{work}(g_1) + \mathit{work}(g_2) \\
\\
\mathit{span}(\mathbf{0}) & \triangleq & 0 \\
\mathit{span}(\mathbf{1}) & \triangleq & 1 \\
\mathit{span}(g_1 \cdot g_2) & \triangleq & \mathit{span}(g_1) + \mathit{span}(g_2) \\
\mathit{span}(g_1 \,||\, g_2) & \triangleq & \tau + \mathit{max}(\mathit{span}(g_1), \mathit{span}(g_2))
\end{array}
$$

Figure 1. Cost graphs, and definition of work and span.

presentation allows us to describe the creation of threads in the semantics without having to explicitly manipulate the set of live threads. In particular, the rule describing the parallel evaluation of a pair involves evaluation premises describing distinct, independent instances of the abstract machine.

To formally reason about the work and span of an evaluation, we instrument the big-step judgments in such a way that, in addition to an output value, they also produce a *cost graph*. This cost graph describes the operations and control dependencies performed during the corresponding execution. All vertices have unit cost except fork-join operations, which are given the weight τ, to reflect the overhead of thread management. On a cost graph, we define work and span in the usual way, as the weight of the complete graph and as the weight of the critical path, respectively.

Based on the formal semantics and the cost graphs, we establish the correctness and the efficiency of heartbeat scheduling. The correctness results assert that the heartbeat semantics produces the same results as the fully-sequential and the fully-parallel semantics. The efficiency results have two components. The first component asserts that the heartbeat semantics adds at most a fraction $\frac{\tau}{N}$ of work compared with the fully-sequential semantics, where N is a parameter under user control. In other words, overheads are bounded. The second component asserts that the span of a program executed in the heartbeat semantics increases at most by a constant multiplicative factor $(1 + \frac{N}{\tau})$ compared with the fully-parallel semantics. Thus, the asymptotic amount of parallelism inherent to the program is preserved when using heartbeat scheduling.

For increased confidence, we have formalized in the Coq proof assistant all the technical contents of this section.

3.1 Work and span of a cost graph

We use *cost graphs* as a convenient way to formalize the work and span of an execution. The execution of a fork-join program induces a series-parallel, directed acyclic graph. Figure 1 gives the grammar of cost graphs, which includes: the empty graph, written $\mathbf{0}$, the one-vertex graph, written $\mathbf{1}$, sequential composition of two graphs, written $(g_1 \cdot g_2)$, and parallel composition of two graphs, written $(g_1 \,||\, g_2)$.

Figure 1 also gives the formal definition of the work and span of cost graph g, written $\mathit{work}(g)$ and $\mathit{span}(g)$, respectively. Unlike prior work, we do not assign unit cost to fork-join operations, but instead weight these operations with some cost τ. This fixed parameter τ represents the runtime overhead associated with a fork-join operation.

Intuitively, the work of a cost graph is equal to the number of vertices plus τ times the number of fork vertices involved in the graph. The span of a cost graph is equal to the length of the longest path in that graph, when counting τ units on every traversal of a fork vertex.

$$
\begin{array}{llcl}
\text{Expression} & e & ::= & x \mid \lambda x.e \mid (e\ e) \mid (e \,||\, e) \\
\text{Value} & v & ::= & (v, v) \mid (\lambda x.e)\{\sigma\} \\
\text{Environment} & \sigma & \in & \mathit{Var} \rightarrow_{\mathit{fin}} \mathit{Val} \\
\text{Frame} & f & ::= & \mathsf{APPL}(\square, e, \sigma) \mid \mathsf{APPR}(x, e, \sigma, \square) \\
 & & & \mid \mathsf{PAIRL}(\square, e, \sigma) \mid \mathsf{PAIRR}(v, \square) \\
\text{Stack} & k & ::= & \mathsf{TOP} \mid f :: k \\
\text{Code} & c & ::= & e \mid v \\
\text{Configuration} & m & ::= & \langle c \mid \sigma \mid k \rangle
\end{array}
$$

Figure 2. Syntax of the source language, and components of the abstract machine.

3.2 Syntax and Machine Transitions

The syntax of our calculus is given in the first three lines of Figure 2. An *expression* e is either a variable, an abstraction, an application, or a parallel pair, written $(e_1 \,||\, e_2)$. Such a pair marks an opportunity for parallelism that may or may not actually execute in parallel, depending on the scheduling decision. A *value* v denotes a completely evaluated expression. It is either a pair of values (v_1, v_2), or a closure $(\lambda x.e)\{\sigma\}$. Such a closure packages an *environment* σ, which consists of a finite map from variables to values. We write $\sigma[x \mapsto v]$ for the environment σ updated to map x to v. For brevity, we omit projection functions, whose semantics is standard.

A *machine configuration* m is a triple $\langle c \mid \sigma \mid k \rangle$, where the *code* c, either a value or an expression, is executing in an environment σ, against a *stack* k. A stack consists of a list of *frames* terminated by the TOP token. We let the metavariable f range over stack frames. Conceptually, a frame is an expression constructor with a hole, written $\square$, and it describes a partially evaluated expression.

Figure 3 defines the judgment $m \longrightarrow m'$, which describes the sequential transitions of our abstract machine. The sequential transition rules are standard.

For example, let us describe the steps involved in the evaluation of an application $(e_1\ e_2)$. First, the rule AppL puts the function e_1 in the code component of the machine, and extends the stack with a frame $\mathsf{APPL}(\square, e_2, \sigma)$, thereby saving the argument e_2 and the current environment σ. Once the

$$
\begin{array}{rllllllll}
\textsc{Var} & \langle\, x & \mid \sigma \mid & k \,\rangle & \to & \langle\, \sigma(x) & \mid - & \mid & k \,\rangle \\
\textsc{Abs} & \langle\, \lambda x.e & \mid \sigma \mid & k \,\rangle & \to & \langle\, (\lambda x.e)\{\sigma\} & \mid - & \mid & k \,\rangle \\
\textsc{AppL} & \langle\, (e_1\; e_2) & \mid \sigma \mid & k \,\rangle & \to & \langle\, e_1 & \mid \sigma & \mid & \mathsf{APPL}(\square, e_2, \sigma) :: k \,\rangle \\
\textsc{AppR} & \langle\, (\lambda x.e)\{\sigma\} & \mid - \mid & \mathsf{APPL}(\square, e_2, \sigma') :: k \,\rangle & \to & \langle\, e_2 & \mid \sigma' & \mid & \mathsf{APPR}((\lambda x.e)\{\sigma\}, \square) :: k \,\rangle \\
\textsc{Body} & \langle\, v & \mid - \mid & \mathsf{APPR}((\lambda x.e)\{\sigma\}, \square) :: k \,\rangle & \to & \langle\, e & \mid \sigma[x \mapsto v] & \mid & k \,\rangle \\
\textsc{PairL} & \langle\, (e_1 \,||\, e_2) & \mid \sigma \mid & k \,\rangle & \to & \langle\, e_1 & \mid \sigma & \mid & \mathsf{PAIRL}(\square, e_2, \sigma) :: k \,\rangle \\
\textsc{PairR} & \langle\, v_1 & \mid - \mid & \mathsf{PAIRL}(\square, e_2, \sigma) :: k \,\rangle & \to & \langle\, e_2 & \mid \sigma & \mid & \mathsf{PAIRR}(v_1, \square) :: k \,\rangle \\
\textsc{Pair} & \langle\, v_2 & \mid - \mid & \mathsf{PAIRR}(v_1, \square) :: k \,\rangle & \to & \langle\, (v_1, v_2) & \mid - & \mid & k \,\rangle
\end{array}
$$

Figure 3. Sequential machine transitions: $m \to m'$.

$$
\textsc{SeqVal}\quad \frac{}{\langle v \mid - \mid \mathsf{TOP} \rangle \Rightarrow_{\text{seq}} v;\, \mathbf{0}}
\qquad
\textsc{SeqStep}\quad \frac{m \to m' \qquad m' \Rightarrow_{\text{seq}} v;\, g}{m \Rightarrow_{\text{seq}} v;\, (\mathbf{1} \cdot g)}
$$

Figure 4. Sequential semantics: $m \Rightarrow_{\text{seq}} v;\, g$.

$$
\textsc{ParVal}\quad \frac{}{\langle v \mid - \mid \mathsf{TOP} \rangle \Rightarrow_{\text{par}} v;\, \mathbf{0}}
$$

$$
\textsc{ParStep}\quad \frac{c \neq (_ \,||\, _) \qquad \langle c \mid \sigma \mid k \rangle \to m' \qquad m' \Rightarrow_{\text{par}} v;\, g}{\langle c \mid \sigma \mid k \rangle \Rightarrow_{\text{par}} v;\, (\mathbf{1} \cdot g)}
$$

$$
\textsc{ParPair}\quad \frac{\langle e_1 \mid \sigma \mid \mathsf{TOP} \rangle \Rightarrow_{\text{par}} v_1;\, g_1 \qquad \langle e_2 \mid \sigma \mid \mathsf{TOP} \rangle \Rightarrow_{\text{par}} v_2;\, g_2 \qquad \langle (v_1, v_2) \mid - \mid k \rangle \Rightarrow_{\text{par}} v;\, g_3}{\langle (e_1 \,||\, e_2) \mid \sigma \mid k \rangle \Rightarrow_{\text{par}} v;\, ((g_1 \,||\, g_2) \cdot g_3)}
$$

Figure 5. Parallel semantics: $m \Rightarrow_{\text{par}} v;\, g$.

function has evaluated to a closure of the form $(\lambda x.e)\{\sigma\}$, the rule AppR pops the frame $\mathsf{APPL}(\square, e_2, \sigma)$ from the stack, puts the argument e_2 in the code component of the machine, and extends the stack with a frame $\mathsf{APPR}((\lambda x.e)\{\sigma\}, \square)$. Once the argument has evaluated to a value v, the rule Body pops the frame $\mathsf{APPR}((\lambda x.e)\{\sigma\}, \square)$ from the stack, and begins the evaluation of the body e in an extended environment obtained by adding to σ a binding from x to v.

In the machine transitions, parallel pairs are evaluated in a similar way as applications: the left branch evaluates first, then the right branch, then the two results are paired up into a value. In the parallel semantics presented further on, parallel pairs have their branches evaluated concurrently by distinct instances of the abstract machine.

3.3 Sequential and Parallel Cost Semantics

We next present big-step judgments describing the fully-sequential and the fully-parallel evaluation of a program.

Figure 4 presents the evaluation judgment $m \Rightarrow_{\text{seq}} v;\, g$, which is essentially a big-step wrapper around the small-step judgment $m \to m'$. This wrapper associates a cost graph g to a sequential execution of a machine m that terminates with result value v. The cost graph produced by such a sequential execution consists of a long chain of $\mathbf{1}$, terminated by a $\mathbf{0}$. Both the work and the span of the graph g are equal to the length of that chain and match the number of machine transitions performed during the execution.

Figure 5 presents the evaluation judgment $m \Rightarrow_{\text{par}} v;\, g$, which corresponds to the fully-parallel evaluation of a program, that is, an evaluation in which every parallel pair gets evaluated in parallel by spawning new machines. Concretely, the rule ParPair evaluates a parallel pair $(e_1 \,||\, e_2)$ by considering independently the evaluation of e_1 and the evaluation of e_2 in two distinct abstract machines, producing the results v_1 and v_2, and the cost graphs g_1 and g_2, respectively. In the third premise of that rule, the pair of results (v_1, v_2) is passed to the remainder of the computation, which is represented by the stack k. The third premise evaluates to a final result v, with a corresponding cost graph called g_3. The cost graph $((g_1 \,||\, g_2) \cdot g_3)$ that appears in the conclusion of the rule ParPair reflects the fact that g_1 and g_2 are composed in parallel, while g_3 comes in sequence after the join point.

The other two rules that define the parallel evaluation judgment, namely ParVal and ParStep, are the counterparts of the rules SeqVal and SeqStep from the sequential evaluation judgment, with the only difference that the rule ParStep includes one extra premise. This extra premise prevents the rule ParStep from being triggered on parallel pairs, which are meant to be treated by rule ParPair.

3.4 Hearbeat Semantics

Figure 6 presents the evaluation judgment $m; n \Rightarrow_{\text{hb}} v;\, g$, which describes the evaluation of a program using heartbeat scheduling. It expresses that the machine m terminates with the result value v and cost graph g, starting from a state with n *credits*, indicating that n transitions were performed on the machine since the previous promotion.

The evaluation of a parallel pair begins as in the fully-sequential semantics. During the execution of the left branch, either the right branch gets promoted and evaluates in parallel on a distinct machine; or it remains as an unpromoted frame in the stack, in which case it gets evaluated on the same machine as the one that processed the left branch.

HBVal

$$\frac{}{\langle v \mid - \mid \mathsf{TOP} \rangle; n \Rightarrow_{\mathrm{hb}} v;\ \mathbf{0}}$$

HBStep

$$\frac{n < N \ \vee\ \neg promotable(k) \qquad \langle c \mid \sigma \mid k \rangle \to m' \qquad m'; (n+1) \Rightarrow_{\mathrm{hb}} v;\ g}{\langle c \mid \sigma \mid k \rangle; n \Rightarrow_{\mathrm{hb}} v;\ (\mathbf{1} \cdot g)}$$

HBPromote

$$\frac{n \geq N \qquad \neg promotable(k_2) \qquad \langle c \mid \sigma \mid k_1 \rangle; 0 \Rightarrow_{\mathrm{hb}} v_1;\ g_1 \qquad \langle e_2 \mid \sigma' \mid \mathsf{TOP} \rangle; 0 \Rightarrow_{\mathrm{hb}} v_2;\ g_2 \qquad \langle (v_1, v_2) \mid - \mid k_2 \rangle; 0 \Rightarrow_{\mathrm{hb}} v;\ g_3}{\langle c \mid \sigma \mid k_1 \mathbin{@} \mathsf{PAIRL}(\Box, e_2, \sigma') :: k_2 \rangle; n \Rightarrow_{\mathrm{hb}} v;\ ((g_1 \,||\, g_2) \cdot g_3)}$$

where $promotable(k) \ \triangleq\ \mathsf{PAIRL}(\Box, _, _) \in k$.

Figure 6. heartbeat semantics: $m; n \Rightarrow_{\mathrm{hb}} v;\ g$.

The first two rules that define the heartbeat evaluation judgment, namely HBVal and HBStep, are the counterpart of the rules SeqVal and SeqStep from the sequential evaluation judgment. The rule HBStep performs a sequential transition and increments the number of credits by one unit, from n to $n + 1$. Sequential transitions are performed unless the time has come to perform a promotion, as captured by the first premise of HBStep: "$n < N \vee \neg\, promotable(k)$". The negation of this premise asserts that the stack k contains a frame of the form $\mathsf{PAIRL}(\Box, _, _)$ that could be promoted, and that at least N transitions were performed since the previous promotion.

When these two conditions are met, the rule HBPromote applies. In short, the rule takes the oldest PAIRL frame in the stack and promotes it by spawning the corresponding right branch into a separate abstract machine. It also creates another machine for evaluating the continuation that processes the result of that parallel pair, i.e., the join continuation.

Let us look more closely at rule HBPromote. Its conclusion asserts that the rule HBPromote applies to a configuration whose stack is of the form $k_1 \mathbin{@} \mathsf{PAIRL}(\Box, e_2, \sigma') :: k_2$, where @ denotes stack concatenation, and k_1 and k_2 denote two pieces of stacks (possibly empty). The second premise, $\neg promotable(k_2)$, ensures that the frame $\mathsf{PAIRL}(\Box, e_2, \sigma')$ considered for promotion is the oldest promotable frame from the stack, that is, the frame corresponding to the outermost parallel pair. Promoting the oldest pair is necessary to minimize the span degradation, similarly to how one should steal the oldest frame in work-stealing.

The rule HBPromote contains three evaluation premises. The first one, with configuration $\langle c \mid \sigma \mid k_1 \rangle$, describes what remains of the machine after the promotion takes place. The second one, with configuration $\langle e_2 \mid \sigma' \mid \mathsf{TOP} \rangle$, describes the evaluation of the right branch of the parallel pair that was promoted. The third one, with configuration $\langle (v_1, v_2) \mid - \mid k_2 \rangle$, describes the join continuation. The join continuation processes the pair made of the results produced by the two branches. Just like in the rule Rule ParPair, the cost graph involved in the rule HBPromote is of the form $(g_1 \,||\, g_2) \cdot g_3$.

3.5 Formal Results

Our correctness result asserts that the three semantics compute the same output values. This result is completely independent from the cost graphs. Thereafter, to hide cost graphs, we write $m \Rightarrow_{\mathrm{seq}} v$ as short for $\exists g.(m \Rightarrow_{\mathrm{seq}} v;\ g)$, and likewise for the two other evaluation judgments.

Theorem 1 (Correctness). *For any machine m and value v,*

$$(m \Rightarrow_{\mathrm{seq}} v) \quad\Leftrightarrow\quad (m \Rightarrow_{\mathrm{par}} v) \quad\Leftrightarrow\quad (m; 0 \Rightarrow_{\mathrm{hb}} v).$$

Our first efficiency result asserts that the overheads induced by heartbeat scheduling are bounded by a fraction of the work performed in the fully-sequential semantics.

Theorem 2 (Work bound). *Assume: $m \Rightarrow_{\mathrm{seq}} v;\ g_s$. Then, there exists a cost graph g_h such that $m; 0 \Rightarrow_{\mathrm{hb}} v;\ g_h$ and*

$$work(g_h) \ \leq\ \left(1 + \frac{\tau}{N}\right) \cdot work(g_s)$$

where τ is the overhead of task creation (recall §3.1), and N is a parameter under user control (recall §3.4).

Proof. The proof is by induction on the sequential derivation. To prove the inequality, we need to generalize the statement of the theorem to deal with a nonzero number of credits. The generalized induction hypothesis has for conclusion $m; n \Rightarrow_{\mathrm{hb}} v;\ g_h$ with $work(g_h) \leq \left(1 + \frac{\tau}{N}\right) \cdot work(g_s) + n \cdot \frac{\tau}{N}$. Once the induction is set up, the remainder of the proof is rather mechanical. We refer the reader to the Coq formalization for additional proof details. □

Our second efficiency result asserts that the span increases at most by a multiplicative factor compared with the fully-sequential semantics.

Theorem 3 (Span bound). *Assume: $m \Rightarrow_{\mathrm{par}} v;\ g_p$. Then, there exists a cost graph g_h such that $m; 0 \Rightarrow_{\mathrm{hb}} v;\ g_h$ and*

$$span(g_h) \ \leq\ \left(1 + \frac{N}{\tau}\right) \cdot span(g_p)$$

where τ and N are as in Theorem 2.

Proof. The proof of the span bound is much trickier than that of the work bound. It proceeds by induction on the parallel derivation. Again, we generalize the induction hypothesis. It has for conclusion $\langle c \mid \sigma \mid k \rangle; n \Rightarrow_{\mathrm{hb}} v;\ g_h$ with

$$span(g_h) \leq \ \left(1 + \tfrac{N}{\tau}\right) \cdot span(g_p) - (\text{if } promotable(k) \text{ then } \min(n, N) \text{ else } 0).$$

Our proof uses as technical device an auxiliary semantics that sits halfway between the fully-parallel and heartbeat semantics. Interestingly, the proof critically relies on the fact that we systematically promote the oldest promotable frame, that is, the frame which corresponds to the outermost pair. Again, we refer the reader to the Coq proof for details. □

4 Implementation

This section describes the reference system, one we will refer to as Heartbeat, that we implemented to test the practicality of heartbeat scheduling. Like many other lightweight-threading systems, such as TBB [36], Heartbeat begins by launching one pthread per core. Each of these *workers* alternate between evaluating a lightweight thread and participating in the load balancing scheme. Heartbeat is agnostic to the load balancing algorithm—we discuss particular work-stealing-based implementations in Section 5.

The cactus stack data structure. A well-known problem in scheduling parallel computations is worst-case stack space usage: lost stack space can become problematic as a result of stacks being attached to suspended computations. A classic data structure addressing this problem is the so-called *cactus stack* data structure [32]. A cactus stack is a tree representation of the call stack of the program in which branching points correspond to parallel forks. For Heartbeat, we employ a classic optimization based on *stacklets* [28], which correspond to small, contiguous regions of memory (e.g., of size 4k bytes). Stacklets enable cheap allocation of frames and avoid allocating them on the heap. One limitation of the cactus stack is that it requires a modified calling convention, which breaks interoperability with legacy third-party binaries. Alternative approaches have been proposed [40, 58], but it appears that one must either sacrifice binary interoperability, time bounds, or space bounds. We expect that the extension that Heartbeat uses to set up links between promotable frames could be adapted to a number of these alternative approaches.

We next explain why Heartbeat uses a doubly linked list between promotable frames. To implement promotion efficiently, we need constant-time access to the top-most promotable frame. Once this frame is promoted, we need to access the next one, and so on. Thus, at a minimum, we need a singly linked list between promotable frames, from top to bottom, i.e. starting from the oldest frames. Yet, at the same time, the execution of a thread pushes and pops frames at the bottom of its stack. In particular, it is possible that a promotable frame gets popped before it is promoted (e.g., the left branch of a pair terminates before the right branch gets promoted). Efficiently removing the frame from the singly linked list between promotable frames requires reverse pointers, hence the need for a doubly linked list. To represent this doubly linked list, promotable frames include a *prev* and a *next* pointer, using *null* to terminate the list. Setting up these two pointers adds a minor overhead compared with the setting up of the frame.

Implementation of the scheduler. To evaluate our scheduling algorithm, we developed a reference implementation that is based on an interpreter. Concretely, we write each of our benchmark programs as an abstract syntax tree (AST), whose leaves carry C++ functions describing the sequential blocks. At load time, the AST is flattened into a control flow graph that is ready to be executed by our interpreter. Of course, for use in production, compiler support would be desirable. We next describe the main features of our interpreter, focusing especially on the interactions with the cactus stack.

Heartbeat exhibits features common to parallel schedulers. In particular, each worker features a main loop for executing *ready* threads, i.e. threads that have no pending dependencies. To track dependencies, each thread stores a *join counter* to count the number of pending dependencies and a pointer to its join thread. When it terminates, a thread decrements the join counter of its join thread. In the case that the join counter reaches zero, the join thread becomes ready and gets added to a pool of ready threads.

To implement the heartbeat promotion mechanism, we rely on a combination of software polling and querying of the hardware cycle counter. Querying the hardware counter amounts to reading a register. Software polling is used in a number of other scheduling techniques, such as lazy scheduling and private-deque work stealing [3, 21, 55]. The implementation of software polling is a well-studied problem, with both hardware and software solutions [22, 49]. However, hardware polling based on interrupts is delicate to implement at the resolution of the order of $10\mu s$, but it may be possible given special hardware support. Fortunately, software polling can be implemented in an efficient manner via compiler (or manual) instrumentation of the code.

Given that Heartbeat is an interpreter, we implement polling in a simple way by inserting checks: (1) in between every sequential block being interpreted, (2) in between a fixed number of iterations for an innermost loop whose body does not trigger any parallel call, (3) after every iteration in other loops. For a production implementation of heartbeat scheduling, it should suffice to rely on one of the aforementioned solutions to polling.

In the remainder of this section, we describe the interactions between Heartbeat and the cactus stack. Each thread carries a code pointer and an instance of the cactus stack. Each cactus-stack instance consists of a pointer to the bottom-most frame of the stack, a pointer to the next free byte in the stack, and the head and tail pointers of the doubly linked list of promotable frames.

The heartbeat semantics initially performs machine transitions like the fully-sequential semantics (recall Section 3). Now, let us first describe the transitions involved in a non-parallel call. When it makes a function call, a thread pushes a frame on the current stack (possibly triggering the allocation of a new stacklet). When returning from the call, that frame gets popped. Eventually, the stack becomes empty, indicating that the thread has completed.

Before we explain the promotion transition, let us describe what would happened if our scheduler executed a parallel pair according to the fully-parallel semantics. In that case,

the scheduler would immediately create two new threads. The first is a thread for left branch. This thread reuses the current stack. This stack is then altered with the write of a null-pointer as parent-frame pointer in order to delimit the execution of the left branch. The second is a thread for the right branch. This thread is initialized with a fresh stack. The currently running thread is used to represent the join continuation, and as such its join counter is set to 2. This thread also carries the current stack, but note that this stack will be used only after the left branch completes.

In heartbeat scheduling, the evaluation of a parallel pair begins according to the fully-sequential semantics, pushing a frame in the stack to describe the right branch. Eventually, that frame might be subject to promotion. When this promotion happens, the scheduler creates two threads. The first thread is for the right branch, and uses a fresh stack. The second thread is for *what remains of* the left branch, whose execution has already begun. This thread captures the current stack, here again altered by nullifying the parent-frame pointer. The currently-running thread is reused to describe the join continuation, as described in the previous paragraph. In summary, the promotion process is similar to the eager creation of threads from the fully-parallel semantics, with one main difference: when a promotion occurs, the left-branch has already begun its execution. In addition to thread creation, a promotion operation also involves updating the head of the doubly linked list of promotable frames.

Native support for parallel loops. Finally, we describe, at a high level, our treatment of parallel loops. In theory, parallel loops can be encoded using fork-join, by creating a binary tree of threads. Past work gives this approach the name Eager Binary Splitting and identifies a number of limitations, such as manual tuning effort and poor portability across different inputs and hardware configurations [54].

Another possible approach is to introduce one frame for each parallel loop. While this approach would improve the situation, overheads might still be significant in the worst case. Consider, for example, a program featuring an outer loop whose body contains an inner parallel-loop that, on the input data provided, only runs for a couple iterations. In that case, each iteration of the outer loop would trigger the creation of a frame, yet would involve insufficient work to amortize the cost of that creation.

In Heartbeat, we chose to introduce frames for function calls only. Heartbeat provides dedicated support for the loop nests that may in function bodies. To describe a range of iterations to be performed within a loop nest, we use *loop descriptors* (a.k.a. task descriptors for nested loops). A loop descriptor consists of: (1) a code pointer, (2) for each loop enclosing that code pointer, the range of remaining iterations to be processed for these loops, and (3) a pointer to the frame storing the local variables involved in that code fragment.

For a promotion operation, the scheduler considers the outermost parallel loop with remaining iterations (in addition to the current iteration). It splits the range of that loop in half, leading to the creation of an independent loop descriptor describing the upper half of the split range. In addition, the scheduler creates one join thread per loop instance, but creates these threads only when the first promotion occurs. The exact implementation details are specific to Heartbeat and the fact that it is an interpreter.

5 Empirical Evaluation

We present a study of ten benchmarks taken from the Problem Based Benchmark Suite (PBBS) [8]. The benchmarks represent state-of-the-art algorithms for multicore architectures, solving problems on sequences, strings, and graphs, and in geometry and graphics.

The PBBS codes were implemented using Cilk Plus, a parallel version of C++ in which a few Cilk keywords are used as hints to express opportunities parallelism. The benchmarks include irregular parallel applications, where granularity control is particularly challenging. The authors' original code therefore relies on a number of manual techniques to control granularity, with careful engineering to select the techniques and hand-tune threshold settings. In particular, three main techniques are involved:

- A number of data-parallel loops were parallelized by systematically splitting input sequences into fixed-size blocks of 2048 items. This approach is used throughout the PBBS sequence library and used extensively by all PBBS benchmarks. This technique works well under certain assumptions. We replaced all such loops with Heartbeat's parallel-loop construct.
- In many cases outside of the sequence library, parallel loops were expressed using Cilk parallel for-loops. The algorithm underlying Cilk for-loops uses a heuristic that splits the loop range into $\min(8P, 2048)$ blocks, where P is the number of cores. This heuristic ensures that sufficient parallelism is created to feed all P cores. However, in an already-parallel context, it might end up creating an overwhelming number of threads. We replaced all such loops with Heartbeat's native parallel loop construct.
- A number of loops were forced to always make one spawn per iteration, by forcing the grain size to be 1. Doing so is crucial in situations where any nontrivial grain size may dramatically reduce parallelism, for example, in the case of an outermost parallel loop with potentially few iterations. We replaced such loops with Heartbeat's parallel-looping construct, simply dropping the grain size annotation.

In summary, the PBBS codes apply careful granularity control to ensure good performance. In contrast, the heartbeat approach uses a single, uniform method. Such uniformity greatly reduces the burden of performance tuning.

5.1 Benchmarking Environment

Input data and baseline setup. Our experiments use much of the same input data as was used in the original PBBS study [8], as well as some non-synthetic inputs that we added. Owing to space limitations, we summarize the input data in a technical appendix. We use as baselines for each benchmark the authors' original code. This code was tuned by the authors offline, on a collection of inputs, using a test machine similar to ours, and using GCC, like we did.

Hardware and software environment. We used an Intel machine with 40 cores, featuring four 10-core Intel E7-4870 chips, at 2.4GHz, with 32Kb of L1 and 256Kb L2 cache per core, 30Mb of L3 cache per chip, and 32GB RAM, and runs Ubuntu Linux kernel v3.13.0-66-generic. We compiled the code using GCC (version 6.3, options `-O2 -march=native`), using for PBBS Cilk Plus extensions (option `-fcilkplus`). Parallel runs involve a little bit of noise (with standard deviation usually around 3% to 5%). Thus for each data point we report the average over 30 runs.

We have also run our experiments on a 48-core AMD machine; the results—not shown, due to space limitation—are very similar to the ones presented throughout this section.

Load balancing. Cilk relies on work stealing implemented with concurrent deques [17, 26]. In contrast, Heartbeat currently supports three load-balancing algorithms: work stealing with concurrent deques, work stealing with private deques (as described in [3]), as well as a *mixed* variant that involves both a concurrent cell for storing the top-most deque item and a private deque for storing all other items.[1] Preliminary experiments suggest that the three variants give similar results, with a slight advantage for the mixed variant of work stealing. For this reason, we benchmark Heartbeat using this mixed variant.

Setting for the parameter N. Recall that the parameter N controls the pace at which promotions are performed in heartbeat scheduling. We illustrate in Figure 7 the effect of the choice of N through two examples that are representative of what we have observed over a range of benchmarks.

As the figure shows, values of N below $10\mu s$ are suboptimal due to significant parallelism overheads (overparallelization). At the same time, values of N above $100\mu s$ are also suboptimal, due to poor utilization (underparallelization). The exact point at which performance degrades for large values of N depends on the benchmark considered, but all programs ultimately suffer from too-large values of N. Somewhere in between, for values of N close to $30\mu s$, we find the sweet spot that we seek. The existence of a sweet spot, as predicted by the theoretical analysis, enables Heartbeat to deliver bounded overheads (e.g., 3% or 5%), while preserving as much parallelism as possible.

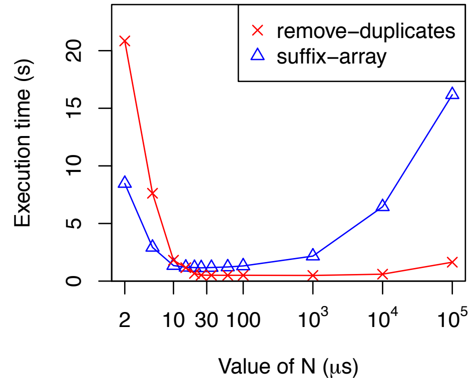

Figure 7. Impact of varying the value of the parameter N on the 40-core run time for two sample PBBS benchmarks.

As explained in Section 2, the value of N is a system-wide setting that should be set once-and-for-all to a multiple of the value of τ. Recall, the value τ denotes the cost of thread creation. For example, to ensure overheads below 5%, it suffices to set the value of N to 20τ. We next describe a simple protocol for measuring the value of τ on a given architecture. Although this protocol requires performing measures on one particular benchmark program, all benchmark programs should yield similar estimates.

The protocol for measuring τ requires performing single-core runs of a parallel program. First, execute the program with a very large value of N (e.g., $10^7\mu s$), so as to generate zero (or few) threads, and measure the execution time T. Then, execute the same program with a small value of N for which overheads are significant (e.g., $1\mu s$). For that run, let T' denote the execution time, and let C denote the number of threads created. The ratio $\frac{T'-T}{C}$ gives a good estimate of the cost of creating one thread.

On our machine, following this protocol on several benchmark programs, we systematically measured values for τ between $1.2\mu s$ and $1.9\mu s$, with an average close to $1.5\mu s$. Hence, to target overheads below 5%, we set $N = 30\mu s$.

5.2 Benchmark Results

Overheads of the Heartbeat interpreter. We implemented Heartbeat as an interpreter for basic blocks, and thus we pay a cost for interpretive overhead. To evaluate these overheads, we compare the *sequential elision* of the original PBBS program (using Cilk) and its Heartbeat counterpart. The sequential elision is a mode in which no parallel threads are created. More precisely, a sequential elision in Cilk replaces parallel function calls with conventional calls: (`cilk_spawn` and `cilk_sync` become no-ops) and the parallel for-loop `cilk_for` becomes the ordinary, sequential `for`. Regarding

[1] The mixed variant of work stealing that we consider here benefits from reduced latency for serving steals. The structure requires a local CAS only for acquiring the last item locally available, all other deque operations require no atomic operations. Each successful steal involves a single CAS operation. The structure involves polling on the top-most cell for populating that cell when it becomes empty as a result of a successful steal.

1	2	3	4	5	6	7	8	9
	Sequential elision		1-core execution		40-core execution			
Application/input	PBBS	Heartbeat	PBBS	Heartbeat	PBBS	Heartbeat	Heartbeat / PBBS	
	(s)		(relative to elision)		(s)		Idle time	Nb threads
radixsort								
random	3.39	+8.6%	+0.5%	+1.5%	0.21	-8.4%	-6.8%	-94.2%
exponential	3.46	+6.9%	+0.1%	+2.0%	0.20	-9.0%	-7.9%	-95.7%
random pair	5.63	+5.0%	+46.3%	+1.7%	0.51	-24.8%	-25.3%	-94.1%
samplesort								
random	22.77	+43.5%	+64.9%	+1.8%	1.21	-14.6%	-14.0%	-8.9%
exponential	16.48	+34.8%	+59.4%	+1.3%	0.91	-18.9%	-18.2%	-35.5%
almost sorted	7.55	+108.3%	+137.4%	+1.9%	0.71	-17.9%	-16.4%	-47.7%
suffixarray								
dna	23.62	+1.5%	+22.7%	+3.1%	1.33	-10.4%	-14.7%	-98.0%
etext	85.96	+3.5%	+25.5%	+2.6%	4.08	-6.2%	-8.4%	-90.7%
wikisamp	75.90	+2.4%	+21.4%	+2.7%	3.65	-5.9%	-7.9%	-89.5%
removeduplicates								
random	10.12	+33.4%	+11.5%	+3.6%	0.48	+9.5%	+11.3%	-46.2%
bounded random	2.97	+15.2%	-3.1%	-0.2%	0.22	-16.5%	-11.2%	-93.2%
exponential	7.08	+8.2%	+0.4%	+3.0%	0.38	-7.4%	-4.7%	-81.1%
string trigrams	11.27	+0.0%	-0.1%	+2.1%	0.54	-25.1%	-23.5%	+1.6%
convexhull								
in circle	14.27	-0.8%	-1.7%	+2.8%	0.69	-6.5%	-4.6%	-67.0%
kuzmin	8.65	+5.0%	-0.5%	-2.7%	0.48	-16.4%	-7.7%	-92.6%
on circle	10.77	-1.1%	+192.3%	+4.3%	1.35	-63.9%	-65.0%	-68.4%
nearestneighbors								
kuzmin	26.66	+17.8%	+10.7%	+4.3%	1.35	+0.9%	-0.7%	-89.0%
plummer	32.80	+14.0%	+8.6%	+4.7%	2.35	-12.9%	+17.1%	-92.1%
delaunay								
in square	90.47	-2.3%	-1.1%	+1.5%	3.53	+5.7%	-14.7%	-96.7%
kuzmin	97.80	-1.0%	+0.3%	+2.7%	4.13	+6.1%	-12.8%	-98.1%
raycast								
happy	11.15	+8.9%	+6.1%	+3.3%	0.48	+1.3%	-2.5%	-92.8%
xyzrgb	359.79	+1.2%	+0.7%	+0.7%	9.25	+0.9%	+1.3%	-73.2%
mst								
cube	49.46	+17.0%	+25.7%	+1.7%	2.57	+0.0%	-9.2%	-91.5%
rMat24	44.11	+14.1%	+22.5%	+1.5%	2.40	-1.8%	-6.6%	-91.4%
spanning								
cube	13.74	+8.5%	+3.2%	+2.0%	0.68	-11.9%	+3.5%	-96.3%
rMat24	8.78	+8.8%	+3.4%	+2.1%	0.50	-7.5%	+3.2%	-96.5%

Figure 8. Benchmark results. Negative numbers indicate that Heartbeat is performing better. Column 3 gives an estimate of the interpretive overhead of Heartbeat. Column 4 gives a lower bound on the overheads of the original PBBS code, with figures relative to column 2. Column 5 gives an estimate of the thread-creation overheads in Heartbeat, with figures relative to column 3. For 40-core runs, column 6 gives PBBS execution time, and column 7 gives the Heartbeat figure relative to column 6. Columns 8 and 9 shows the ratios Heartbeat divided by PBBS for total idle time and for number of threads created.

Heartbeat, we simply set a flag to disable promotion. Concretely, Heartbeat's frames remain in the stack and never get promoted, and the innermost parallel loops in the benchmark codes are turned into purely sequential loops.

Column 3 from Figure 8 shows the relative performance difference between Heartbeat sequential elision and PBBS sequential elision. Figures vary greatly across the rows. Indeed, the interpretive overheads depend largely on the contents of the basic block containing the critical loops. In fact, when

there are several such critical loops, the overheads depend on the extent to which each critical loop is being exercised by the input data. Thus, even for the same benchmark program, the interpretive overheads may vary significantly with the input data (as, e.g., for sample-sort).

For most benchmarks, the overheads are below 20%. (Only sample-sort and one instance of remove-duplicates have greater interpretive overheads.) Such limited interpretive overheads are achieved thanks to the fact that the compiler is able to optimize every basic block independently of our interpreter. Overall, we believe that the performance penalty is small enough to draw a meaningful comparison against the original Cilk programs.

Overheads of thread creation. We next evaluate parallelism overheads both in PBBS and in Heartbeat. To evaluate the overheads of thread creation in the original PBBS codes, we compare the execution of the sequential elision of each program against the single-core execution of the Cilk parallel binary. Whereas the former eliminates all spawn-sync constructs and sequentializes all loops, the latter is slowed down by compiler instrumentation, including thread creation. The estimation of overheads in Cilk may be incomplete because the Cilk system in some places detects at runtime that there is only one active worker thread in the system. Nevertheless, the comparison should give us a lower bound on the parallelism overheads affecting Cilk programs. Column 4 from Figure 8 shows that these overheads can be significant in some benchmarks, in 10 cases (out of 28) over 25%, and in 2 cases over 100%.

To evaluate the overheads of thread creation in Heartbeat, we compare in a similar fashion the sequential elision against the single-core execution of the parallel code. This time, the only difference between the two programs is that the former never tries to promote the parallel fork points, while the latter does so at regular pace. Promotion events occur every $30\mu s$ in Heartbeat—a fraction more in practice, because Heartbeat waits for the first polling event beyond the $30\mu s$ time interval before actually performing a promotion. Column 5 from Figure 8 shows that, when setting N such that $\frac{\tau}{N} \approx 5\%$, the overheads of thread creation (and destruction) in Heartbeat executions are systematically below 5%, as desired.

In practice, overheads may be less than the theoretical upper bound. One example is a program that involves strands of sequential work that do not produce any parallelism for a duration exceeding N, in a context where there are no promotable frames left in the stack. In our benchmarks, Heartbeat overheads exceed 3% in only 6 cases.

Parallel execution time. We now compare the execution time of PBBS and Heartbeat on parallel runs involving 40 cores. Columns 6 and 7 from Figure 8 show the results. These results show that Heartbeat, despite its interpretive overheads, is able to match (or improve over) the performance of PBBS codes. Thus, Heartbeat delivers a runtime-based approach that makes unnecessary the manual selection of granularity control technique and the manual tuning involved in the original PBBS programs. Replacing the Heartbeat interpreter with a compiler-based implementation could only improve the execution time.

To gain further insight on the parallel-execution-time differences between PBBS and Heartbeat programs, we included two additional columns in Figure 8. Column 8 shows the ratio between idle time (counting periods during which workers are out of work) in Heartbeat and idle time in PBBS. The figures show that the idle time is of the same order of magnitude.[2] Column 9 shows the ratio between the number of threads created (i.e. the number of promotions) in Heartbeat and the number of threads created in PBBS. There, figures show that Heartbeat creates fewer threads, in many cases at least one order magnitude fewer. Taken together, these last two columns indicate that Heartbeat is able to achieve similar utilization despite generating manyfold fewer threads, and as a result, the Heartbeat running time often benefits from the correspondingly decreased overheads.

6 Related Work

Parallel Scheduling. Brent's theorem [14] gives the first bound for scheduling a parallel program with work W and span S on P processors as $\frac{W}{P} + S$ by showing that a "level-by-level" schedule would yield such a bound. Brent's result was later generalized to greedy schedulers, which do not allow a processor to idle if there is work that can be performed [7, 20]. Blumofe and Leiserson [13] show that randomized work stealing algorithm can generate greedy schedules for fork-join parallel programs, also when including certain scheduling costs, e.g., steals. Their results were subsequently generalized to broader classes of parallel programs by Arora et al [6]. The space consumption of various scheduling algorithms have also been studied [5, 11, 12, 45], as well as their locality properties [1, 2, 9, 10, 18, 39, 52]. Nearly all of the aforementioned work assumes that spawning a thread has unit or asymptotically constant cost.

Taming overheads of thread creation. Reducing the thread-creation overheads involves reducing either the number of threads created, or reducing the cost of creating a thread. The naive approach is to create, for each parallel fork, a thread for the right branch and a thread for the join continuation.

Cilk's *clone optimization* [26] avoids the cost of creating certain threads when both branches of a parallel fork execute on the same processor. In such a case, clone optimization reuses the current stack and avoids a synchronization operation before executing the join continuation.

[2] Because utilization in these benchmarks is generally between 80% and 99%, the total idle time represents less than 20% of the total execution time, thus a 20% change in idle time would affect the execution time by less than 4%.

Tzannes et al. propose *lazy scheduling*, a scheduling technique where the creation of parallel threads is guided by demand for parallel work [55]. That demand is estimated by observing the occupancy of the local deque. This heuristic can fail and sometimes increase the span significantly, as also remarked by Tzannes et al. The authors give an upper bound on execution time, but for the case of a single parallel loop taken in isolation, not for the whole program.

In fact, more generally, any approach that makes irrevocable sequentialization decisions could increase span significantly. For example, suppose that, at a time of high load, one processor P_i proceeds to sequentialize a large task and starts to work on it. Throughout we use the term "task" to refer to each branch (part) of a parallel fork. Assume that, soon afterwards, all other processors run out of work and complete executing the threads in P_i's work queue. There then remains only one active thread: the large task that P_i sequentialized. Thus, only one is processor working, even though that task could have been executed in parallel. The work of the sequentialized task, which is large, now directly factors into the span of the execution, which could have been small if the task were executed in parallel. A technique based on estimating the work of parallel calls, such as oracle-guided scheduling [4], could avoid this problem in some cases.

Granularity control using cost estimation. One way to tame the total overhead of thread creation is to control the granularity of threads, ensuring that each thread created holds a sizeable piece work. Perhaps the oldest technique for granularity control is to use a manual, programmer-inserted "cut-off" condition, that switches from a parallel to a sequential mode of execution. Cilk programmers sometimes annotate parallel `cilk_for` loops to batch several parallel iterations into a sequential unit of work—the exact batching factor can be made parametric in the number of processors. Iwasaki et al. present a static analysis technique for synthesizing cut-offs for divide-and-conquer functions in Cilk-style programs [37]. Such granularity control, while pragmatic, is necessarily brittle in the presence of irregular control-flow.

In contrast to static approaches, dynamic approaches may exploit valuable information depending on the input data. Duran et al. [19] propose a method for controlling granularity in nested parallel loops, relying on profiling information collected at runtime. For recursive algorithms, older work [47, 57] has proposed to make sequentialization decisions based on the height or depth of the recursion tree. Yet, as Iwasaki et al. point out [37], making irrevocable sequentialization decisions may significantly harm parallelism.

One way to ensure that sequentializing a task will not harm parallelism in irregular applications is to have some guarantee that sequentialized tasks are small. Lopez et al. [42] propose, in the context of logic programming, an approach by which the programmer annotates functions with asymptotic cost annotations. The cost annotations are evaluated online, by the runtime, to make sequentialization decisions. However, using the asymptotic cost alone can lead to poor outcomes, because, on modern processors, execution time depends also on constant factors. Constant factors can be large due to, e.g., effects of caching, pipelining, etc.

Oracle-guided scheduling [4] considers a cost model that accounts for the cost of thread creation. Under certain assumptions on the shape of the computation graph, the authors establish work and span bounds for nested-parallel programs. Its formal bounds take into account the overheads of making time predictions, as well as the gap between predicted and effective values of the execution time. The authors also provide an implementation strategy that requires the programmer to provide an asymptotic cost function for each parallel task and that performs run-time measurements to estimate the constant factors missing from the asymptotic notation. As discussed in Section 1, oracle-guided scheduling does not apply to highly irregular programs where size of parallel tasks are difficult to estimate in advance.

7 Conclusion

In the current state of the art, writing fast parallel programs requires extensive optimization and tuning to limit the overheads of parallelism. One reason is that the existing scheduling techniques do not provide tight theoretical or practical bounds on the cost of creating and destroying threads. In this paper, we show that such bounds are achievable for all nested parallel programs written in the fork-join model, both in theory and in practice.

To this end, we present an algorithm that controls the overheads of thread creation by restricting thread creation to occur at periodic intervals, in effect, the heartbeat of the computation. The insight is to promote, at each beat, a stack frame that holds potential for parallelism to a thread. We specify the technique by formalizing it as an abstract-machine semantics and proving that the overheads are always bounded by a small fraction of the sequential work, the cost of which is to slightly increase the span (and thus decrease parallelism).

Our implementation and experiments show that heartbeat scheduling can eliminate a variety of tuning parameters and heuristics and still remain competitive with the hand-optimized codes. One notable result is that heartbeat scheduling is able to reduce the number of created threads, sometimes by 90% relative to hand-tuned codes and do so without unnecessarily reducing parallelism and performance.

Acknowledgments

This research is partially supported by the European Research Council (grant ERC-2012-StG-308246).

References

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

[2] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The data locality of work stealing. *Theory of Computing Systems (TOCS)* 35, 3 (2002), 321–347.

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*.

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

[5] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, R. K. Shyamasundar, and Katherine A. Yelick. 2007. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*. 229–240. https://doi.org/10.1145/1248377.1248416

[6] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures (SPAA '98)*. ACM Press, 119–129.

[7] 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.

[8] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *PPoPP '12*. 181–192.

[9] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*. 355–366.

[10] Guy E. Blelloch and Phillip B. Gibbons. 2004. Effectively sharing a cache among threads. In *SPAA*. https://doi.org/10.1145/1007912.1007948

[11] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. 1999. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM* 46 (March 1999), 281–321. Issue 2.

[12] Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. Comput.* 27, 1 (1998), 202–229.

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

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

[15] F. Warren Burton and M. Ronan Sleep. 1981. Executing functional programs on a virtual tree of processors. In *Functional Programming Languages and Computer Architecture (FPCA '81)*. ACM Press, 187–194.

[16] 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 (OOPSLA '05)*. ACM, 519–538.

[17] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *SPAA '05*. 21–28.

[18] 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*. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/1378533.1378574

[19] 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.

[20] 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.

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

[22] Marc Feeley. 1993. Polling efficiently on stock hardware. In *Proceedings of the conference on Functional programming languages and computer architecture (FPCA '93)*. 179–187.

[23] Matthias Felleisen and Daniel P. Friedman. 1987. Control Operators, the SECD-Machine, and the Lambda-Calculus. In *Formal Description of Programming Concepts - III*, M. Wirsing (Ed.). Elsevier Science Publisher B.V. (North-Holland), 193–219.

[24] 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.

[25] Matthew Fluet, Mike Rainey, John H. Reppy, and Adam Shaw. 2008. Implicitly-threaded parallelism in Manticore. In *ICFP*. 119–130.

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

[27] Seth Copen Goldstein, Klaus Erik Schauser, and David Culler. 1995. Enabling Primitives for Compiling Parallel Languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. Troy, New York.

[28] 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.

[29] John Greiner and Guy E. Blelloch. 1999. A Provably Time-efficient Parallel Implementation of Full Speculation. *ACM Transactions on Programming Languages and Systems* 21, 2 (March 1999), 240–285.

[30] Adrien Guatto, Sam Westrick, Ram Raghunathan, and Umut A. Acarand Matthew Fluet. 2018. Hierarchical Memory Management for Mutable State. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM Press.

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

[32] E. A. Hauck and B. A. Dent. 1968. Burroughs' B6500/B7500 Stack Mechanism. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference (AFIPS '68 (Spring))*. ACM, New York, NY, USA, 245–251. https://doi.org/10.1145/1468075.1468111

[33] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. 2009. Backtracking-based load balancing. *Proceedings of the 2009 ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* 44, 4 (February 2009), 55–64. https://doi.org/10.1145/1594835.1504187

[34] 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 (LFP '94)*. 79–90.

[35] 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.

[36] Intel. 2011. Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/.

[37] 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.

[38] Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande (JAVA '00)*. 36–43.

[39] 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. https://doi.org/10.1145/2809808

[40] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. 2010. 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, New York, NY, USA, 411–420. https://doi.org/10.1145/1854273.1854324

[41] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. 227–242.

[42] 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. https://doi.org/10.1006/jsco.1996.0038

[43] Simon Marlow. 2013. *Parallel and Concurrent Programming in Haskell*. O'Reilly.

[44] 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.

[45] Girija J. Narlikar and Guy E. Blelloch. 1999. Space-Efficient Scheduling of Nested Parallelism. *ACM Transactions on Programming Languages and Systems* 21 (1999).

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

[47] 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.

[48] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *ICFP 2016*. ACM Press.

[49] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible architectural support for fine-grain scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS '10)*. ACM, New York, NY, USA, 311–322. https://doi.org/10.1145/1736020.1736055

[50] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*. 68–70.

[51] 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.

[52] 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 (SPAA '09)*. ACM, New York, NY, USA, 91–100.

[53] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. 2010. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *Symposium on Principles & Practice of Parallel Programming*. 179–190.

[54] 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.

[55] 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. https://doi.org/10.1145/2629643

[56] Leslie G. Valiant. 1990. A bridging model for parallel computation. *CACM* 33 (Aug. 1990), 103–111. Issue 8.

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

[58] Chaoran Yang and John Mellor-Crummey. 2016. A Practical Solution to the Cactus Stack Problem. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '16)*. ACM, New York, NY, USA, 61–70. https://doi.org/10.1145/2935764.2935787