JFP **26**, e23, 40 pages, 2016.
doi:10.1017/S0956796816000101

*Oracle-guided scheduling for controlling granularity in implicitly parallel languages**

UMUT A. ACAR
Carnegie Mellon University, Pittsburgh, PA, USA
Inria, Paris, France
(*e-mail:* umut@cs.cmu.edu)

ARTHUR CHARGUÉRAUD
Inria, Université Paris-Saclay, Palaiseau, France
LRI, CNRS & Univ. Paris-Sud, Université Paris-Saclay, Orsay, France
(*e-mail:* arthur.chargueraud@inria.fr)

MIKE RAINEY
Inria, Paris, France
(*e-mail:* mike.rainey@inria.fr)

Abstract

A classic problem in parallel computing is determining whether to execute a thread in parallel or sequentially. If small threads are executed in parallel, the overheads due to thread creation can overwhelm the benefits of parallelism, resulting in suboptimal efficiency and performance. If large threads are executed sequentially, processors may spin idle, resulting again in suboptimal efficiency and performance. This "granularity problem" is especially important in implicitly parallel languages, where the programmer expresses all potential for parallelism, leaving it to the system to exploit parallelism by creating threads as necessary. Although this problem has been identified as an important problem, it is not well understood—broadly applicable solutions remain elusive. In this paper, we propose techniques for automatically controlling granularity in implicitly parallel programming languages to achieve parallel efficiency and performance. To this end, we first extend a classic result, Brent's theorem (a.k.a. the work-time principle) to include thread-creation overheads. Using a cost semantics for a general-purpose language in the style of lambda calculus with parallel tuples, we then present a precise accounting of thread-creation overheads and bound their impact on efficiency and performance. To reduce such overheads, we propose an oracle-guided semantics by using estimates of the sizes of parallel threads. We show that, if the oracle provides accurate estimates in constant time, then the oracle-guided semantics reduces the thread-creation overheads for a reasonably large class of parallel computations. We describe how to approximate the oracle-guided semantics in practice by combining static and dynamic techniques. We require the programmer to provide the asymptotic complexity cost for each parallel thread and use runtime profiling to determine hardware-specific constant factors. We present an implementation of the proposed approach as an extension of the Manticore compiler for Parallel ML. Our empirical evaluation shows that our techniques can reduce thread-creation overheads, leading to good efficiency and performance.

* This research was partially supported by the National Science Foundation (grants CCF-1320563 and CCF-1408940), European Research Council (grant ERC-2012-StG-308246), and by Microsoft Research.

1 Introduction

Explicitly threaded programming provides full control over parallelism resources by offering primitives for creating, scheduling, and synchronizing parallel threads. As a result, the programmer can write efficient parallel programs by performing a careful cost-benefit analysis to determine which threads should be executed in parallel and under what conditions. This approach, however, requires reasoning about low-level execution details, such as the effects of scheduling, data races, or concurrent effects, which is known to be notoriously hard. In addition, such low-level programming can lead to over-engineering of software and harm portability. For example, the resulting code may perform well on a particular hardware setting but not on others.

The complexities of programming with explicitly threaded languages have motivated interest in *implicitly threaded* or *implicitly parallel* languages, such as Multilisp (Halstead, 1985), NESL (Blelloch *et al*., 1994), Cilk (Blumofe and Leiserson, 1999), Parallel Haskell (Chakravarty *et al*., 2007; Peyton Jones *et al*., 2008), Parallel ML (PML) in Manticore (Fluet *et al*., 2008; Fluet *et al*., 2011), and in MultiMlton (Sivaramakrishnan *et al*., 2014). In explicit threading, all programs accept a sequential (serial) semantics. It is the responsibility of the run-time system of the programming language to fulfill the intended semantics by creating and scheduling threads as needed. Implicit threading can therefore simplify writing parallel programs significantly. For example, the programmer can express parallelism at a high level by using parallel versions of a variety of serial data types such as sequences, arrays, and tuples.

As an implicit parallel program executes, it exposes opportunities for parallelism. The language run-time system exploits the potential for parallelism by creating lightweight threads (variously called tasks, strands, fibers, etc.) as needed and by mapping them to the processors or cores for fast execution. To achieve efficiency, implicit programming languages rely on a scheduler for distributing threads among the processors to perform load balancing. Many scheduling techniques and practical schedulers have been developed, including work-stealing schedulers (e.g., Blumofe and Leiserson (1999)), and depth-first-search schedulers (Blelloch and Greiner, 1996).

Experience with implicitly parallel programs shows that one of the most important decisions that any implicitly parallel language must make is determining whether to exploit an opportunity for parallelism by creating a parallel thread or to forego the opportunity by falling back to sequential execution. On the one hand, creating a thread for each parallelism opportunity can lead to poor efficiency, because the cost of creating and managing parallel threads can be very high. On the other hand, foregoing a parallelism opportunity can lead to sub-optimal performance because of the lost opportunity for parallelization of a large chunk of work. Therefore, solving this dilemma requires finding just the right "sweet spot", where no more than necessary threads are created to achieve the best performance. This *granularity problem* is important because the overhead of managing parallelism matters: Since the speedups achievable via parallel computation is bounded by the number of processors, often a small constant factor, any increase in the overhead can impact performance negatively.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

Even though the granularity problem is broadly accepted to be an important problem, it is a poorly understood one. Theoretical analyses often ignore thread-creation overheads, offering no significant clues about how such overheads may affect efficiency. Practical implementations often focus on reducing thread-creation overheads, instead of controlling granularity so that fewer threads can be created without harming parallelism. As a result, practitioners try to solve the granularity problem by estimating the amount of work that would be sufficiently large to justify parallel execution. More specifically, programmers try to determine the input sizes at which threads become too small to amortize the cost of parallel thread creation and sequentialize such threads. Since the running time of a thread depends on the hardware, the programmer must make the best decision they can by taking into account the specifics of the hardware. This manual granularity control is thus bound to yield code whose performance is specific to the hardware and therefore likely not portable (Tzannes *et al*., 2014).

In this paper, we propose theoretical and practical techniques for solving the granularity problem in implicitly parallel programming languages. First, we present theorems that take into account thread-creation overheads and characterize their impact on parallel run time. Our theorems show such impact to be significant (Section 4). We then consider a cost semantics for a calculus that extends the lambda calculus with parallel tuples (Section 5) and propose a technique for controlling granularity based on an oracle (Section 6). We show that if the oracle can be implemented efficiently and accurately, it can be used to improve efficiency for a relatively large class of computations.

Based on this result, we propose a practical realization of the oracle that uses complexity functions defined by the user to approximate accurately the work of the computations involved (Section 7). We then describe a compilation scheme for implementing the oracle-guided semantics based on such complexity functions (Section 8). We present an evaluation of the proposed approach on a subset of ML extended with parallel tuples and complexity annotations (Section 9). We conclude with a discussion of the limitations of our implementation of the oracle based on complexity functions (Section 11).

2 Overview

We present a high-level overview of our the techniques proposed in this paper.

Brent's theorem (Brent, 1974), which leads to the *work-time scheduling principle*, characterizes arguably the most important property of parallel programs: that they can be scheduled to execute efficiently with multiple processors—within a factor of two of the optimal. More precisely, Brent shows that a level-by-level schedule can execute a parallel computation with w work and s span in no more than $w/P + s$ steps on P processors. This theorem generalizes to any greedy schedule, that is, any schedule that never leaves processors idle when there is work to do. Prior research shows that greedy schedules can be computed by online scheduling algorithms such as the work-stealing algorithm (e.g., Blumofe and Leiserson (1999)) under modest assumptions.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

The work-time scheduling principle ignores an important factor: The overhead for creating threads or parallelism, which is assumed to be zero. To understand the impact of thread-creation overheads, we start with this fundamental theorem and generalize it to take the overheads into account (Section 4). Specifically, we consider the standard directed-acyclic-graph (DAG) mode for parallel computations and show that a computation with *total work* $\mathbb{W}$ and *total span* $\mathbb{S}$, where both include the thread-creation overheads, can be executed in no more than $\mathbb{W}/P + \mathbb{S}$ steps. What is somewhat special about thread-creation overheads are that they are not divisible. Nevertheless, the generalized theorem follows by a modest extension of an existing proof.

Having established the contribution of thread-creation overheads to parallel run time, we then move on to the problem of determining precisely the overheads of thread creation in implicitly parallel programs. To this end, we consider a lambda calculus with parallel tuples and present a cost-semantics for evaluating expression of this language (Section 5). The *cost semantics* yield raw work/span and total work/span of each evaluated expression. Using this cost semantics, we show that thread-creation overheads can be significant: A multiplicative factor times the raw work. When applied to the generalized Brent's theorem, this result implies that such multiplicative increases in work affect the parallel run-time directly. To reduce thread-creation overheads, we propose an alternative *oracle-guided semantics* that captures a known principle for avoiding the thread-creation overheads: For a thread, create a parallel thread and evaluate it in parallel only if the thread is sufficiently large, i.e., greater than some constant κ. We show that the oracle semantics can decrease the overheads of thread-creation by any (desired) constant factor κ, but only at the cost of increasing the total span by a similar factor. This result suggests that in practice, some care will be needed to select κ, because otherwise it can reduce an important quantity called parallel slackness (Valiant, 1990).

The bounds with the oracle-guided semantics suggest that we can reduce the thread-creation overheads significantly, if we can realize the semantics in practice. Such a realization is impossible, unfortunately, because it requires the ability to determine *a priori* the running time of a thread and do so without incurring other overheads. We show, however, that a realistic oracle that can give constant-factor approximations to the thread running times can still result in similar reductions in the overheads for a reasonably broad class of computations (Section 6.2). We also show that, unless care is taken, the realistic oracle can actually further increase the overheads, due to the direct cost of evaluating the oracle. This outcome, i.e., that attempts at controlling the granularity can actually backfire and slow down the program further, is an interesting outcome of our analysis. For a broad class of computations, including many recursive, divide-and-conquer computations, we show that this worst case can be avoided.

While the oracle-guided semantics is effective in controlling the cost of thread creation without detrimentally harming parallelism, it is technically impossible to realize in practice because it requires predicting the work (sequential run-time) of computations. As we describe in Section 7, however, the work to be performed by a given thread can be approximated by using a combination of static and dynamic

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

information. Specifically, we describe an approximation technique that relies on an estimator that uses asymptotic cost functions (asymptotic complex bounds) and judicious use of run-time profiling techniques to estimate actual run-times accurately and efficiently. This approach combines data from the asymptotic complexity bounds and the profiling techniques to approximate the work to be performed by a given thread. In this work, we only consider simple recursive functions for which the execution time is proportional to the value obtained by evaluating the asymptotic complexity expression. We refer to Section 11 for discussion of more general patterns.

We present a prototype implementation of the proposed approach (Section 8) by extending the ML to support parallel tuples, and complexity functions, and translating programs written in this extended language to the PML language (Fluet *et al*., 2011). Although our implementation requires the programmer to enter the complexity information, these could also be inferred in some cases cases via static analysis (e.g., Jost *et al.* (2010) and references thereof). We extend the Manticore compiler for PML to support oracle-guided scheduling and use it to compile generated PML programs. Our experiments (Section 9) show that our oracle implementation can reduce the overheads of a single processor parallel execution to 3% and 13% of the sequential time. When using 16 processors, we achieve 7- to 15-fold speedups on our benchmark machine.

3 Terminology

In this paper, we consider implicitly parallel programs, where a program executes by employing two kinds of threads: system-level threads and user-level threads. An execution creates one *system-level thread* per processor and usually pins the system-level thread to that processor. It then dynamically creates lighter weight, *user-level threads*, and maps them to the system-level threads. A key property of implicitly parallel programs is that the number of user-level threads created during an execution can be very large relative to the number of system-level threads. For example, an execution may create millions of user-level threads mapped to 10 system-level threads, which are then mapped to 10 processors or cores by the system. For the purpose of succinctness, throughout the paper, we refer to "user-level threads" simply as "threads" and to "system-level threads" as "processes".[1]

4 Generalizing Brent's theorem

We represent a parallel computation with a DAG, called *computation DAG*. Vertices in the DAG represent atomic computations, or operations. For simplicity, we refer to each vertex as an *operation*. Edges between operations represent precedence relations, in the sense that an edge from u to v indicates that the execution of u must be completed before the execution of v can start. Every computation DAG includes a *source* operation and a *sink* operation, representing the starting and the

[1] User-level threads are sometimes called "strands"; unfortunately, the same term is sometimes used to refer to hardware threads.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

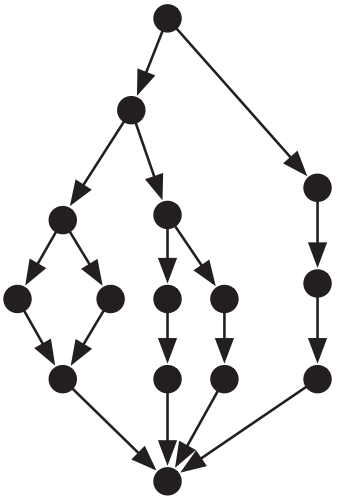

Fig. 1. An example computation DAG.

end points of the computation, respectively. All operations of a computation DAG are reachable from the source, and the sink is reachable from all operations. An example computation DAG appears in Figure 1.

Traditionally, every operation in the DAG is considered to take a single unit of time and given a weight of 1. In this setting, we can define the standard notion of work and span, which we here call *raw work* and *raw span*. The *raw work* of a computation graph is equal to the total number of operations that it contains. The *raw span* of the computation DAG is equal to the total number of operations along the longest path. Brent proved the following bound, which we recall since our aim is to generalize it.

Theorem 4.1 (*Brent's theorem*)
Let G be a computation DAG with w raw work and s raw span. A level-by-level or depth-wise schedule of G requires $O(\frac{w}{P} + s)$ time on P processors.

Proof
Consider scheduling the operations *level-by-level*, that is, starting with operations at depth 1, then executing operations at depth 2, and so on, ending with operations at depth s. Let w_i denote the number of operations at depth i in the DAG. These operations can be scheduled on P processors in $\lceil \frac{w_i}{P} \rceil$ rounds, each lasting exactly one unit of time. Summing up over all the possible depths, we deduce that the total execution time is bounded by

$$\sum_{i=1}^{s} \left\lceil \frac{w_i}{P} \right\rceil \leqslant \sum_{i=1}^{s} \left(\frac{w_i}{P} + 1 \right) \leqslant \frac{\sum_{i=1}^{s} w_i}{P} + s \leqslant \frac{w}{p} + s. \qquad \square$$

Observe that the optimal execution time cannot be less than $\frac{w}{P}$, which corresponds to having all processors busy at all time, and that it cannot be less than s, which

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

corresponds to the length of a critical path. Therefore, Brent's bound, expressed as $\frac{w}{p} + s$, is never more than a factor 2 away from the optimal.

Brent's theorem does not take into account the overheads associated with thread creation. Our goal is to refine the model and generalize Brent's theorem to take thread-creation costs into account. To that end, we assign to each operation with out-degree two or greater au weight of $1 + \tau$ instead of just 1. We then define the *total work* as the sum of the weights of all the operations in this revised computation DAG. Similarly, we define the *total span* as the maximum weight of a path from the source to the sink in the revised DAG.

Theorem 4.2 generalizes Brent's theorem for weighted DAGs and arbitrary greedy schedules (rather that level-by-level schedules). The proof is a relatively straightforward generalization of that of Arora *et al.* (1998).

Theorem 4.2 (*Greedy-scheduling of weighted DAGs*)
Let G be a computation DAG with $\mathbb{W}$ total work and $\mathbb{S}$ total span. Any greedy P processors schedule of G takes $O(\frac{\mathbb{W}}{P} + \mathbb{S})$ steps.

Proof
Consider any greedy execution of the DAG G. At each execution step, each processor places a token in the *work bucket* if it is busy at this step, otherwise it places a token in the *idle bucket*. The work bucket contains exactly $\mathbb{W}$ tokens at the end of the execution. Let I denote the number of tokens contained in the idle bucket at the end of the execution, and let T denote the total number of steps in the execution. Because a total TP tokens are created during the T execution steps, $TP = \mathbb{W} + I$. In order to establish the result $T \leqslant \frac{\mathbb{W}}{P} + \mathbb{S}$, it thus suffices to establish the inequality $I \leqslant P\,\mathbb{S}$.

Consider a given time step. If all processors execute an operation at that step, then the idle bucket receives zero tokens. Otherwise, a number of processors are idle. In this case, the idle bucket receives between one and $P - 1$ tokens. If one or more processors are idle, then the idle processors cannot find a ready operation to execute, because otherwise this would not be a greedy schedule. This means that at this step, all the ready operations (operations whose ancestor have been executed) are executed. Therefore, at such a time step, the span of the sub-DAG induced by the operations that are not yet executed decreases by one. Therefore, there are at most $\mathbb{S}$ such steps and thus $(P - 1)\,\mathbb{S}$ tokens in the idle bucket. □

5 Source language and cost semantics

To give an accurate account of cost of thread creation, and to specify later our compilation strategy, we consider a source language in the style of λ-calculus and present a dynamic cost semantics for it. The semantics and the costs are parameterized by τ and ϕ that represent the cost of creating a parallel thread and the cost of consulting an external oracle.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

$$
\begin{array}{lcl}
v & ::= & x \mid \mathtt{n} \mid (v,v) \mid \mathtt{inl}\ v \mid \mathtt{inr}\ v \mid \mathtt{fun}\ f.x.e \\
e & ::= & v \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid (v\ v) \mid \mathtt{fst}\ v \mid \mathtt{snd}\ v \mid \\
 & & \mathtt{case}\ v\ \mathtt{of}\ \{\mathtt{inl}\ x.e, \mathtt{inr}\ x.e\} \mid (e,e) \mid (|e,e|)
\end{array}
$$

Fig. 2. Abstract syntax of the source language.

5.1 Syntax

The source language includes recursive functions, pairs, sum types, and parallel tuples. Parallel tuples enable expressing computations or *branches* that can be performed in parallel, similar to the fork-join or nested data parallel computations. More precisely, in $(|e_1, e_2|)$, we refer to e_1 and e_2 as (parallel) branches. We note that although we only consider parallel tuples of arity two, our results generalize to fixed (constant) arity trivially.

To streamline the presentation, we assume programs to be in A-normal form, with the exception of pairs and parallel pairs, which we treat symmetrically because our compilation strategy involves translating parallel pairs to sequential pairs. Figure 2 illustrates the abstract syntax of the source language. We note that, even though the presentation is only concerned with a purely functional language, it is easy to add local mutable state (i.e., mutable memory cells that are not subject to concurrent accesses); in this case, however, they contribute no additional insight and thus are omitted for clarity.

5.2 Dynamic cost semantics

We define a dynamic semantics where parallel tuples are evaluated selectively either in parallel or sequentially, as determined by their relative size compared with some constant κ, called the cut-off value and such that $\kappa \geqslant 1$. To model this behavior, we present an evaluation semantics that is parameterized by an identifier that determines the *mode* of execution, i.e., sequential or not. For the purpose of comparison, we also define a *(fully) parallel* semantics where components of a parallel tuple are always evaluated in parallel regardless of their size. The *mode* of an evaluation is sequential (written seq), parallel (written par), or oracle (written orc). We let α range over modes:

$$\alpha \quad ::= \quad \mathsf{seq} \mid \mathsf{par} \mid \mathsf{orc}.$$

In addition to evaluating expression, the dynamic semantics also returns cost measures including *raw work* and *raw span* denoted by w and s (and variants), and *total work* and *total span*, denoted by $\mathbb{W}$ and $\mathbb{S}$ (and variants). Dynamic semantics, whose inductive definition appears in Figure 3, is presented in the style of a natural (big-step) semantics and consists of evaluation judgments of the form

$$e \Downarrow^{\alpha} v, (w,s), (\mathbb{W},\mathbb{S}).$$

This judgment states that evaluating expression e in mode α yields value v resulting in raw work of w and raw span of s and total work of $\mathbb{W}$ and total span of $\mathbb{S}$.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

(value)

$$\frac{}{v \Downarrow^{\alpha} v, (1,1), (1,1)}$$

(let)

$$\frac{e_1 \Downarrow^{\alpha} v_1, (w_1, s_1), (\mathbb{W}_1, \mathbb{S}_1) \qquad e_2[v_1/x] \Downarrow^{\alpha} v, (w_2, s_2), (\mathbb{W}_2, \mathbb{S}_2)}{(\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2) \Downarrow^{\alpha} v, (w_1 + w_2 + 1, s_1 + s_2 + 1), (\mathbb{W}_1 + \mathbb{W}_2 + 1, \mathbb{S}_1 + \mathbb{S}_2 + 1)}$$

(app)

$$\frac{(v_1 = \texttt{fun}\ f.x.e) \qquad e[v_2/x, v_1/f] \Downarrow^{\alpha} v, (w, s), (\mathbb{W}, \mathbb{S})}{(v_1\ v_2) \Downarrow^{\alpha} v, (w+1, s+1), (\mathbb{W}+1, \mathbb{S}+1)}$$

(first)

$$\frac{}{(\texttt{fst}\ (v_1, v_2)) \Downarrow^{\alpha} v_1, (1,1), (1,1)}$$

(second)

$$\frac{}{(\texttt{snd}\ (v_1, v_2)) \Downarrow^{\alpha} v_2, (1,1), (1,1)}$$

(case-left)

$$\frac{e_1[v_1/x_1] \Downarrow^{\alpha} v, (w, s), (\mathbb{W}, \mathbb{S})}{\texttt{case}\ (\texttt{inl}\ v_1)\ \texttt{of}\ \{\texttt{inl}\ x_1.e_1, \texttt{inr}\ x_2.e_2\} \Downarrow^{\alpha} v, (w+1, s+1), (\mathbb{W}+1, \mathbb{S}+1)}$$

(case-right)

$$\frac{e_2[v_2/x_2] \Downarrow^{\alpha} v, (w, s), (\mathbb{W}, \mathbb{S})}{\texttt{case}\ (\texttt{inr}\ v_2)\ \texttt{of}\ \{\texttt{inl}\ x_1.e_1, \texttt{inr}\ x_2.e_2\} \Downarrow^{\alpha} v, (w+1, s+1), (\mathbb{W}+1, \mathbb{S}+1)}$$

(tuple)

$$\frac{e_1 \Downarrow^{\alpha} v_1, (w_1, s_1), (\mathbb{W}_1, \mathbb{S}_1) \qquad e_2 \Downarrow^{\alpha} v_2, (w_2, s_2), (\mathbb{W}_2, \mathbb{S}_2)}{(e_1, e_2) \Downarrow^{\alpha} (v_1, v_2), (w_1 + w_2 + 1, s_1 + s_2 + 1), (\mathbb{W}_1 + \mathbb{W}_2 + 1, \mathbb{S}_1 + \mathbb{S}_2 + 1)}$$

(ptuple-seq)

$$\frac{e_1 \Downarrow^{\mathsf{seq}} v_1, (w_1, s_1), (\mathbb{W}_1, \mathbb{S}_1) \qquad e_2 \Downarrow^{\mathsf{seq}} v_2, (w_2, s_2), (\mathbb{W}_2, \mathbb{S}_2)}{(|e_1, e_2|) \Downarrow^{\mathsf{seq}} (v_1, v_2), (w_1 + w_2 + 1, s_1 + s_2 + 1), (\mathbb{W}_1 + \mathbb{W}_2 + 1, \mathbb{S}_1 + \mathbb{S}_2 + 1)}$$

(ptuple-par)

$$\frac{\begin{array}{c} e_1 \Downarrow^{\mathsf{par}} v_1, (w_1, s_1), (\mathbb{W}_1, \mathbb{S}_1) \\ e_2 \Downarrow^{\mathsf{par}} v_2, (w_2, s_2), (\mathbb{W}_2, \mathbb{S}_2) \end{array}}{(|e_1, e_2|) \Downarrow^{\mathsf{par}} (v_1, v_2), (w_1 + w_2 + 1, \max(s_1, s_2) + 1), (\mathbb{W}_1 + \mathbb{W}_2 + 1 + \tau, \max(\mathbb{S}_1, \mathbb{S}_2) + 1 + \tau)}$$

(ptuple-orc-parallelize)

$$\frac{\begin{array}{c} w_1 \geq \kappa \wedge w_2 \geq \kappa \\ e_1 \Downarrow^{\mathsf{orc}} v_1, (w_1, s_1), (\mathbb{W}_1, \mathbb{S}_1) \\ e_2 \Downarrow^{\mathsf{orc}} v_2, (w_2, s_2), (\mathbb{W}_2, \mathbb{S}_2) \end{array}}{(|e_1, e_2|) \Downarrow^{\mathsf{orc}} (v_1, v_2), (w_1 + w_2 + 1, \max(s_1, s_2) + 1), (\mathbb{W}_1 + \mathbb{W}_2 + 1 + \tau + \phi, \max(\mathbb{S}_1, \mathbb{S}_2) + 1 + \tau + \phi)}$$

(ptuple-orc-sequentialize)

$$\frac{\begin{array}{c} w_1 < \kappa \vee w_2 < \kappa \\ e_1 \Downarrow^{(\mathsf{if}\ w_1 < \kappa\ \mathsf{then\ seq\ else\ orc})} v_1, (w_1, s_1), (\mathbb{W}_1, \mathbb{S}_1) \\ e_2 \Downarrow^{(\mathsf{if}\ w_2 < \kappa\ \mathsf{then\ seq\ else\ orc})} v_2, (w_2, s_2), (\mathbb{W}_2, \mathbb{S}_2) \end{array}}{(|e_1, e_2|) \Downarrow^{\mathsf{orc}} (v_1, v_2), (w_1 + w_2 + 1, s_1 + s_2 + 1), (\mathbb{W}_1 + \mathbb{W}_2 + 1 + \phi, \mathbb{S}_1 + \mathbb{S}_2 + 1 + \phi)}$$

Fig. 3. Dynamic cost semantics.

Figure 3 shows the complete inductive definition of the dynamic cost semantics judgment $e \Downarrow^{\alpha} v, (w, s), (\mathbb{W}, \mathbb{S})$. When evaluating any expression that is not a parallel tuple, we calculate the (raw or total) work and the (raw or total) span by summing

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

up those of the premises (sub-expressions) and adding one unit to include the cost of the judgment. For all expressions, including parallel tuples, each evaluation step contributes 1 to the raw work or raw span. When calculating total work and total span, we take into account the cost of creating a parallel thread τ and the cost of making an oracle decision ϕ.

Evaluation of parallel tuples vary depending on the mode.

- **Sequential mode.** Parallel tuples are treated exactly like sequential tuples: Evaluating a parallel tuple simply contributes 1 to the raw and the total work (span), which are computed as the sum of the work (span) of the two branches plus 1. In the sequential mode, raw and total work (span) are the same.
- **Parallel mode.** The evaluation of parallel tuples induces an additional constant cost τ. The span is computed as the maximum of the spans of the two branches of the parallel tuple plus 1, and work is computed as the sum of the work of the two branches plus τ.
- **Oracle mode.** The scheduling of a parallel tuple depends on the amount of raw work involved in the two branches. If the raw work of each branch is more than κ, then the tuple is evaluated in parallel in the oracle mode. Otherwise, the raw work of at least one branch is less than κ, and the tuple is evaluated sequentially. In this case, the evaluation mode of each branch depends on the work of the branch. If a branch contains more than κ units of raw work, then it is evaluated in oracle mode, otherwise it is evaluated in sequential mode. This switch to sequential mode on small branches ensures that the oracle is not called too frequently during the evaluation of a program.
  If the parallel tuple is evaluated sequentially, then the raw/total work and span are both calculated as the sum of the span of the branches plus one. If the parallel tuple is evaluated in parallel, then an extra τ is included in the total work and span and the span is computed as the maximum of the span of the two branches.

Theorem 5.1 makes it possible to apply directly the greedy-scheduling theorem to the cost semantics. The basic idea of the proof is to show a correspondence between the cost semantics and DAGs.

Theorem 5.1 (*Greedy-scheduling for the cost semantics*)
Assume $e \Downarrow^{\mathsf{orc}} v, (w, s), (\mathbb{W}, \mathbb{S})$ to hold for some v, w, and s. Any greedy scheduler executes the expression e in no more than $\frac{\mathbb{W}}{P} + \mathbb{S}$ computations steps on P processors.

Proof
In order to invoke Theorem 4.2, which applies to computation DAGs, we build the computation DAG associated with the execution of the expression e, including vertices that represent the cost of scheduling. To that end, we describe a recursive algorithm for turning an expression e with total work $\mathbb{W}$ and total span $\mathbb{S}$ into the corresponding computation DAG, in which the sum of the weights of the vertices is equal to $\mathbb{W}$, and the maximal weight of a path is $\mathbb{S}$. The algorithm follows the structure of the derivation that e has total work $\mathbb{W}$ and total span $\mathbb{S}$.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

- If the last rule has zero premises, then e is an atomic expression and $\mathbb{W} = \mathbb{S} = 1$. We build the corresponding DAG as a single vertex of unit weight.
- If the last rule has one premise, then $\mathbb{W}$ takes the form $\mathbb{W}_1 + 1$ and $\mathbb{S}$ takes the form $\mathbb{S}_1 + 1$. Let G_1 be the DAG corresponding to the sub-expression described in the premise. We build G by extending G_1 with one unit-weight vertex at the bottom, that is, by sequentially composing G_1 with a DAG made of a single vertex.
- Otherwise, the last rule has two premises. First, consider the case where e is a let-expression. $\mathbb{W}$ takes the form $\mathbb{W}_1 + \mathbb{W}_2 + 1$ and $\mathbb{S}$ takes the form $\mathbb{S}_1 + \mathbb{S}_2 + 1$. Let G_1 and G_2 be the DAGs corresponding to the two sub-expressions. We build G by sequentially composing G_1 with a single unit-weight vertex and then with G_2.
- Consider now the case of a parallel tuple that is sequentialized. $\mathbb{W}$ takes the form $\mathbb{W}_1 + \mathbb{W}_2 + 1 + \phi$ and $\mathbb{S}$ takes the form $\mathbb{S}_1 + \mathbb{S}_2 + 1 + \phi$. Let G_1 and G_2 be the DAGs corresponding to the two branches. We build G by sequentially composing a vertex of weight $1 + \phi$ with the sequential composition of G_1 and G_2.
- Finally, consider the case of a parallel tuple that is parallelized. $\mathbb{W}$ takes the form $\mathbb{W}_1 + \mathbb{W}_2 + 1 + \tau + \phi$ and $\mathbb{S}$ takes the form $\mathtt{max}(\mathbb{S}_1, \mathbb{S}_2) + 1 + \tau + \phi$. Let G_1 and G_2 be the DAGs corresponding to the two branches. We build G by sequentially composing a vertex of weight $1 + \tau + \phi$ with the parallel composition of G_1 and G_2.

It is straightforward to check that, in each case, $\mathbb{W}$ and $\mathbb{S}$ match the sum of the weights of the vertices and the total span of the DAG being produced. □

6 Work, span, and execution time analysis

We analyze the impact of thread-creation overheads on parallel execution time and show how these costs can be reduced dramatically by using our oracle semantics. For our analysis, we first consider an *ideal oracle* that always makes perfectly accurate predictions (about the raw work of expressions) without any overhead (i.e., $\phi = 0$). Such an ideal oracle is unrealistic, because it is practically impossible to determine perfectly accurately the raw work of computations. We therefore consider a realistic oracle that approximates the raw work of computations by performing constant work. Our main result is a theorem that shows that the realistic oracle can reduce the thread-creation overheads to any desired constant fraction of the raw work with some increase in span, which we show to be small for a reasonably broad class of computations.

6.1 Ideal oracle

Theorem 6.1 quantifies the relationships between raw work/raw span and total work/total span for the three possible modes.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

Theorem 6.1 (*Work and span*)
Consider an expression e such that $e \Downarrow^{\alpha} v, (w, s), (\mathbb{W}, \mathbb{S})$. Assume $\phi = 0$. The following tight bounds can be obtained for total work and total span, on a machine with P processors where the cost of creating parallel threads is τ.

α	*Bound on total work*	*Bound on total span*
seq	$\mathbb{W} = w$	$\mathbb{S} = s = w$
par	$\mathbb{W} \leqslant (1 + \frac{\tau}{2})\, w$	$\mathbb{S} \leqslant (1 + \tau)\, s$
orc	$\mathbb{W} \leqslant (1 + \frac{\tau}{\kappa+1})\, w$	$\mathbb{S} \leqslant (1 + \max(\tau, \kappa))\, s$

Proof
Results for the sequential mode is straightforward by inspection of the semantics of the source language (Figure 3). The other results can be obtained by specializing the general bounds that we present later in this section (Theorems 6.2 and 6.3). In what follows, we give examples that attain the bounds for the parallel and the oracle modes.

• For the work in parallel mode, consider an expression consisting only of parallel tuples with n leaves, and thus $n-1$ "internal nodes". The raw work w is equal to $n+(n-1)$ while the total work $\mathbb{W}$ is equal to $n+(n-1)(1+\tau)$. The ratio $\mathbb{W}/w$ can be rewritten as $1 + \frac{(n-1)\tau}{2n-1}$, which tends to $1 + \frac{\tau}{2}$ as n grows.

• For the span in parallel mode, we can use the same example. Each parallel tuple accounts for 1 in the raw span but for $1+\tau$ in the total span. So, the total span can be as much as $1+\tau$ times greater than the raw span.

• For the work in oracle mode, consider an expression with n nested parallel tuples, where tuples are always nested in the right branch of their parent tuple. The tuples are built on top of expressions that involve κ units of work. In the oracle semantics, all the tuples are executed in parallel. The raw work w is equal to $n + (n+1)\kappa$, and the total work $\mathbb{W}$ is equal to $n(1+\tau) + (n+1)\kappa$. The ratio $\mathbb{W}/w$ is equal to $1 + \frac{n\tau}{n(\kappa+1)+\kappa}$, which tends to $1 + \frac{\tau}{\kappa+1}$ when n gets large.

• For the span in oracle mode, in the case $\tau \geqslant \kappa$, we use the same example as for the work. The raw span is $n+1$ and the total span is $n(1+\tau)+\kappa$. The ratio $\mathbb{S}/s$ is equal to $1 + \frac{n\tau+\kappa-1}{n+1}$, which approaches $1+\tau$ as n grows.

• For the span in oracle mode, in the case $\kappa \geqslant \tau$, we change the example slightly so that now the tuples are built on leaves that involve just less than κ units of work. In the oracle semantics, all the tuples thus get executed sequentially. In this case, the raw span is $n+\kappa$ and the total span is equal to the total work, which is $n+(n+1)\kappa$. The ratio $\mathbb{S}/s$ can be expressed as $1 + \frac{n\kappa}{n+\kappa}$, which approaches $1+\kappa$ as n grows. □

This theorem leads to some important conclusions. First, the theorem shows that thread-creation (scheduling) costs matter a great deal. In a parallel evaluation, the total work and total span can be as much as τ times larger than the raw span and raw work. This essentially implies that a parallel program can be significantly slower than its sequential counterpart. If τ is large compared to the number of processors, then even in the ideal setting, where the number of parallel processors

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

is small relative to τ, we may observe no speedups. In fact, it is not uncommon to hear anecdotal evidence of this kind of slowdown in modern computer systems.

Second, the theorem shows that evaluation of a program with an ideal oracle can require as much as $\frac{\kappa}{2}$ less work than in the parallel mode. This comes at a cost of increasing the span by a factor of up to $\frac{\kappa}{\tau}$. Increasing the span of a computation can hurt parallel execution time because many parallel schedulers rely on *parallel slackness*, i.e., the availability of large degree of parallelism to achieve optimal speedups, or $\frac{w}{P} \gg s$. Unless done carefully, increasing the span can dramatically reduce parallel slackness. In the common case, however, where there is large amounts of parallel slackness, we can safely increase span by a factor of $\frac{\kappa}{\tau}$ to reduce the thread-creation overheads. Concretely, if parallel slackness is high, then, in the oracle mode, we can select κ such that parallel slackness is preserved—total span remains small compared to $\frac{w}{P}$, because κs remains relatively small compared to $\frac{w}{P}$—and the total work is reduced approximately by a factor of $\frac{\kappa}{2}$.

6.2 Realistic oracles

The analysis that we present above makes two unrealistic assumptions about oracles: (1) that they can accurately predict the raw work for a thread, and (2) that the oracle can make predictions in zero time. Realizing a very accurate oracle in practice is difficult, because it requires determining *a priori* the execution time of a thread. We therefore generalize the analysis by considering an approximate or realistic oracle that can make errors up to a multiplicative factor μ when estimating raw work. For example, an oracle can approximate raw work up to a constant factor of $\mu = 3$, i.e., a thread with raw work w would be estimated to perform raw work between $\frac{w}{3}$ and $3w$. Additionally, we allow the oracle to take some fixed time, written ϕ, to provide its answer.

We show that even with a realistic oracle, we can reduce thread-creation overheads. We start with bounding the span; the result implies that the total span is no larger than $\mu\kappa$ times the raw span when κ is large compared to τ and ϕ.

Theorem 6.2 (*Span with realistic oracle*)

$$e \Downarrow^{\mathsf{orc}} v, (w, s), (\mathbb{W}, \mathbb{S}) \quad \Rightarrow \quad \mathbb{S} \leqslant (1 + \mathtt{max}\,(\tau, \mu\kappa) + \phi)\, s$$

Proof

Let ρ denote $1 + \mathtt{max}\,(\tau, \mu\kappa) + \phi$; we want to prove that $\mathbb{S} \leqslant \rho s$. The proof is by induction on the derivation $e \Downarrow^{\mathsf{orc}} v, (w, s), (\mathbb{W}, \mathbb{S})$.

- For a rule with zero premise, we have $\mathbb{S} = s = 1$. Because $\rho \geqslant 1$, it follows that $\mathbb{S} \leqslant \rho s$.
- For a rule with one premise, we know by induction hypothesis that $\mathbb{S} \leqslant \rho s$. Using again the fact that $\rho \geqslant 1$, we can deduce the inequality $\mathbb{S} + 1 \leqslant \rho(s + 1)$.
- For a rule with two premises that does not correspond to a parallel tuple, we can similarly establish the conclusion $\mathbb{S}_1 + \mathbb{S}_2 + 1 \leqslant \rho(s_1 + s_2 + 1)$ using the induction hypotheses $\mathbb{S}_1 \leqslant \rho s_1$ and $\mathbb{S}_2 \leqslant \rho s_2$.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

- Now, consider the case of a parallel tuple. First, assume that the two branches of this tuple are predicted to be large. In this case, the tuple is executed in parallel and the branches are executed in oracle mode. We exploit the induction hypotheses $\mathbb{S}_1 \leqslant \rho s_1$ and $\mathbb{S}_2 \leqslant \rho s_2$ to conclude as follows:

$$\begin{aligned}\mathbb{S} &= \mathtt{max}(\mathbb{S}_1, \mathbb{S}_2) + 1 + \tau + \phi \\ &\leqslant \mathtt{max}(\rho s_1, \rho s_2) + 1 + \mathtt{max}(\tau, \mu\kappa) + \phi \\ &\leqslant \mathtt{max}(\rho s_1, \rho s_2) + \rho \\ &\leqslant \rho(\mathtt{max}(s_1, s_2) + 1) \\ &\leqslant \rho s\end{aligned}$$

- Consider now the case where both branches are predicted to be small. In this case, the tuple is executed sequentially. Because the oracle predicts the branches to be smaller than κ, they must be actually smaller than $\mu\kappa$. So, we have $w_1 \leqslant \mu\kappa$ and $w_2 \leqslant \mu\kappa$. Moreover, both branches are executed according to the sequential mode, so we have $\mathbb{S}_1 = w_1$ and $\mathbb{S}_2 = w_2$. It follows that $\mathbb{S}_1 \leqslant \mu\kappa$ and $\mathbb{S}_2 \leqslant \mu\kappa$. Below, we also exploit the fact that $\mathtt{max}(s_1, s_2) \geqslant 1$, which comes from the fact that raw span is at least one unit. We conclude as follows:

$$\begin{aligned}\mathbb{S} &= \mathbb{S}_1 + \mathbb{S}_2 + 1 + \phi \\ &\leqslant \mu\kappa + \mu\kappa + 1 + \phi \\ &\leqslant (1 + \mu\kappa + \phi) * 2 \\ &\leqslant (1 + \mathtt{max}(\tau, \mu\kappa) + \phi) \cdot (\mathtt{max}(s_1, s_2) + 1) \\ &\leqslant \rho s\end{aligned}$$

- It remains to consider the case where one branch is predicted to be smaller than the cut-off while the other branch is predicted to be larger than the cutoff. In this case again, both branches are executed sequentially. Without loss of generality, assume that the second branch is predicted to be small. In this case, we have $w_2 \leqslant \mu\kappa$. This second branch is thus executed according to the sequential mode, so we have $\mathbb{S}_2 = s_2 = w_2$. It follows that $\mathbb{S}_2 \leqslant \mu\kappa$. For the first branch, which is executed according to the oracle mode, we can exploit the induction hypothesis $\mathbb{S}_1 \leqslant \rho s_1$. We conclude as follows:

$$\begin{aligned}\mathbb{S} &= \mathbb{S}_1 + \mathbb{S}_2 + 1 + \phi \\ &\leqslant \rho s_1 + \mu\kappa + 1 + \phi \\ &\leqslant \rho s_1 + (1 + \mathtt{max}(\tau, \mu\kappa) + \phi) \\ &\leqslant \rho(s_1 + 1) \\ &\leqslant \rho(\mathtt{max}(s_1, s_2) + 1) \\ &\leqslant \rho s\end{aligned}$$

□

This ends our analysis of the span. Now, let us focus on the work. The fact that every call to the oracle can induce a cost ϕ can lead the work to be multiplied by a factor in proportion with ϕ. For example, consider a program made of a complete tree made of $2n$ leaves, n parallel tuples on the last layer of the tree, and $n - 1$ sequential tuples in the upper layers of the tree. This program has raw work equal

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

to $(n-1)+n+2n$, and total work equal to $(n-1)+n\phi+2n$. Thus, the increase in work $\mathbb{W}/w$ is equal to $\frac{(3+\phi)n-1}{4n-1}$, which is close to $\frac{\phi}{4}$ when n and ϕ are not small. This example shows that a program executed according to the oracle semantics can slow down by as much as a factor $\frac{\phi}{4}$ compared with a purely sequential execution.

The problem with the above example is that the oracle is called infrequently—only at the leaves of the computation—preventing us from amortizing the cost of the oracle toward larger pieces of computations. Fortunately, most programs do not exhibit this pathological behavior, because parallel tuples are often performed close to the root of the computation, allowing us to detect smaller pieces of work early.

One way to prevent the oracle from being called on smaller pieces of work is to make sure that it is called at regular intervals. For proving a strong bound on the work, we will simply assume that the oracle is not called on small threads by restricting our attention to *balanced programs*. To this end, we define balanced programs as programs that call the oracle only on expressions that are no smaller than some constant γ off from the value $\frac{\kappa}{\mu}$, for some $\gamma \geqslant 1$. Note that we use $\frac{\kappa}{\mu}$ as a target and not κ so as to accommodate possible over-estimations in the estimations of raw work. The formal definition follows:

Definition 6.1 (*Balanced programs*)
For $\gamma \geqslant 1$, a program or expression e is γ-balanced if evaluating e in the oracle mode invokes the oracle only for sub-expressions whose raw work is no less than $\frac{\kappa}{\mu\gamma}$.

Note that if a program is γ-balanced and if $\gamma < \gamma'$, then this program is also γ'-balanced. We will later give a sufficient condition for proving that particular programs are balanced (Section 6.4). For balanced programs, we are able to bound the total work with respect to the raw work.

Theorem 6.3 (*Work with a realistic oracle*)
Let e be a γ-balanced program.

$$e \Downarrow^{\mathsf{orc}} v, (w, s), (\mathbb{W}, \mathbb{S}) \quad \Rightarrow \quad \mathbb{W} \leqslant \left(1 + \frac{\mu(\tau + \gamma\phi)}{\kappa + 1}\right) w$$

Proof
We establish the following slightly tighter inequality (tighter because $\gamma \geqslant 1$ and $\mu \geqslant 1$).

$$\mathbb{W} \leqslant \left(1 + \frac{\tau}{\kappa/\mu + 1} + \frac{\phi}{\kappa/(\mu\gamma) + 1}\right) w$$

Define κ' as a shorthand for κ/μ and κ'' as a shorthand for $\kappa/(\mu\gamma)$. Note that, because $\gamma \geqslant 1$, we have $\kappa'' \leqslant \kappa'$. Let x^+ be defined as the value x when x is non-negative and as zero otherwise. We establish the following inequality by induction:

$$\mathbb{W} \leqslant w + \tau \left\lfloor \frac{(w-\kappa')^+}{\kappa'+1} \right\rfloor + \phi \left\lfloor \frac{(w-\kappa'')^+}{\kappa''+1} \right\rfloor$$

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

This is indeed a strengthened result because we have

$$\tau \left\lfloor \frac{(w-\kappa')^+}{\kappa'+1} \right\rfloor \leqslant \tau \frac{w}{\kappa'+1} \leqslant \frac{\tau}{\kappa/\mu+1} w$$
$$\text{and} \quad \phi \left\lfloor \frac{(w-\kappa'')^+}{\kappa''+1} \right\rfloor \leqslant \phi \frac{w}{\kappa''+1} \leqslant \frac{\phi}{\kappa/(\mu\gamma)+1} w.$$

The proof is conducted by induction on the derivation of the reduction hypothesis.

- For a rule with zero premises, which describe an atomic operation, we have $\mathbb{W} = w = 1$, so the conclusion is satisfied.
- For a rule with a single premise, the induction hypothesis is

$$\mathbb{W} \leqslant w + \tau \left\lfloor \frac{(w-\kappa')^+}{\kappa'+1} \right\rfloor + \phi \left\lfloor \frac{(w-\kappa'')^+}{\kappa''+1} \right\rfloor.$$

  So, we can easily derive the conclusion:

$$\mathbb{W} + 1 \leqslant (w+1) + \tau \left\lfloor \frac{((w+1)-\kappa')^+}{\kappa'+1} \right\rfloor + \phi \left\lfloor \frac{((w+1)-\kappa'')^+}{\kappa''+1} \right\rfloor.$$

- For a rule with two premises, we exploit the mathematical inequality $\left\lfloor \frac{n}{q} \right\rfloor + \left\lfloor \frac{m}{q} \right\rfloor \leqslant \left\lfloor \frac{n+m}{q} \right\rfloor$. We have

$$\begin{aligned}
\mathbb{W} &= \mathbb{W}_1 + \mathbb{W}_2 + 1 \\
&\leqslant w_1 + \tau \left\lfloor \frac{(w_1-\kappa')^+}{\kappa'+1} \right\rfloor + \phi \left\lfloor \frac{(w_1-\kappa'')^+}{\kappa''+1} \right\rfloor \\
&\quad + w_2 + \tau \left\lfloor \frac{(w_2-\kappa')^+}{\kappa'+1} \right\rfloor + \phi \left\lfloor \frac{(w_2-\kappa'')^+}{\kappa''+1} \right\rfloor + 1 \\
&\leqslant w + \tau \left\lfloor \frac{(w_1-\kappa')^+ + (w_2-\kappa')^+}{\kappa'+1} \right\rfloor \\
&\quad + \phi \left\lfloor \frac{(w_1-\kappa'')^+ + (w_2-\kappa'')^+}{\kappa''+1} \right\rfloor.
\end{aligned}$$

  To conclude, we need to establish the following two mathematical inequalities.

$$\begin{aligned}
(w_1-\kappa')^+ + (w_2-\kappa')^+ &\leqslant ((w_1+w_2+1)-\kappa')^+, \\
(w_1-\kappa'')^+ + (w_2-\kappa'')^+ &\leqslant ((w_1+w_2+1)-\kappa'')^+.
\end{aligned}$$

  The two equalities can be proved in a similar way. Let us establish the first one. There are four cases to consider. First, if both w_1 and w_2 are less than κ', then the right-hand side is zero, so we are done. Second, if both w_1 and w_2 are greater than κ', then all the expressions are non-negative, and we are left to check the inequality $w_1 - \kappa' + w_2 - \kappa' \leqslant w_1 + w_2 + 1 - \kappa'$. Third, if w_1 is greater than κ' and w_2 is smaller than κ', then the inequality becomes $(w_1-\kappa')^+ \leqslant ((w_1-\kappa') + (w_2+1))^+$, which is clearly true. The case $w_1 \geqslant \kappa'$ and $w_2 < \kappa'$ is symmetrical. This concludes the proof.
- Consider now the case of a parallel tuple where both branches are predicted to involve more than κ units of work. This implies $w_1 \geqslant \kappa'$ and $w_2 \geqslant \kappa'$. In this case, a parallel thread is created. Note that, because $\kappa'' \leqslant \kappa'$, we also have $w_1 \geqslant \kappa''$ and $w_2 \geqslant \kappa''$. So, all the values involved in the following computations

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

are non-negative. Using the induction hypotheses, we have

$$\begin{aligned}
\mathbb{W} &= \mathbb{W}_1 + \mathbb{W}_2 + 1 + \tau + \phi \\
&\leqslant w_1 + \tau \left\lfloor \tfrac{w_1-\kappa'}{\kappa'+1} \right\rfloor + \phi \left\lfloor \tfrac{w_1-\kappa''}{\kappa''+1} \right\rfloor \\
&\quad + w_2 + \tau \left\lfloor \tfrac{w_2-\kappa'}{\kappa'+1} \right\rfloor + \phi \left\lfloor \tfrac{w_2-\kappa''}{\kappa''+1} \right\rfloor + 1 + \tau + \phi \\
&\leqslant (w_1 + w_2 + 1) + \tau\left(\left\lfloor \tfrac{w_1-\kappa'}{\kappa'+1} \right\rfloor + \left\lfloor \tfrac{w_2-\kappa'}{\kappa'+1} \right\rfloor + 1 \right) \\
&\quad + \phi\left(\left\lfloor \tfrac{w_1-\kappa''}{\kappa''+1} \right\rfloor + \left\lfloor \tfrac{w_2-\kappa''}{\kappa''+1} \right\rfloor + 1 \right) \\
&\leqslant w + \tau \left\lfloor \tfrac{(w_1-\kappa')+(w_2-\kappa')+(\kappa'+1)}{\kappa'+1} \right\rfloor \\
&\quad + \phi \left\lfloor \tfrac{(w_1-\kappa'')+(w_2-\kappa'')+(\kappa''+1)}{\kappa''+1} \right\rfloor \\
&\leqslant w + \tau \left\lfloor \tfrac{(w_1+w_2+1)-\kappa'}{\kappa'+1} \right\rfloor + \phi \left\lfloor \tfrac{(w_1+w_2+1)-\kappa''}{\kappa''+1} \right\rfloor \\
&\leqslant w + \tau \left\lfloor \tfrac{w-\kappa'}{\kappa'+1} \right\rfloor + \phi \left\lfloor \tfrac{w-\kappa''}{\kappa''+1} \right\rfloor .
\end{aligned}$$

- Assume now that the two branches are predicted to be less than the cut-off. This implies $w_1 \leqslant \kappa'$ and $w_2 \leqslant \kappa'$. Both these threads are executed sequentially, so $\mathbb{W}_1 = w_1$ and $\mathbb{W}_2 = w_2$. Since the program is γ-balanced, we have $w_1 \geqslant \kappa''$ and $w_2 \geqslant \kappa''$. Those inequalities ensure that we are able to pay for the cost of calling the oracle, that is, the cost ϕ. Indeed, since we have $w_1 + w_2 + 1 - \kappa'' \geqslant \kappa'' + 1$, we know that $\left\lfloor \frac{w_1+w_2+1-\kappa''}{\kappa''+1} \right\rfloor \geqslant 1$. Therefore,

$$\begin{aligned}
\mathbb{W} &= \mathbb{W}_1 + \mathbb{W}_2 + 1 + \phi \\
&\leqslant w_1 + w_2 + 1 + \phi \\
&\leqslant (w_1 + w_2 + 1) + \phi \left\lfloor \tfrac{w_1+w_2+1-\kappa''}{\kappa''+1} \right\rfloor \\
&\leqslant w + \tau \left\lfloor \tfrac{(w-\kappa')^+}{\kappa'+1} \right\rfloor + \phi \left\lfloor \tfrac{w-\kappa''}{\kappa''+1} \right\rfloor .
\end{aligned}$$

- It remains to consider the case where one branch is predicted to be bigger than the cut-off while the other is predicted to be smaller than the cut-off. For example, assume $w_1 \geqslant \kappa'$ and $w_2 \leqslant \kappa'$. The parallel tuple is thus executed as a sequential tuple. The first thread is executed in oracle mode, whereas the second thread is executed in the sequential mode. For the first thread, we can invoke the induction hypothesis $\mathbb{W}_1 \leqslant w_1 + \tau\left\lfloor \frac{w_1-\kappa'}{\kappa'+1} \right\rfloor + \phi\left\lfloor \frac{w_1-\kappa''}{\kappa''+1} \right\rfloor$. For the second thread, which is executed sequentially, we have $\mathbb{W}_2 = w_2$. Moreover, since the oracle is invoked to predict the size of this second thread, we know by the hypothesis of γ-balance that $w_2 \geqslant \kappa/(\mu\gamma) = \kappa''$. Hence, we have $\left\lfloor \frac{w_2+1}{\kappa''+1} \right\rfloor \geqslant 1$. We conclude

$$\begin{aligned}
\mathbb{W} &= \mathbb{W}_1 + \mathbb{W}_2 + 1 + \phi \\
&\leqslant w_1 + \tau \left\lfloor \tfrac{w_1-\kappa'}{\kappa'+1} \right\rfloor + \phi \left\lfloor \tfrac{w_1-\kappa''}{\kappa''+1} \right\rfloor + w_2 + 1 + \phi \\
&\leqslant w_1 + \tau \left\lfloor \tfrac{w_1-\kappa'}{\kappa'+1} \right\rfloor + \phi \left\lfloor \tfrac{w_1-\kappa''}{\kappa''+1} \right\rfloor + w_2 + 1 + \phi \left\lfloor \tfrac{w_2+1}{\kappa'+1} \right\rfloor \\
&\leqslant w + \tau \left\lfloor \tfrac{w_1+w_2+1-\kappa'}{\kappa'+1} \right\rfloor + \phi \left\lfloor \tfrac{w_1+w_2+1-\kappa''}{\kappa''+1} \right\rfloor \\
&\leqslant w + \tau \left\lfloor \tfrac{w-\kappa'}{\kappa'+1} \right\rfloor + \phi \left\lfloor \tfrac{w-\kappa''}{\kappa''+1} \right\rfloor .
\end{aligned}$$

□

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

We are now ready to combine the version of Brent's theorem adapted to our cost semantics with the bounds that we have established for the total work and span in γ-balanced parallel programs executed under the oracle semantics.

Theorem 6.4 (*Execution time with a realistic oracle*)
Assume an oracle that costs ϕ and makes an error by a factor not exceeding μ. Assume $\kappa > \tau$, which is always the case in practice. The execution time of a parallel γ-balanced program on a machine with P processors under the oracle semantics with a greedy scheduler does not exceed the value:

$$\left(1 + \frac{\mu(\tau + \gamma\phi)}{\kappa}\right) \frac{w}{P} + (\kappa\mu + \phi + 1)\, s.$$

Proof
The bound follows by our generalized version of Brent's theorem (Theorem 5.1), and by the bounds established in Theorems 6.3 and 6.2. For simplicity, we replace the denominator $\kappa + 1$ with κ. This change does not loosen the bound significantly because κ is usually large compared to the unit cost. Also for simplicity, we have replaced the term $\max(\tau, \mu\kappa)$ with $\mu\kappa$, exploiting the assumption $\kappa > \tau$ and the fact that $\mu \geqslant 1$. □

6.3 Choice of the cut-off

Theorem 6.4 shows that the running time of a parallel program can be controlled by changing the constant κ. The formula reveals an interesting trade-off: We can reduce task-creation overheads but this comes at the cost of increasing the span. To see this connection better, consider the bound that appears in the statement of Theorem 6.4 and notice that as the value of κ increases, the work (first) term decreases but the span (second) term increases. The parallel run time thus decreases as we increase κ up to some inflection point and then starts increasing. We compute the optimal value for κ by solving for the root of the derivative. We obtain

$$\kappa^* = \sqrt{\tau + \gamma\phi} \cdot \sqrt{\frac{w}{Ps}}.$$

Thus, with prior knowledge of the raw work and raw span of a computation, we can pick κ to ensure efficiency of parallel programs.

Such knowledge, however, is often unavailable. As we now show, we can improve efficiency of parallel programs by selecting a fixed κ that guarantees that the task creation overheads can be bounded by any constant fraction of the raw work, without increasing the span of the computation significantly.

Theorem 6.5 (*Run time with fixed* κ)
Consider an oracle with ϕ cost and μ error. For any $\gamma \geqslant 1$ and for any constant r such that $0 < r < 1$, there exists a constant κ and a constant c such that the evaluation with the oracle semantics of a γ-balanced program reduces task creation overheads to a fraction r of the raw work, while in the same time increasing the total span by no more than a factor $\frac{c}{r}$. With a greedy scheduler, the total parallel run time on P processors of such a program thus does not exceed $(1 + r)\frac{w}{P} + \frac{c}{r}\, s$.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

Proof
Consider a particular γ-balanced program with raw work w and raw span s, and consider its evaluation under the oracle semantics. By Theorem 6.3, we know that total work does not exceed $(1+\frac{\mu(\tau+\gamma\phi)}{\kappa})\,w$. To achieve the desired bound on execution time, we take $\kappa = \frac{\mu(\tau+\gamma\phi)}{r}$. Plugging this value of κ into the formula yields $(1+r)\,w$ for total work, showing that task creation overheads are reduced to a fraction r of the raw work.

Furthermore, by Theorem 6.2, we know that the total span is bounded by $(\mathtt{max}\,(\tau, \mu\kappa) + \phi + 1)\,s$. Plugging in the same value for κ yields the following bound on total span:

$$\mathbb{S} \leqslant \left(\mathtt{max}\left(\tau, \frac{\mu^2(\tau+\gamma\phi)}{r}\right) + \phi + 1\right) s.$$

Using $\mu \geqslant 1$ and $r < 1$, we can derive the inequality

$$\mathbb{S} \leqslant \left(\frac{\mu^2(\tau+\gamma\phi)}{r} + \frac{\phi+1}{r}\right) s.$$

Choosing $c = \mu^2(\tau+\gamma\phi) + \phi + 1$ therefore ensures that the total span does not exceed the desired bound $\frac{c}{r}\,s$. The run-time bound follows by an application of our generalized version of Brent's theorem (Theorem 5.1). □

This final theorem enables us to reduce task creation overheads to any desired constant fraction of the raw work by choosing a κ that is independent of the specific inputs. This comes at the cost of increasing the span, but only by a constant factor of $\frac{c}{r}$. In the common case, when the work is asymptotically greater than span, e.g., $\Theta(n)$ versus $O(\log n)$, the resulting run-time guarantees that the increase in span remain small: specifically, the span term itself is a fraction of the work term for all but a constant number of small inputs.

6.4 Balanced programs

Our bounds with the realistic oracle hold only for what we called γ-balanced programs, where the oracle is not called on small threads. This assumption can be satisfied by calling the oracle "regularly". It seems likely that this assumption would hold for many programs without requiring any changes to the program code. In this section, we show that recursive, divide-and-conquer programs are γ-balanced.

To that end, we introduce the notion of γ-regularity. Intuitively, a program is γ-regular if, between any two calls to the oracle involved in the execution of this program, the amount of work does not reduce by more than a factor γ. We will then establish that any γ-regular program is a γ-balanced program. Before giving the formal definition of γ-regularity, we need to formally define what it means for a parallel tuple to be dominated by another parallel tuple.

Definition 6.2 (*Domination of a parallel branch*)
A branch e of a parallel tuple is said to be *dominated* by the branch e_i of another parallel tuple $\left(|e_1, e_2|\right)$ if the expression e is involved in the execution of the branch e_i.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

Definition 6.3 (*Regularity of a parallel program*)
A program is said to be γ-regular if, for any parallel branch involving, say, w units of raw work, either w is greater than $\kappa/(\mu\gamma)$ or this branch is dominated by another parallel branch that involves less than γw units of work.

Remark: The condition "w is greater than $\kappa/(\mu\gamma)$" is typically used to handle the outermost parallel tuples, which are not dominated by any other tuple.

Note that the regularity of a program is always greater than 2. Indeed, if one of the branch of a parallel tuple is more than half of the size of the entire tuple, then the other branch must be smaller than half of that size. On the one hand, algorithms that divide their work in equal parts are γ-regularity with γ very close to 2. On the other hand, ill-balanced programs can have a very high degree of regularity. Observe that every program is ∞-regular.

For example, consider a program that traverses a complete binary tree in linear time. A call on a tree of size n has raw work nc, for some constant c. If the tree is not a leaf, its size n has to be at least 3. The next recursive call involves raw work $\left\lfloor \frac{n-1}{2} \right\rfloor c$, The ratio between those two values is equal $n/\left\lfloor \frac{n-1}{2} \right\rfloor$. This value is always less than 3 when $n \geqslant 3$. So, the traversal of a complete binary tree is a 3-regular algorithm.

The following lemma explains how the regularity assumption can be exploited to ensure that the oracle is never invoked on threads of size less than $\kappa/(\mu\gamma)$. This suggests that, for the purpose of amortizing well the costs of the oracle, a smaller regularity is better.

Lemma 6.1 (*From regularity to balanced*)
If a program is γ-regular, then it is γ-balanced.

Proof
We have to show that, during the execution of a γ-regular program according to oracle semantics, the oracle is never invoked on sub-expressions involving less than $\kappa/(\mu\gamma)$ raw work. Consider a particular sub-expression e involving w units of raw work, and assume that the oracle is invoked on this sub-expression. Because the oracle is being invoked, e must correspond to the branch of a parallel tuple. By the regularity assumption, either w is no less than $\kappa/(\mu\gamma)$, in which case the conclusion holds immediately, or the branch e is dominated by a branch e_i that involves that involves w' units of work, with $w' \leqslant \gamma w$. For the latter case, we need to establish $w \geqslant \kappa/(\mu\gamma)$. To that end, it suffices to prove that $w' \geqslant \kappa/\mu$, which amounts to showing that the amount of raw work associated with the dominating branch e_i contains at least κ/μ raw work.

We conclude the proof by establishing the inequality $w' \geqslant \kappa/\mu$. Because the oracle is being invoked on the sub-expression e, it means that e is being evaluated in the mode orc. Therefore, the call to the oracle on the dominating branch e_i must have predicted e_i to contain more than κ raw work. (Otherwise, e_i and its sub-expression e would have both been executed in the sequential mode.) Given that the oracle makes error by no more than a factor μ, if e_i is predicted to contain more than κ units of raw work, then e_i must contain at least κ/μ units of raw work. So, $w' \geqslant \kappa/\mu$. □

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

```
signature Estimator =
sig
  type cost
  type estimator
  val create: unit → estimator
  val report: estimator × cost × float → unit
  val predict: estimator × cost → float
end
```

Fig. 4. The interface for the estimator data structure.

7 Approximating the oracle-guided semantics

In the previous section, we have established that, if we have access to an oracle that can estimate actual raw work, i.e., sequential run-time, of an expression within a factor of no more than μ and at a cost of no more than a constant ϕ, then we can effectively control the thread-creation overheads and thus enable efficient parallel execution. In this section, we describe how to approximate such an oracle by combining certain information from the programmer along with run-time measurements. We first describe our approximation algorithm and then, in Section 8, show how some of the information needed by the algorithm can be derived via a program transformation.

The basic idea behind our approximation algorithm is to use some crucial information provided by the programmer to estimate the actual raw work of a computation. More precisely, we require the user to provide a *cost function* for each function in the program. When applied to an argument `v`, a cost function of `f` returns an *abstract cost* of the raw-work of application of `f` to `v`. A crucial property of the abstract costs is that they should be abstract enough that the programmer can write the cost functions without necessarily knowing the details of the hardware where the programs will be executed on. Yet, abstract costs should provide sufficient information to estimate the actual run times.

Asymptotic complexity specifications serve as a natural cost function by satisfying both of these properties. Since they eliminate hardware specific constants, they can be specified easily. Using complexity functions, we can approximate the actual run time of sequentially executed functions by simply determining the constants hidden by the asymptotic complexity notation. Such an approximation can be performed, e.g., by using the least squares method or similar techniques for data fitting from known samples. To perform this approximation, we use an *estimator* data structure, that, given abstract cost samples for a function, can estimate the raw work for that function on a given argument.

Figure 4 shows the interface for the estimator. To perform accurate estimates, the estimator utilizes profiling data obtained from actual execution times. The sampling operation `report(t, c, d)` adds a cost `c` and an execution time `d` to the set of samples in an estimator `t`. An estimate of the actual execution time is obtained by calling `predict`. Given an estimator `t` and cost `c`, the call `predict(t, c)` returns a predicted execution time.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

Since the abstract cost simply a measure of the asymptotic work of a function, all that remains for the estimator data structure is to calculate at run-time the actual constant factors for the hardware, where execution takes place. In our implementation, we represent a value of type `cost` as an integer that represents the application of the complexity function applied to the input size. We approximate the actual run time by calculating a single constant, assuming that the constants in all terms of the asymptotic complexity are the same. Although assuming a single constant can decrease the precision of the approximations, we believe that it suffices because we only have to compute lower bounds for our functions; i.e., we only need to determine whether they are "big enough" for parallel execution. We note that, for our theoretical bounds to apply, complexity expressions should require constant time to evaluate, which is usually the case because, in the common case, the cost functions are relatively simple functions.

To ensure lightweight execution, our implementation simply computes the constant factor for each reported cost, and averages such constants over a period of time, which it then uses for prediction. The constants calculated for each reported cost may evolve over time. For example, if the current program is sharing the machine with another program, a series of memory reads by the other program may slow down the current program. For this reason, we do not just compute the average across the entire history, but instead maintain a moving average, that is, an average of the values gathered across a certain number of runs.

Computing averages is not entirely straightforward. On the one hand, storing the average in a memory cell that is shared by all processors is not satisfying, because it would involve some synchronization problems. On the other hand, using a different memory cell for every processor is not satisfying either, because it leads to slower updates of the constants when they change. In particular, in the beginning of the execution of a program, it is important that all processors quickly share a relatively good estimate of the constant factors.

For these reasons, we have opted for an approach that uses not only a shared memory cell but also one data structure local to every processor. The shared memory cell associated with each estimator contains the estimated value for the constant that is read by all the processors when they need to predict execution times. The local data structures are used to accumulate statistics on the value of the constant. Those statistics are reported on a regular basis to the shared memory cell, by computing a weighted mean between the value previously stored in the shared memory cell and the value obtained out of the local data structure. We treat initializations somewhat specially: For the first few measures, a processor always begins by reporting its current average to the shared memory cell. This policy ensures a fast propagation of the information gathered from the first runs, so as to quickly improve the accuracy of the predictions.

In the rest of this section, we present a more detailed description of our implementation of estimators. An estimator is represented as (1) a shared floating-point value storing the global estimated value, (2) an integer value storing, in the early phases, the number of times that the shared value has been updated, and (3) a processor-indexed array of pairs, each made of an integer and a floating-point value,

for storing the number of measures, and the sum of the constant factors measured. We define the *local average value* associated with a given processor as the ratio between the sum of the measured constants and the number of those measures.

The function `create` allocates a fresh estimator. It sets all fields to zero, except the global constant which is set to a very pessimistic constant, e.g., 1 microsecond. As a result, in the first few calls, the estimator largely over-approximates the work, leading only very small pieces of computation to be executed sequentially.

The function `predict` simply computes the product of the global estimated value of the constant with the integer cost produced by the user-provided asymptotic cost function.

The function `report` refines the estimation of the constant factor. First, it updates the pair of values associated with the processor that performed the measure, by incrementing the number of measures and adding to the sum field the ratio between the execution time and the complexity value being reported. Second, it decides whether the local average value of the constant should be reported to the global value. It does so if either (a) the number of measures exceeds $c * P$, for some real constant c, or if (b) the number of times that the shared value has been updated is less than a fixed bound. The latter condition helps for fast propagation of the constant in the early phases.

When the function `report` decides to report its local average value to the global value, it writes into the shared value field a weighted average of the local average value and of the previous global value. For safety (that is, to avoid potentially sequentializing too large pieces of computations), we limit a single change to the global value to be of at most one order of magnitude (e.g., a factor 10). Besides, as an optimization, we skip the write operation if we notice that the new value is not significantly different from the old (e.g., when the value would change by less than 20%). After updating the shared value, the function resets to zero the pair of values associated with the current processor, in order to begin a new round of measurements. Moreover, in case the update of the shared value has been triggered by the condition (b) described above, the function also increments the shared field that stores the number of updates to the global value.

Note that the write operation that updates the shared value may sometimes get discarded as a result of a data race between processors. Such races, however, are benign.

When implementing the oracle, we faced three technical difficulties. First, we had to pay attention to the fact that the memory cells allocated for the different processors are not allocated next to each other. Otherwise, those cells would fall in the same cache line, in which case writing in one of these cells would make the other cells be removed from caches, making subsequent reads more costly. Second, we observed that the time measures typically yield a few outliers. Those are typically due to the activity of the garbage collector or of another program being scheduled by the operating system on the same processor. Fortunately, we have found detecting these outliers to be relatively easy because the measured times are at least one or two orders of magnitude greater than the cut-off value. Third, the default system function that reports the time is only accurate by 1 microsecond. This is good enough

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

when the cut-off is greater than 10 microseconds. However, if one were to aim for a smaller cut-off, which could be useful for programs exhibiting only a limited amount of parallelism, then more accurate techniques would be required, for example, using the specific processor instructions for counting the number of processor cycles.

8 Implementation via source-to-source translation

We describe how to compile parallel codes with complexity annotations into codes that implement our oracle-guided semantics. Our compilation technique performs a source-to-source translation and relies on the technique described in Section 7. The idea is to compile every piece of code in two versions: one for the sequential semantics (`seq` mode), where all parallel pairs are simply erased to sequential pairs, and one for the oracle-guided semantics (`orc` mode), where parallel pairs are instrumented in such a way as to perform predictions and possibly measures, and to decide whether to execute the branches sequentially or in parallel.

For simplicity, we assume that constituents of parallel tuples are function applications, i.e., they are of the form $(|f_1\, v_1, f_2\, v_2|)$. Note that this assumption does not cause loss of expressiveness, because a term e can always be replaced by a trivial application of a "thunk", a function that ignores its argument (typically of type "unit") and evaluates e to a dummy argument. Throughout, we write "`fun` $f.x.e_b$ $[e_c]$" to denote a function "`fun` $f.x.e_b$" for which the cost function for the body e_b is described by the expression e_c. This expression e_c, which may refer to the argument x, should be an expression whose evaluation always terminates and produces a cost of type `cost`.

To associate an estimator with each function, in a simple pass over the source code, we allocate and initialize an estimator for each syntactic function definition. For example, if the source code contains a function of the form "`fun` $f.x.e_b$ $[e_c]$", then our compiler allocates an estimator specific to that function definition. Specifically, if the variable r refers to the allocated estimator, then the translated function, written "`fun` $f.x.e_b$ $[e_c|r]$", is annotated with r.

The second pass of our compilation scheme uses the allocated estimators to approximate the actual raw work of function applications and relies on an `MakeBranch` function to determine whether an application should be run in the oracle or in the sequential mode. Figure 5 defines more precisely the second pass. We write $[\![v]\!]$ for the translation of a value v, and we write $[\![e]\!]^{\alpha}$ for the translation of the expression e according to the semantics α, which can be either `seq` or `orc`. When specifying the translation, we use triples, quadruples, projections, sequence, if-then-else statements, and unit value; these constructions can all be easily defined in our core programming language.

Translation of values other than functions does not depend on the mode and is relatively straightforward. We translate functions, which are of the form "`fun` $f.x.e_b$ $[e_c|r]$", into a quadruple consisting of the estimator `r`, a sequential cost function, the sequential version of the function, and the oracle versions of the function. Translation of a function application depends on the mode. In the sequential mode, the sequential version of the function is selected (by projecting the

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

$$
\begin{array}{lcl}
[\![x]\!] & \equiv & x \\
[\![(v_1, v_2)]\!] & \equiv & ([\![v_1]\!], [\![v_2]\!]) \\
[\![\texttt{inl}\ v]\!] & \equiv & \texttt{inl}\ [\![v]\!] \\
[\![\texttt{inr}\ v]\!] & \equiv & \texttt{inr}\ [\![v]\!] \\
[\![\texttt{fun}\ f.x.e_b\ [e_c|r]]\!] & \equiv & (r, (\texttt{fun}\ \_.x.[\![e_c]\!]^{\mathsf{seq}}), (\texttt{fun}\ f.x.[\![e_b]\!]^{\mathsf{seq}}), (\texttt{fun}\ f.x.[\![e_b]\!]^{\mathsf{orc}})) \\
[\![v]\!]^{\alpha} & \equiv & [\![v]\!] \\
[\![v_1\ v_2]\!]^{\mathsf{seq}} & \equiv & \texttt{proj}^3\ [\![v_1]\!]\ [\![v_2]\!] \\
[\![v_1\ v_2]\!]^{\mathsf{orc}} & \equiv & \texttt{proj}^4\ [\![v_1]\!]\ [\![v_2]\!] \\
[\![(e_1, e_2)]\!]^{\alpha} & \equiv & ([\![e_1]\!]^{\alpha}, [\![e_2]\!]^{\alpha}) \\
[\![\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2]\!]^{\alpha} & \equiv & \texttt{let}\ x\ =\ [\![e_1]\!]^{\alpha}\ \texttt{in}\ [\![e_2]\!]^{\alpha} \\
[\![\texttt{fst}\ v]\!]^{\alpha} & \equiv & \texttt{fst}\ [\![v]\!] \\
[\![\texttt{snd}\ v]\!]^{\alpha} & \equiv & \texttt{snd}\ [\![v]\!] \\
[\![\texttt{case}\ v\ \texttt{of}\ \{\texttt{inl}\ x.e_1, \texttt{inr}\ x.e_2\}]\!]^{\alpha} & \equiv & \texttt{case}\ [\![v]\!]\ \texttt{of}\ \{\texttt{inl}\ x.[\![e_1]\!]^{\alpha}, \texttt{inr}\ x.[\![e_2]\!]^{\alpha}\} \\
[\![(|f_1\ v_1, f_2\ v_2|)]\!]^{\mathsf{seq}} & \equiv & (\texttt{proj}^3\ [\![f_1]\!]\ [\![v_1]\!], \texttt{proj}^3\ [\![f_2]\!]\ [\![v_2]\!]) \\
[\![(|f_1\ v_1, f_2\ v_2|)]\!]^{\mathsf{orc}} & \equiv & \left\{\begin{array}{l}\texttt{let}\ (b_1, k_1)\ =\ \texttt{MakeBranch}([\![f_1]\!], [\![v_1]\!])\ \texttt{in} \\ \texttt{let}\ (b_2, k_2)\ =\ \texttt{MakeBranch}([\![f_2]\!], [\![v_2]\!])\ \texttt{in} \\ \texttt{if}\ (b_1\ \&\&\ b_2)\ \texttt{then}\ (|k_1\ (), k_2\ ()|)\ \texttt{else}\ (k_1\ (), k_2\ ())\end{array}\right.
\end{array}
$$

Fig. 5. Translation for oracle scheduling.

third component of the function) and used in the application. Similarly, in the oracle mode, the oracle version of the function is selected and used in the application. To translate a tuple, we recursively translate the sub-expression, while preserving the mode. Similarly, translation of the `let`, projections, and `case` constructs are entirely structural.

In the sequential mode, a parallel tuple is turned into a simple tuple. In the oracle mode, the translation applies the oracle-based scheduling policy with the aid of the meta-function `MakeBranch`. This meta-function, shown in Figure 6, describes the template of the code generated for preparing the execution of a parallel tuple. `MakeBranch` expects a (translated) function f and its (translated) argument v, and it returns a boolean b indicating whether the application of f to v is expected to take more or less time than the cut-off κ, and a thunk t to execute this application. On the one hand, if the application is predicted to take more time than the cut-off (in which case b is true), then the thunk t corresponds to the application of the oracle-semantics version of the function f. On the other hand, if the application is predicted to take less time than the cut-off (in which case b is false), then the thunk t corresponds to the application of the sequential-semantics version of the function f. Moreover, in the latter case, the time taken to execute the application sequentially is measured. This time measure is reported to the estimator by the auxiliary meta-function `MeasuredRun` (Figure 6), so as to enable its approximations.

Observe that the translation introduces many quadruples and applications of projection functions. However, in practice, the quadruples typically get inlined so most of the projections can be computed at compile time. Observe also that the compilation scheme involves some code duplication, because every function is translated once for the sequential mode and once for the oracle mode. In theory, the code could grow exponentially when the code involves functions defined inside

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

```
MakeBranch(f, v) ≡
    let r = proj^1 f in
    let m = proj^2 f v in
    let b = predict(r, m) > κ in
    let fun k_seq () = proj^3 f v in
    let fun k'_seq () = MeasuredRun(r, m, k_seq) in
    let fun k_orc () = proj^4 f v in
    let k = if b then k_orc else k'_seq in
    (b, k)
MeasuredRun(r, m, k) ≡
    let t = get_time () in
    let v = k () in
    let t' = get_time () in
    report (r, m, (t' − t));
    v
```

Fig. 6. Auxiliary meta-functions used for compilation.

the body of other functions. In practice, the code the growth is limited because functions are rarely deeply nested. If code duplication was a problem, then we can use flattening to eliminate deep nesting of local functions, or pass the mode α as an extra argument to functions.

9 Empirical evaluation

In this section, we evaluate the effectiveness of our implementation through several experiments. We consider results from a range of benchmarks run on a machine with 16 processors. The results show that, in each case, our oracle implementation improves on the plain work-stealing implementation. Furthermore, the results show that the oracle implementation scales well with up to 16 processors.

9.1 Implementation in Manticore

In this section, we describe the implementation of our scheduling technique in an actual language and system. In our approach, source programs are written in our own dialect of the Caml language (Leroy *et al.*, 2005), which is a strict functional language. Our Caml dialect corresponds to the core Caml language extended with syntax for parallel pairs and complexity annotations.

We use the Caml type checker to obtain a typed syntax tree, on which we perform the oracle-scheduling translation defined in Figure 5. We then produce code in the syntax of PML (Fluet *et al.*, 2011), a parallel language close to Standard ML. The translation from Caml to PML is straightforward because the two languages are relatively similar. We compile our source programs to x86-64 binaries using Manticore, which is the optimizing PML compiler.

The original theorem of Brent as well as our generalization both assume a greedy-scheduler that can find available work (threads to execute) immediately with no overhead. This assumption is unrealistic of course in a literal sense, but work-

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

stealing schedulers are able to control the overheads for balancing work remarkably well, both in theory (Blumofe and Leiserson, 1999; Arora *et al*., 2001; Acar *et al*., 2013) and in practice (Frigo *et al*., 1998). On the one hand, overheads related to queueing threads are tiny, and, on the other hand, the delay due to steals is tamed by the fact that steals are relatively rare. Work stealing has also been shown to have good locality behavior, which further improves its effectiveness (Acar *et al*., 2002).

In Manticore's work-stealing scheduler, all system processors are assigned to collaborate on the computation. Each processor owns a deque (doubly ended queue) of threads represented as thunks. Processors treat their own deques like call stacks. When a processor starts to evaluate a parallel-pair expression, it creates a thread for the second sub-expression of the pair and pushes the thread onto the bottom of the deque. Processors that have no work left try to *steal* threads from others. More precisely, they repeatedly select a random processor and try to pop a thread from this processor's deque.

Manticore's implementation of work stealing (Rainey, 2010) adopts a code-specialization scheme, called clone translation, taken from Cilk-5's implementation (Frigo *et al*., 1998).[2] With clone translation, each parallel-pair expression is compiled into two versions: the fast clone and the slow clone. The purpose of a fast clone is to optimize the code that corresponds to evaluating on the local processor, whereas the slow clone is used when the second branch of a parallel-pair is migrated to another processor. A common aspect between clone translation and our oracle translation (Figure 5) is that both generate specialized code for the sequential case. But the clone translation differs in that there is no point at which parallelism is cut-off entirely, as the fast clone may spawn threads.

The scheduling cost involved in the fast clone is a (small) constant, because it involves just a few local operations, but the scheduling cost of the slow clone is variable, because it involves inter-processor communication. It is well established, both through analysis and experimentation, that (with high probability) no more than $O(P\,\mathbb{S})$ steals occur during the evaluation (Arora *et al*., 2001). So, for programs that exhibit parallel slackness ($\mathbb{W} \gg P\,\mathbb{S}$), we do not need to take into account the cost of slow clones because there are relatively few of them. We focus only on the cost of creating fast clones, which correspond to the cost τ. A fast clone needs to package a thread, push it onto the deque and later pop it from the deque. So, a fast clone is not quite as fast as the corresponding sequential code. The exact slowdown depend on the implementation, but in our case we have observed that a fast clone is three to five times slower than a simple function call.

9.2 Test machine

Our test machine has four quad-core AMD Opteron 8380 processors running at 2.5 GHz. Each core has 64 Kb each of L1 instruction and data cache, and a 512 Kb L2 cache. Each processor has a 6 Mb L3 cache that is shared with the four cores of the

[2] In the Cilk-5 implementation, it is called clone compilation.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

processor. The system has 32 Gb of RAM and runs Debian Linux (kernel version 2.6.31.6-amd64).

9.3 Measuring scheduling costs

We report estimates of the thread-creation overheads for each of our test machine. To estimate, we use a synthetic benchmark expression e whose evaluation sums integers between 0 and 30 million using a parallel divide-and-conquer computation. We chose this particular expression because most of its evaluation time is spent evaluating parallel pairs.

First, we measure w_s: the time required for executing a sequentialized version of the program (a copy of the program where parallel tuples are systematically replaced with sequential tuples). This measure serves as the baseline. Second, we measure w_w: the time required for executing the program using work stealing, on a single processor. This measure is used to evaluate τ. Third, we measure w_o: the time required for executing a version of the program with parallel tuples replaced with ordinary tuples but where we still call the oracle. This measure is used to evaluate ϕ.

We then define the work-stealing overhead $c_w = \frac{w_w}{w_s}$. We estimate the cost τ of creating a parallel thread in work stealing by computing $\frac{w_w - w_s}{n}$, where n is the number of parallel pairs evaluated in the program. We also estimate the cost ϕ of invoking the oracle by computing $\frac{w_o - w_s}{m}$, where m is the number of times the oracle is invoked.

Our measures are as follows: $c_w = 4.86$ and $\tau = 0.09$ microseconds and $\phi = 0.18$ microseconds. The first value indicates that work stealing alone can induce a slowdown by a factor of 4 or 5, for programs that create a huge number of parallel tuples. The value of τ, close to one-tenth of a microsecond, indicates, that the cost of creating parallel threads is roughly between 200 and 300 processor cycles (since our benchmark machine runs at 2.5 GHz). This cost is quite significant in front of basic operations that execute in just a few cycles.

The oracle cost ϕ is even larger than τ—about twice larger. The fact that the oracle is associated with significant costs is not surprising: Recall that the estimations and measures associated with our implementation of the oracle involve functions calls and memory operations. However, also keep in mind that our investment in performing oracle predictions is supposed to pay off as soon as we are able to sequentialize large pieces of computation. Indeed, as we have established, the cost of the oracle can be expected to be well amortized and to only account for a tiny fraction of the total execution time.

In our experiments, we used the cut-off value $\kappa = 26$ microseconds. We use Theorem 6.4 to estimate an upper-bound on the overheads associated with thread creation and oracle predictions. To that end, we first need to evaluate μ and γ. We can estimate μ, the bound on the relative error associated with the predictions, by performing runs where we logged both our predictions and our measures. We observed that, apart from a few outliers (i.e. exceptionally large measures for sequentialized threads), our oracle is always accurate within a factor 2. So, we assume $\mu = 2$. Besides, our benchmark programs are fairly regular: We can assume $\gamma = 3$ for all of them.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

Now, Theorem 6.4 tells us that, for programs exhibiting sufficient parallelism, the overheads are essentially bounded by $\frac{\mu(\tau+\gamma\phi)}{\kappa}$. In our setting, this ratio evaluates to just below 5%—we actually set the value of κ to meet this target. In practice, since the errors associated with our time predictions tend to balance out, the μ term that appears in front of the bound does not impact us as much as it could in theory. Likewise, the irregularity factor ϕ does not actually reach 3 for every parallel tuple executed. For these reasons, we believe that, in practice, overheads usually do not exceed 3%.

9.4 Benchmarks

We used five benchmarks in our empirical evaluation. Each benchmark program was originally written by other researchers and ported to our dialect of Caml.

The Quicksort benchmark sorts a sequence of 2 million integers. Our program is adapted from a functional, tree-based algorithm (Blelloch and Greiner, 1995). The algorithm runs with $O(n \log n)$ raw work and $O(\log^2 n)$ raw depth, where n is the length of the sequence. Sequences of integers are represented as binary trees in which sequence elements are stored at leaf nodes and each internal node caches the number of leaves contained in its subtree.

The Quickhull benchmark calculates the convex hull of a sequence of 3 million points contained in 2-d space. The algorithm runs with $O(n \log n)$ raw work and $O(\log^2 n)$ raw depth, where n is the length of the sequence. The representation of points is similar to that of Quicksort, except that leaves store 2-d points instead of integers.

The Barnes–Hut benchmark is an n-body simulation that calculates the gravitational forces between n particles as they move through 2-d space (Barnes and Hut, 1986). The Barnes–Hut computation consists of two phases. In the first, the simulation volume is divided into square cells via a quadtree, so that only particles from nearby cells need to be handled individually and particles from distant cells can be grouped together and treated as large particles. The second phase calculates gravitational forces using the quadtree to accelerate the computation. The algorithm runs with $O(n \log n)$ raw work and $O(\log n)$ raw depth. Our benchmark runs 10 iterations over 100,000 particles generated from a random Plummer distribution (Plummer, 1911). The program is adapted from a Data-Parallel Haskell program (Peyton Jones, 2008). The representation we use for sequences of particles is similar to that of Quicksort.

The SMVM benchmark multiplies an $m \times n$ matrix with an $n \times 1$ dense vector. Our sparse matrix is stored in the compressed sparse-row format. The program contains parallelism both between dot products and within individual dot products. We use a sparse matrix of dimension $m = 500{,}000$ and $n = 448{,}000$, containing 50,400,000 non-zero values.

The DMM benchmark multiplies two dense, square $n \times n$ matrices using the recursive divide-and-conquer algorithm of Frens and Wise (1997). We have recursion go down to scalar elements. The algorithm runs with $O(n^3)$ raw work and $O(\log n)$ raw depth. We selected $n = 512$.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

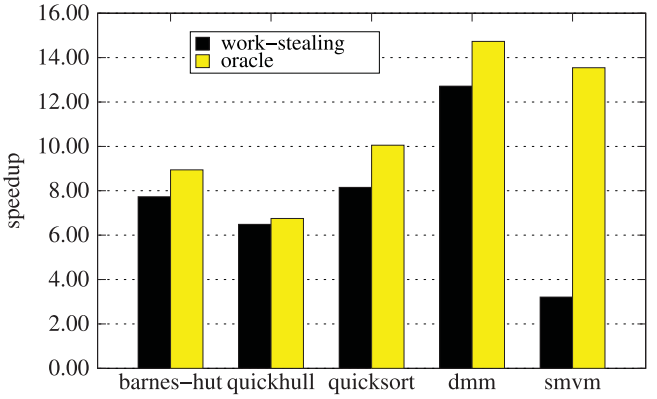

Fig. 7. Comparison of the speedup on 16 processors. Higher bars are better.

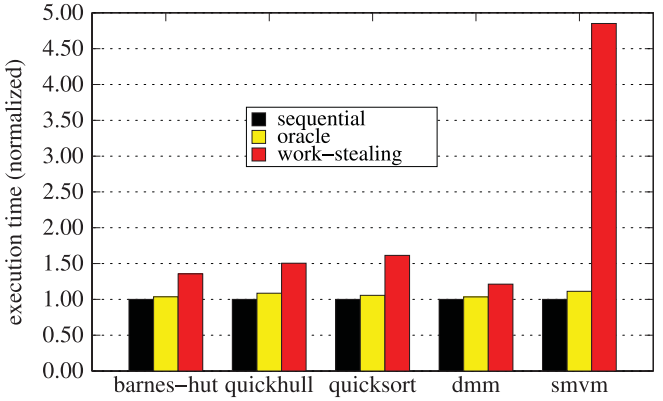

Fig. 8. Comparison of execution times (normalized) on a single processor. Lower bars are better.

9.5 Performance

For every benchmark, we measure several values. T_{seq} denotes the time to execute the sequential version of the program. We obtain the sequential version of the program by replacing each parallel tuple with an ordinary tuple and erasing complexity functions, so that the sequential version includes none of the thread-creation overheads. T^{P}_{par} denotes the execution time with work stealing on P processors. T^{P}_{orc} denotes the execution time of our oracle-based work stealing on P processors.

The most important results of our experiments come from comparing plain work stealing and our oracle-based work stealing side by side. Figure 7 shows the speedup on 16 processors for each of our benchmarks, that is, the values $T^{16}_{\mathsf{par}}/T_{\mathsf{seq}}$ and $T^{16}_{\mathsf{orc}}/T_{\mathsf{seq}}$. The speedups show that, on 16 cores, our oracle implementation is always between 4% and 76% faster than work stealing.

The fact that some benchmarks benefit more from our oracle implementation than others is explained by Figure 8. This plot shows execution time for one processor, normalized with respect to the sequential execution times. In other words, the values plotted are 1, $T^{1}_{\mathsf{orc}}/T_{\mathsf{seq}}$, and $T^{1}_{\mathsf{par}}/T_{\mathsf{seq}}$. The values $T^{1}_{\mathsf{orc}}/T_{\mathsf{seq}}$ range from 1.03 to 1.13

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

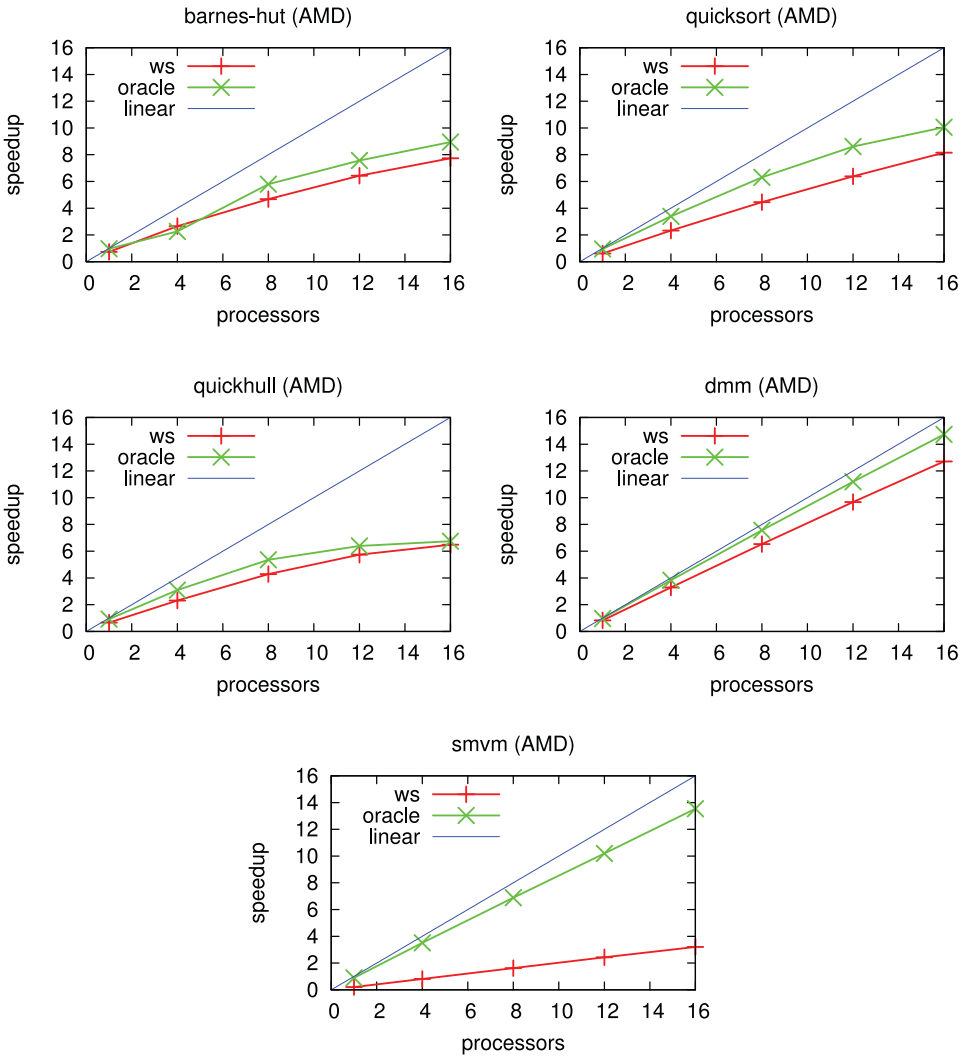

Fig. 9. Comparison between work stealing and oracle.

(with an average of 1.07), indicating that the thread-creation overheads in the oracle implementation do not exceed 13% of the raw work in any benchmark. The cases where we observe large improvements in speedup are the same cases where there is a large difference between sequential execution time and plain work-stealing execution time. When the difference is large, there is much room for our implementation to improve on work stealing, whereas when the difference is small we can only improve the execution time by a limited factor.

Figure 9 shows speedup curves for each of our experiments, that is, values of $T^P_{\mathsf{par}}/T_{\mathsf{seq}}$ and $T^P_{\mathsf{orc}}/T_{\mathsf{seq}}$ against the number of processors P. The curves show that our oracle implementation generally scales well up to 16 processors. There is one exception, which is the quickhull benchmark. For this benchmark, the curve tails off after reaching 12 processors. We believe that this tailing is either due to a lack of

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

parallelism in the program or to a bottleneck on memory accesses. Notice, however, that our scheduler does not fall below plain work stealing.

10 Related work

10.1 Cutting-off excess parallelism

This study is not the first to propose using cost prediction to determine when to cut-off parallelism. One approach, developed in early work in functional programming, uses list size to determine cut-offs (Huelsbergen *et al.*, 1994). Using list size alone is limited, because the technique assumes linear work complexity for every parallel operation.

Another way to handle cost prediction is to use the depth and height of the recursion tree (Weening, 1989; Pehoushek and Weening, 1990). But depth and height are not, in general, the most direct means to predict the execution time of sub-computations. In our oracle scheduling, we ask for either the programmer or compiler to provide for each function a cost function that expresses the asymptotic cost of applying the function.

Lopez *et al.* take this approach as well, but in the context of logic programming (Lopez *et al.*, 1996). On the surface, their technique is similar to our oracle scheduling, except that their cost estimators do not utilize profiling to estimate constant factors. An approach without constant-factor estimation is overly simplistic for modern processors, because it relies on complexity function predicting execution time exactly. On modern processors, execution time depends heavily on factors such as caching, pipelining, etc. and it is not feasible in general to predict execution time from a complexity function alone.

10.2 Reducing thread-creation cost

One approach to the granularity problem is to focus on reducing the cost of creating a thread, rather than limiting the number of threads created. This approach is taken by implementations of work stealing with lazy thread creation (Mohr *et al.*, 1990; Feeley, 1992; Feeley, 1993; Frigo *et al.*, 1998; Hiraishi *et al.*, 2009; Rainey, 2010; Sanchez *et al.*, 2010; Tzannes *et al.*, 2014). In lazy thread creation, the work stealing scheduler is implemented so as to avoid, in the common case, the major scheduling costs, in particular, those of inter-processor communication. But, in even the most efficient lazy thread creation, there is still a non-negligable scheduling cost for each implicit thread.

Lazy Binary Splitting is an improvement to lazy thread creation that applies to parallel loops (Tzannes *et al.*, 2014). The crucial optimization comes from extending the representation of a thread so that multiple loop iterations can be packed into a single thread. This representation enables the scheduler to both avoid creating closures and executing deque operations for most iterations. A limitation of Lazy Binary Splitting is that it addresses only parallel loops whose iteration space is over integers. Lazy Tree Splitting generalizes Lazy Binary Splitting to handle parallel aggregate operations that produce and consume trees, such as map and

reduce (Bergstrom *et al.*, 2010). Lazy Tree Splitting is limited, however, by the fact that it requires a special cursor data structure to be defined for each tree data structure.

10.3 Amortizing per-thread costs

Aharoni *et al.* study the granularity problem in the setting of distributed computing (Aharoni *et al.*, 1992), where the crucial issue is how to minimize the cost of inter-processor communication. In their setting, the granularity problem is modeled as a staging problem, in which there are two stages. The first stage consists of a set of processor-local thread pools and the second stage consists of a global thread pool. Moving a thread to the global thread pool requires inter-processor communication. The crucial decision is how often each processor should promote threads from its local thread pool to the global thread pool. We consider a different model of staging in which there is one stage for parallel evaluation and one for sequential evaluation.

The approach proposed by Aharoni *et al.* is based on an online algorithm called CG. In this approach, it is assumed that the cost of moving a thread to the global thread pool is an integer constant, called g. The basic idea is to use amortization to reduce the scheduling total cost of moving threads to the global thread pool. In particular, for each thread that is moved to the global thread pool, CG ensures that there are at least $g + 1$ threads added to the local thread pool. Narlikar describes a similar approach based on an algorithm called DFDeques (Narlikar, 1999). Just as with work stealing, even though the scheduler can avoid the communication costs in the common case, the scheduler still has to pay a non-negligible cost for each implicit thread.

10.4 Flattening and fusion

Flattening is a well-known program transformation for nested parallel languages (Blelloch and Sabot, 1990). Implementations of flattening include NESL (Blelloch *et al.*, 1994) and Data Parallel Haskell (Peyton Jones, 2008). Flattening transforms the program into a form that maps well onto SIMD architectures. Flattened programs are typically much simpler to schedule at run time than nested programs, because much of the schedule is predetermined by the flattening (Spoonhower, 2009). Controlling the granularity of such programs is correspondingly much simpler than in general. A limitation of existing flattening is that certain classes of programs generated by the translation suffer from space inefficiency (Blelloch and Greiner, 1996), as a consequence of the transformation making changes to data structures defined in the program. Our transformation involves no such changes.

The NESL (Blelloch *et al.*, 1994) and Data Parallel Haskell (Peyton Jones, 2008) compilers implement fusion transformation in order to increase granularity. Fusion transforms the program to eliminate redundant synchronization points and intermediate arrays. Although fusion reduces scheduling costs by combining adjacent parallel loops, it is not relevant to controlling granularity within loops. As such, fusion is orthogonal to our oracle based approach.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

10.5 Scheduling and locality

In our approach, we make scheduling decisions based on a property of the intrinsic semantics of pieces of computation: the work of parallel threads. In other related work on scheduling for data locality (Acar *et al.*, 2002; Blelloch and Gibbons, 2004), recent work showed that basing scheduling decisions on the space usage of parallel threads, a different property of intrinsic semantics, can be helpful in improving locality. For example, a class of "space-bounded schedulers" (Chowdhury *et al.*, 2010; Cole and Ramachandran, 2010; Blelloch *et al.*, 2011) are shown to be well suited for improved locality on deep memory hierarchies. Since space usage of a parallel thread can be bounded by its work, our techniques may be helpful in scheduling for improved data locality.

10.6 Cost semantics

To give an accurate accounting of thread-creation of overheads in implicitly parallel languages, we use a cost semantics, where evaluation steps (derivation rules) are decorated with work and span information or "costs". This information can then be used directly to bound running time on parallel computers by using standard scheduling theorems that realize Brent's bound. Many previous approaches also use the same technique to study work-span properties, some of which also make precise the relationship between cost semantics and the standard DAG models (Blelloch and Greiner, 1995; Blelloch and Greiner, 1996; Spoonhower *et al.*, 2007). The idea of instrumenting evaluations to generate cost information goes back to the early 90s (Rosendahl, 1989; Sands, 1990).

10.7 Inferring complexity bounds

Our implementation of oracle scheduling requires the programmer to enter complexity bounds for all parallel tasks. In some cases, these bounds can be inferred by various static analyses, for example, using type-based and other static analyses (e.g., Crary and Weirich (2000), Jost *et al.* (2010)), symbolic techniques (e.g., Goldsmith *et al.* (2007), Gulwani *et al.* (2009)). Our approach can benefit from these approaches by reducing the programmer burden, making it ultimately easier to use the proposed techniques in practice.

10.8 Further validation

Since the publication of the conference version of this paper (Acar *et al.*, 2011), we have conducted additional research that further validates the proposed techniques, extends their applicability, and explores extensions to other problem domains outside of the fork-join idiom considered here. In addition to the ML library described in this paper, we implemented our techniques as an optimized, lower level library for the C++ language (Acar *et al.*, 2015a), which uses a highly optimized work-stealing scheduler to achieve efficiency (Acar *et al.*, 2013). This library offers techniques for writing parallel programs in the full generality of the C++ language and is now a

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

reasonably mature software system. We used the library in undergraduate teaching at Carnegie Mellon University (Acar and Blelloch, 2015a, 2015b) and elsewhere in intensive courses on parallelism. Our experience in developing this library and the feedback from over 500 students taught thus far suggest that the techniques proposed in this paper can be implemented and used in practice.More recently, we have also started researching the granularity problem in more unstructured parallel-computing problems, including for example, graphs, and presented an algorithm for unordered parallel depth-first-search that achieves its efficiency by carefully controlling granularity (Acar *et al.*, 2015b). We are currently working on extending the automatic granularity-control techniques presented here to such unstructured problems.

11 Challenges for programming with complexity functions

The oracle-guided semantics that we presented is relatively general purpose and broadly applicable. Our implementation of this semantics based on the approximation algorithm (Section 7) and the translation (Section 8), however, make the following two assumptions.

- Each complexity function in the given parallel program is accurate and efficient.
- Our approximation algorithm can always approximate constant factors effectively.

Although these two requirements are easily met for large classes of programs, there are challenging cases. In what follows, we describe several such challenging cases that we have encountered.

11.1 Complexity functions requiring auxiliary data structures

There are programs for which providing a constant-time function that computes the complexity of a computation requires the pre-computation of an auxiliary data structure. We ran into such an example when implementing the sparse-matrix by dense vector multiplication benchmark program. Essentially, we needed to pre-compute a prefix sum array over the input data in order to determine the number of non-zero values covered by a range of consecutive rows in the sparse matrix. Fortunately, the cost of this pre-computation turned out to be relatively small in front of the rest of the computations, so the overhead associated with the introduction of the complexity function was limited. For additional details on this example, we refer to the evaluation section of an earlier description of our work (Acar *et al.*, 2011).

11.2 Complexity functions for higher order functions

Another challenge is that of providing complexity functions for higher order functions. Consider for example a "map" function that can be used to apply a given function f to all the leaves of a binary tree. Clearly, the asymptotic complexity of a call of the map function to a function f and a tree depends not just on the tree but also on the evaluation of the asymptotic complexity function f on the various leaves of

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

the tree. It is far from obvious how to handle the general case properly. Nevertheless, there are at least two large classes of f functions for which we are able to implement a useful asymptotic complexity function for predicting the cost of calls to map.

The first class includes constant-time functions. If we map a constant function to all the leaves of a binary tree, then the asymptotic cost is proportional to the number of leaves in the tree. Therefore, to evaluate the cost of the map function on any subtree, it suffices to pre-compute and store, for each node from the tree, the size of the subtree rooted at this node. Of course, caching additional values in the data structures and maintaining those values upon changes induce additional overheads, but it seems hard to control granularity without any form of additional information.

This approach generalizes to the application of map to functions that do not necessarily take constant time but take a time that depends on the *weight* of the leaf it is applied to, for some definition of *weight*. For example, consider a binary tree that stores lists of integers in its leaves, and assume that we map to this tree a function that increments all the integers stored in the leaves. In this case, we define the weight of a leaf as the length of the list it stores, and define the weight of a subtree as the sum of the weight of its leaves. If we pre-compute the weight of every node in the tree, then we are able to provide a useful asymptotic cost function.

The implementation of the complexity function is not the only challenge associated with higher order functions: evaluation of the constant factors is another. For example, the constant factors of the map function depend both on the constant factors of the function f passed to it and on constant factors inherent to the traversal of tree data structures. One possible solution is to allocate one different estimator for every instantiation of the map function. However, doing so results in imperfect sharing of constant factors, possibly leading to longer convergence phases. In summary, higher order functions are associated with a number of open challenges.

11.3 Mismatch between average and worst-case complexity

Our ability to implement the oracle relies on the assumption that we are able to predict asymptotically the amount of work involved in a computation. However, there are cases where the amount of computation to be performed depends on the data itself, and not just on the size of the data.

For example, consider string comparison. On many pairs of input, the comparison terminates quickly because of a mismatch occurring in the first few characters of the two strings. However, the worst-case execution time is $O(n)$. If we naively use $O(n)$ as the complexity function, but the execution is most frequently $O(1)$, we will conclude that the constant factors are tiny. At this point, if we receive a pair of two identical strings, we will incorrectly predict the comparison to run fast, and as a result we will fully sequentialize the linear-time comparison of the two strings, even though it could have been performed faster in parallel.

Another example is that of (non-randomized) quick sort. The expected complexity is $O(n \log n)$, but in the worst case, it can run in $O(n^2)$. We are tempted to use $O(n \log n)$ for making predictions. What happens if we do so but then run on an input that triggers a quadratic behavior? We will likely sequentialize sub-computations

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

of much larger size than we ideally aim for. However, it turns out that the inputs that triggers quadratic behavior are also those for which the quick sort algorithm does not exhibit any parallelism. In other words, we could not have sorted the items faster using quick sort by controlling granularity differently.

11.4 Initial estimation of the constant factors

The approach described in this paper has another limitation related to the way that we currently handle the initial convergence phase for the estimation of the constant factors. This limitation may affect code whose structure is more complex than simple recursive divide-and-conquer functions.

To understand the limitation, recall that, when a program begins, we do not have any estimation for its constant factors, and that we initially consider a pessimistic (i.e., large) value for the constants. By doing so, we typically begin by sequentializing only tiny pieces of work; later, when a constant factor is measured to be actually smaller than its initial pessimistic value, we are then able to sequentialize larger pieces of work. This scheme suffices in the case of simple recursive functions, because the constant factors that are measured one level of recursion can be exploited to make predictions at a higher level of recursion.

Consider, however, the case of two nested loops, both implemented as basic divide-and-conquer recursive functions. The constant-factor estimation for the inner loop will converge properly. However, the constant-factor estimation for the outer loop may never get updated because a measured run is not necessarily triggered. Even when the full execution of the inner loop is systematically sequentialized, consecutive iterations of the outer loop never get sequentialized, because the estimator of the outer loop, which does not share information with the estimator of the inner loop, continues to use the initial pessimistic constant estimation.

Intuitively, our approximation algorithm is facing a bootstrapping problem here: On the one hand, for sequentializing pieces of work, we need some reasonably accurate estimate of the constant factors; on the other hand, we may only obtain these estimations through sequentialized computations. We leave to future work the design of better approximation algorithms and runtime techniques for either propagating more information or for estimating upper bounds to the constant factors, so as to be able to escape this bootstrapping issue.

12 Conclusion

In this paper, we present an analysis of the impact of thread-creation costs on the performance of implicitly parallel programs and provide a solution for controlling these costs based on an oracle. We formulate our solutions in the context of a nested-parallel functional language and prove that it successfully controls thread-creation costs by using an amortization technique, known also as granularity control. We then present an approximation algorithm for realizing our oracle-guided semantics with the help of asymptotic complexity functions provided by the programmer. We implement our techniques and perform an experimental evaluation, which

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

shows that the approach is effective in controlling the thread-creation costs without detrimentally affecting the performance in the benchmarks considered. While this paper makes important progress towards solving the granularity-control problem, it also shows that there is more work to be done. For example, we have seen theoretically that our approximation algorithm only works for certain kinds of computations. While we have also validated the techniques proposed in this paper in other research projects and in teaching (Section 10.8), our implementations and experiments are still relatively new and would benefit from a more thorough evaluation that considers larger and more varied benchmarks.

References

Acar, U. A., & Blelloch, G. (2015a). 15210: Algorithms: Parallel and sequential. Accessed August 2016. Available at: `http://www.cs.cmu.edu/~15210/`.

Acar, U. A., & Blelloch, G. (2015b). Algorithm design: Parallel and sequential. Accessed August 2016. Available at: `http:www.parallel-algorithms-book.com`.

Acar, U. A., Blelloch, G. E. & Blumofe, R. D. (2002). The data locality of work stealing. *Theory Comput. Syst.* **35**(3), 321–347.

Acar, U. A., Charguéraud, A., & Rainey, M. (2011). Oracle scheduling: Controlling granularity in implicitly parallel languages. In *Proceedings of ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA)*, pp. 499–518.

Acar, U. A., Charguéraud, A. & Rainey, M. (2013). Scheduling parallel programs by work stealing with private deques. In PPoPP '13.

Acar, U. A., Charguéraud, A., & Rainey, M. (2015a). An introduction to parallel computing in c++. Available at: `http://www.cs.cmu.edu/15210/pasl.html`.

Acar, U. A., Charguéraud, A., & Rainey, M. (2015b). A work-efficient algorithm for parallel unordered depth-first search. In *Proceedings of Acm/ieee conference on high performance computing (sc)*. New York, NY, USA: ACM.

Aharoni, G., Feitelson, D. G. & Barak, A. (1992). A run-time algorithm for managing the granularity of parallel functional programs. *J. Funct. Program.* **2**, 387–405.

Arora, N. S., Blumofe, R. D., & Plaxton, C. G. (1998). Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures. SPAA '98*. ACM Press, pp. 119–129.

Arora, N. S., Blumofe, R. D. & Plaxton, C. G. (2001). Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.* **34**(2), 115–144.

Barnes, J. & Hut, P. (December 1986). A hierarchical $O(N \log N)$ force calculation algorithm. *Nature* **324**, 446–449.

Bergstrom, L., Fluet, M., Rainey, M., Reppy, J., & Shaw, A. (2010). Lazy tree splitting. Icfp 2010. ACM Press, pp. 93–104.

Blelloch, G., & Greiner, J. (1995). Parallelism in sequential functional languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture. FPCA '95*. ACM, pp. 226–237.

Blelloch, G. E., Fineman, J. T., Gibbons, P. B. & Simhadri, H. V. (2011). Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures. SPAA, '11*, pp. 355–366.

Blelloch, G. E. & Gibbons, P. B. (2004). Effectively sharing a cache among threads. In *SPAA*.

https://doi.org/10.1017/S0956796816000101 Published online by Cambridge University Press

Blelloch, G. E., & Greiner, J. (1996). A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM Sigplan International Conference on Functional Programming*. ACM, pp. 213–225.

Blelloch, G. E., Hardwick, J. C., Sipelstein, J., Zagha, M. & Chatterjee, S. (1994). Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.* **21**(1), 4–14.

Blelloch, G. E. & Sabot, G. W. (February 1990). Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.* **8**, 119–134.

Blumofe, R. D. & Leiserson, C. E. (September 1999). Scheduling multithreaded computations by work stealing. *J. ACM* **46**, 720–748.

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

Chakravarty, M. M. T., Leshchinskiy, R., Peyton Jones, S., Keller, G. & Marlow, S. (2007). Data parallel Haskell: a status report. In *Workshop on declarative aspects of multicore programming*. DAMP '07, pp. 10–18.

Chowdhury, R. A., Silvestri, F., Blakeley, B. & Ramachandran, V. 2010 (Apr.). Oblivious algorithms for multicores and network of processors. In *Proceedings of International Symposium on Parallel Distributed Processing (ipdps)*, pp. 1–12.

Cole, R. & Ramachandran, V. (2010). Resource oblivious sorting on multicores. In *Proceedings of the 37th International Colloquium Conference on Automata, Languages and Programming. ICALP'10*. Springer-Verlag, pp. 226–237.

Crary, K. & Weirich, S. (2000). Resource bound certification. In *Proceedings of the 27th ACM Sigplan-Sigact Symposium on Principles of Programming Languages. POPL '00*, pp. 184–198.

Feeley, M. (1992). A message passing implementation of lazy task creation. In *Proceedings of Parallel symbolic computing*, pp. 94–107.

Feeley, M. (1993). *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD Thesis, Brandeis University, Waltham, MA, USA, UMI Order No. GAX93-22348.

Fluet, M., Rainey, M. & Reppy, J. (2008). A scheduling framework for general purpose parallel languages. In *Proceedings of ACM Sigplan International Conference on Functional Programming (icfp)*. ACM, pp. 241–252.

Fluet, M., Rainey, M., Reppy, J. & Shaw, A. (2011). Implicitly threaded parallelism in Manticore. *J. Funct. Program.* **20**(5–6), 1–40.

Frens, J. D. & Wise, D. S. (1997). Auto-blocking matrix-multiplication or tracking blas3 performance from source code. In *Proceedings of the Sixth ACM Sigplan Symposium on Principles and Practice of Parallel Programming. PPOPP '97*. New York, NY, USA: ACM, pp. 206–216.

Frigo, M., Leiserson, C. E. & Randall, K. H. (1998). The implementation of the Cilk-5 multithreaded language. In *Pldi*, pp. 212–223.

Goldsmith, S. F., Aiken, A. S. & Wilkerson, D. S. (2007). Measuring empirical computational complexity. In *Proceedings of the 6th joint meeting of the european software engineering conference and the acm symposium on the foundations of software engineering*, pp. 395–404.

Gulwani, S., Mehra, K. K. & Chilimbi, T. (2009). Speed: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM Sigplan-Sigact Symposium on Principles of Programming Languages*, pp. 127–139.

Halstead, R. H. (1985). Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.* **7**, 501–538.

Hiraishi, T., Yasugi, M., Umatani, S. & Yuasa, T. (2009). Backtracking-based load balancing. In *Ppopp '09*. ACM, pp. 55–64.

Huelsbergen, L., Larus, James R. & Aiken, A. (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*, pp. 79–90.

Jost, S., Hammond, K., Loidl, H. & Hofmann, M. (2010). Static determination of quantitative resource usage for higher-order programs. In *Principles of programming languages (popl)*, pp. 223–236.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D. & Vouillon, J. (2005). *The Objective Caml System.*

Lopez, P., Hermenegildo, M. & Debray, S. (June 1996). A methodology for granularity-based control of parallelism in logic programs. *J. Symbol. Comput.* **21**, 715–734.

Mohr, E., Kranz, D. A. & Halstead Jr., R. H. (1990). Lazy task creation: A technique for increasing the granularity of parallel programs. In *Conference Record of the 1990 ACM Conference on Lisp and Functional Programming*. New York, New York, USA: ACM Press, pp. 185–197.

Narlikar, G. J. (1999). *Space-Efficient Scheduling for Parallel, Multithreaded Computations.* PhD Thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Pehoushek, J. & Weening, J. (1990). Low-cost process creation and dynamic partitioning in Qlisp. of: Ito, Takayasu, & Halstead, Robert (eds), In *Parallel lisp: Languages and Systems. Lecture Notes in Computer Science*, vol. 441. Springer Berlin/Heidelberg, pp. 182–199.

Peyton Jones, S. L. (2008). Harnessing the multicores: Nested data parallelism in Haskell. In *Aplas*, p. 138.

Peyton Jones, S. L., Leshchinskiy, R., Keller, G. & Chakravarty, M. M. T. (2008). Harnessing the multicores: Nested data parallelism in Haskell. In *Fsttcs*, pp. 383–414.

Plummer, H. C. (March 1911). On the problem of distribution in globular star clusters. *Mon. Not. R. Astron. Soc.* **71**, 460–470.

Rainey, M. (August 2010). *Effective Scheduling Techniques for High-Level Parallel Programming Languages.* PhD thesis, University of Chicago.

Rosendahl, M. (1989). Automatic complexity analysis. In *Fpca '89: Functional Programming Languages and Computer Architecture*. ACM, pp. 144–156.

Sanchez, D., Yoo, R. M. & Kozyrakis, C. (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*. New York, NY, USA: ACM, pp. 311–322.

Sands, D. (September 1990). *Calculi for Time Analysis of Functional Programs.* PhD Thesis, University of London, Imperial College.

Sivaramakrishnan, K. C., Ziarek, L. & Jagannathan, S. (2014). Multimlton: A multicore-aware runtime for standard ml. *J. Funct. Program.* FirstView:1–62, 6.

Spoonhower, D. (2009). *Scheduling Deterministic Parallel Programs.* PhD Thesis, Pittsburgh, PA, USA: Carnegie Mellon University.

Spoonhower, D., Blelloch, G. E., Harper, R. & Gibbons, P. B. (2008). Space profiling for parallel functional programs. In International Conference on Functional Programming.

Tzannes, A., Caragea, G. C., Vishkin, U. & Barua, R. (September 2014). Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *TOPLAS* **36**(3), 10:1–10:51.

Valiant, L. G. (August 1990). A bridging model for parallel computation. *CACM* **33**, 103–111.

Weening, J. S. (1989). *Parallel Execution of Lisp Programs.* PhD Thesis, Stanford University. Computer Science Technical Report STAN-CS-89-1265.