Oracle Scheduling: Controlling Granularity in Implicitly Parallel Languages

Umut A. Acar Arthur Charguéraud Mike Rainey

Max-Planck Institute for Software Systems

{umut,charguer,mrainey}@mpi-sws.org

Abstract

A classic problem in parallel computing is determining whether to execute a task in parallel or sequentially. If small tasks are executed in parallel, the overheads due to task creation can be overwhelming. If large tasks are executed sequentially, processors may spin idle, resulting again in sub-optimal speedups. Although this "granularity problem" is identified to be an important problem, it is not well understood; broadly applicable solutions remain elusive.

We propose techniques for controlling the granularity in implicitly parallel programming languages to achieve parallel efficiency. To understand the importance of granularity control, we extent Brent's theorem (a.k.a.'s work-time principle) to include task creation overheads. Using a cost semantics for a general-purpose language in the style of lambda calculus with parallel tuples, we then show that task-creation overheads can slowdown parallel execution by a multiplicative factor. We propose *oracle scheduling* to reduce these overheads by using estimates of the sizes of parallel tasks. We show that if the oracle provides in constant time estimates that are accurate within a constant multiplicative factor then oracle scheduling provable reduces the task-creation overheads for a class of parallel computations.

We realize the oracle scheduling by combining static and dynamic techniques. We require the programmer to provide the asymptotic complexity for parallel tasks and use run-time profiling to determine hardware-specific constant factors. We implement the proposed approach and propose a compiler for it as extension of the Manticore compiler for Parallel ML. Our empirical evaluation shows that we can reduce the run-time overheads due to task creation down to between 3 and 13 percent of the sequential time and can obtain scalable speedups when running on multiple processors.

1. Introduction

Explicit parallel programming provides full control over parallel resources by offering primitives for creating and managing parallel tasks, which are small, independent threads of control. As a result, the programmer can, at least in principle, write efficient parallel programs by performing a careful cost-benefit analysis to determine which tasks should be executed in parallel and under what conditions. This approach, however, often requires reasoning about low-level execution details, such data races or concurrent effects, which is known to be notoriously hard; it can also result in code that performs well in a particular hardware setting but not in others.

The complexities of parallel programming with explicit languages have motivated interest in *implicitly parallel languages*, such as Cilk [11], Manticore [14–16], Multilisp [19], NESL [7]. These languages enable the programmer to express parallelism implicitly via language constructs, e.g., parallel sequences, parallel arrays, parallel tuples. This implicit approach delegates the task of utilizing the parallelism exposed by the program to the compiler and the run-time system, enabling a high level of programming style. As an implicit parallel program executes, it exposes opportunities or parallelism (as indicated by the parallel constructs); the language run-time system creates parallel tasks as needed. To execute parallel tasks efficiently, implicit programming languages rely on a scheduler for distributing parallel tasks among the processors to perform load balancing. Various scheduling techniques and practical schedulers have been developed, including work-stealing schedulers [10], and depth-first-search schedulers [6].

Experience with implicitly parallel programs shows that one of the most important decisions that any implicit parallel language must make is determining whether or not to exploit an opportunity for parallelism by creating a parallel task. Put another way, the question is to determine which tasks to execute in parallel and which other tasks to execute sequentially. This problem, often referred to as the *granularity problem*, is important because creating a parallel task requires additional overhead and because every such overhead matters: since the speedups achievable via parallel computation is bounded by the number of processors, often a small constant factor, any increase in the overheads, however small, matters. When combined with the fact that many parallel programs naturally expose many more opportunities for parallelism than the number of available processors, creating many tasks can limit the practical efficiency of parallel programs.

No known broadly applicable solution to the granularity problem exists. Theoretical analysis often ignores task-creation overheads, yielding us no significant clues about how these overheads may affect efficiency. Practical implementations often focus on reducing task-creation overheads instead of attempting to control granularity. As a result, practitioners often deal with this issue by trying to estimate the right granularity of work that would be sufficiently large to execute in parallel. More specifically, programmers try to determine the input sizes at which tasks become too small to pay off the costs of parallel task creation and sequentialize such tasks. Since the running time of a task 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 bound to yield suboptimal results and/or non-portable code [36].

In this paper, we propose theoretical and practical techniques for the granularity problem in implicit parallel-programming languages. Our results include theorems that take into account the task-creation overheads to characterize their impact on parallel run time, which we show to be significant (Sections 2 and 4). To reduce these overheads, we consider a granularity control technique that relies on an oracle for determining the run-time of parallel tasks(Section 4). 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 describe how the oracle approach be realized in practice by combining with known schedulers; we call this technique *oracle scheduling* because it relies on an oracle to estimate task sizes and because it can be used with practically any other scheduler (Section 5). Finally, we propose an implementation of oracle scheduling that uses complexity functions defined by the user to approximate accurately run-time of parallel tasks Section 5. We present an implementation and evaluation of the proposed approach by extending a subset of the Caml language (Sections 6 and 7).

Brent's theorem [12], commonly called the work-time principle, characterizes arguably the most important property of parallel programs: that they can be executed efficiently with multiple processors. More precisely Brent shows that we can execute a computation with w *raw-work* and d *raw-depth*, which do not include task-creation overheads, in no more than $w/P + d$ steps on P processors using any greedy scheduler (note that the bound is tight within a factor of two). However attractive, the theorem ignores an important factor: task-creation overheads, which are assumed to be zero. To understand the impact of task-creation overheads, we therefore start with this fundamental theorem and generalize it to take them into account (Section 2). Specifically, we consider the standard directed-acyclic-graph (DAG) mode for parallel computations and show that a computation with *total work* $\mathcal{W}$ and *total depth* $\mathcal{D}$, where both include the task-creation overheads, can be executed in no more than $\mathcal{W}/P + \mathcal{D}$ steps.

Although generalized Brent's theorem yields a run-time bound that is symmetric to the original bound, the proof is entirely different. In fact, a straightforward generalization of the original proof only yields a weaker bound and requires reasoning about both the raw and the total work/depth. The reason for this increase in the proof complexity is that overheads are not like other unit work: they are indivisible (it is not realistic to assume that they can be divided into parts that can be performed piecewise) but they don't have unit costs. Perhaps the most important point about this result is that it show shows that the task-creation overheads contribute directly to the parallel run time and not in a surprising way: they are just like any other work (even though they are indivisible). We note that Brent's theorem also assumes a greedy scheduler that can find work immediately when available but this assumption is reasonably realistic: parallel schedulers can match Brent's bound asymptotically under mild assumptions.

To determine precisely the overheads of task creation in implicitly parallel programs, we consider a lambda calculus with parallel tuples and present a cost-semantics for evaluating expression of this language. The *cost semantics* yield raw work/depth and total work/depth of each evaluated expression. Using this cost semantics, we show that task creation overheads can be significant: a multiplicative factor times the raw-work. By an application of the generalized Brent's theorem, this implies that such multiplicative increases in work affect the parallel run-time directly. To reduce task-creation overheads, we propose an alternative *oracle semantics* that capture a known principle for avoiding the task-creation overheads: evaluate a task in parallel only if its is sufficiently large, i.e., greater than some constant κ. We show that the oracle semantics can decrease the overheads of task-creation by any (desired) constant factor κ, but only at the cost of increasing the total depth by a similarly large factor. This results suggests that in practice some care will be needed to select κ, because otherwise it can reduce the parallel slackness assumption [37] that some parallel schedulers assume to match the Brent's theorem.

The bounds with the oracle semantics suggests that we can reduce the task-creation overheads significantly, if we can realize the semantics in practice. This is impossible unfortunately because it requires the ability to determine a priori task-creation overheads and without incurring other overheads. We show, however, that a realistic oracle that can give constant-factor approximations to the task run times can still result in similar reductions in the overheads for a reasonably broad class of computations (Section 4). We also show that unless care is taken, the realistic oracle can actually further increase the called unless it is called periodically. This outcome, i.e., that attempts at controlling the granularity can actually backfire and slow down the program further, was 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.

To realize the oracle semantics in practice, we describe (Section 5) a scheduling technique that we call *oracle scheduling* that consists of a *task-size estimator* that can estimate actual run time of parallel tasks in constant-time within a constant factor of accuracy and a conventional greedy scheduling algorithms; many schedulers, e.g., work-stealing algorithm, depth-first schedulers are all greedy. Combined together, these can be used to perform efficient parallel task creation and scheduling by selectively executing in parallel only those tasks that have a large parallel run-time. We describe an instance of the oracle scheduler that relies on an estimator that uses asymptotic cost functions (asymtotic complex bounds) and judicious use of run-time profiling techniques to estimate actual run-times accurately and efficiently. This approach combines an interesting property of asymptotic complexity bounds, which assume away hardware dependent constant, and profiling techniques, which can be used to determine precisely these constants. In this work, we only consider programs for which the execution time is (with high probability) proportional to the value obtained by evaluating the asymptotic complexity expression.

We present a prototype implementation of the proposed approach (Section 6) by extending the OCAML language to support parallel tuples, and complexity functions, and translating programs written in this extended language to the PML (Parallel ML) language [15]. 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., [24] and references thereof). We extend the Manticore compiler for PML to support oracle scheduling and use it to compile generated PML programs. Our experiments (Section 7) show that oracle implementation can reduce the overheads of a single processor parallel execution to 3 and 13 percent of the sequential time When using 16 processors, we achieve 7- to 15-fold speedups on an AMD machine and 17- to 21-fold speedups on an Intel machine (Intel machines typically show superlinear effects).

2. Generalizing Brent's theorem

We represent a parallel computation with a directed acyclic graph, called *computation DAG*. Nodes in the graph represent atomic computations. Edges between nodes represent precedence relations, in the sense that an edge from a to b indicates that the execution of a must be completed before the execution of b can start. Every computation DAG includes

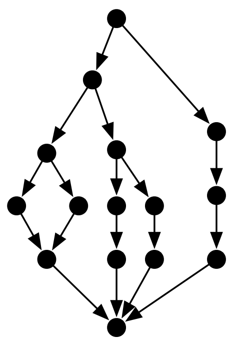

Figure 1. An example computation DAG.

a *source* node and a *sink* node, representing the starting and the end points of the computation, respectively. Those nodes are such that all nodes of a computation DAG are reachable from the source node, and the sink node is reachable from all nodes. An example computation DAG appears in Figure 1.

In the traditional computational model, every atomic computation is considered to take a single unit of time. In other words, every node has weight 1. In this setting, we can define the standard notion of work and depth, which we here call *raw-work* and *raw-depth*. The *raw-work* of a computation graph is equal to the total number of nodes that it contains. The *raw-depth* of the computation graph is equal to the total number of nodes along the longest path. Brent proved the following bound.

Theorem 2.1 (Brent's theorem) *Let G be a computation DAG with w raw work and d raw depth. Any greedy scheduler can execute the computation in G in time $O(\frac{w}{P} + d)$ on a P-processor parallel machine*

Proof We recall Brent's proof since our aim is to generalize it. Consider the nodes at depth i in the DAG, and assume there are w_i of them. A greedy scheduler can spend no more than time $\lceil \frac{w_i}{P} \rceil$ for executing those nodes. Summing up over the various depths, one can thus deduce that the total execution time does not exceed:

$$\sum_{i=1}^{d} \left\lceil \frac{w_i}{P} \right\rceil \le \sum_{i=1}^{d} (\frac{w_i}{P} + 1) \le \frac{\sum_{i=1}^{d} w_i}{P} + d \le \frac{w}{p} + d$$

□

Observe that the bound provided by Brent's theorem is tight, because the execution time is at least $\mathtt{max}\left(\frac{w}{P}, d\right)$.

This theorem does not take into account the overheads associated with task creation. So, we want to refine the model and generalize Brent's theorem. To that end, we consider that if a node creates parallel tasks then an extra computation cost τneeds to be paid for. In other words, any node that has an out-degree two or greater now has weight $1 + \tau$ instead of just 1. We then define the *total work* as the sum of the weights of all the nodes in this revised computation graph. Similarly, we define the *total depth* as the maximum weight of a path from the source to the sink in the revised graph. A first attempt at generalizing Brendt's theorem is as follows.

Theorem 2.2 (Naive generalization of Brent's theorem) *Let G be a computation DAG with $\mathcal{W}$ total work and d raw depth. Any greedy scheduler can execute these computations in time $O(\frac{\mathcal{W}}{P} + (1+\tau)d)$ on P processors.*

Proof Consider layers like in Brent's theorem, with the difference that at every layer there might be tasks of weight 1 and tasks of weight $1+\tau$. Observe that there is still exactly d levels. Hereafter, let r be a shorthand for $1+\tau$. Let $\mathcal{W}_i$ denote the sum of the weights of the nodes at level i. A greedy scheduler executes this work in a time less than $r\left\lceil\frac{\mathcal{W}_i}{rP}\right\rceil$. Thus, the total time execution is bounded by:

$$\sum_{i=1}^{d} r\left\lceil\frac{\mathcal{W}_i}{rP}\right\rceil \le \sum_{i=1}^{d} r(\frac{\mathcal{W}_i}{rP}+1) \le \frac{\mathcal{W}}{P} + rd \quad \square$$

In the above theorem, Brent's original theorem generalizes with respect to total work, in the sense that the ratio $\frac{w}{P}$ gets replaced by $\frac{\mathcal{W}}{P}$, however it does not generalize as well with respect to the depth, because the component d is replaced by $(1+\tau)d$ and not $\mathcal{D}$. For computations that involve task creation all along their critical path, D can be equal to $(1+\tau)d$, so in this case the naive generalization of Brent's theorem already gives a tight bound. However, there are computations for which $(1+\tau)d$ can be significantly bigger than the total depth $\mathcal{D}$. Typically, $\mathcal{D}$ might be of the form $d+n\tau$ for some n. In this case, the bound obtained is extremely loose. We remedy to this situation by establishing a tight bound that nicely generalizes the statement of Brent's theorem.

Theorem 2.3 (Generalized version of Brent's theorem) *Let G be a computation DAG with W total work and D total depth. Any greedy scheduler can execute these computations in time $O(\frac{W}{P} + D)$ on P processors.*

Proof The problem shares similarities with the classic problem known as $P|prec|C_{max}$ in scheduling theory. This problem consists in scheduling tasks on P machines in a way that minimize the total makespan, while satisfying a set of precedence constraints Our problem, however, differs in a significant way: we do not want to establish a bound for a particular scheduler, but instead we want to establish a result for a entire class of scheduler, covering all the schedulers that are greedy (they never wait if there is work to do) and on-line (they are not aware of the existence of a task until is becomes available). Our proof reuses a particular aspect of the proof of 2-optimality of the greedy "earliest-job-first" approximation algorithm for the problem $P|prec|C_{max}$. More specifically, we build a particular sequence of tasks iteratively, starting from the last one. The structure of our proof is, however, significantly different. In particular, the invariants are more complex because we are making fewer assumptions about the scheduler's policy.

Consider a scheduling of tasks by a greedy scheduler. Our goal is to prove a bound on the total execution time T. Let the tasks be labelled using integers from 1 to M. The duration of task i is written w_i, and the time at which it starts is written t_i. We call Δ_i the time interval $[t_i, t_i + w_i]$ during which the task i is executed. To capture the dependencies, we consider a set of precedence constraints: $i \prec j$ indicates that the task j depends directly on indirectly on the result of the task i. For the sake of the proof, we assume that the set of tasks includes a task of duration zero such that all other tasks depend on it. This task is scheduled at time 0. Similarly, we assume the existence of a task of duration zero such that this task depend on all other tasks. This task is scheduled at time T. Hereafter, let π denote a sequence of tasks of the form $\pi_1 \prec \pi_2 \prec \ldots \prec \pi_N$. We write $|\pi|$ the number of tasks in the path π and $||\pi||$ the sum of the duration of the tasks in that path, that is, the value $\sum_{n=1}^{N} \pi_n$. By definition of work and depth, we have $\mathcal{W} = \sum_{1=i}^{M} w_i$ and $\mathcal{D} = \max_\pi ||\pi||$. Our goal is to show $T \le \frac{\mathcal{W}}{P} + D$.

• Let $([u_k, v_k])_{k\in[1,K-1]}$ be the set of nonempty time intervals during which not all processors are working. We define $u_K = v_K = T$. The total time T decomposes into the total time during which all processors are busy, call it T_{full}, and the total time during which not all processors are busy, call it $T_{partial}$. We thus have $T = T_{full} + T_{partial}$. Technically, we have $T_{full} = \sum_{k=1}^{K-1}(u_{k+1} - v_k)$ and $T_{partial} = \sum_{k=1}^{K}(v_k - u_k)$. During the time when processors are fully busy, they execute an amount of work equal to $P \cdot T_{full}$ .This amount cannot exceed the total amount of work available, which is $\mathcal{W}$. So, we have $T_{full} \le \mathcal{W}/P$. In order to establish that $T \le \mathcal{W}/P + \mathcal{D}$, it therefore remains to show that $T_{partial} \le \mathcal{D}$.

• **Observation A:** If a task i starts after the time v_k, for some k, then there exists a task j that executes at time v_k and such that i depends on j. Formally,

$$\forall ik.\ t_i > v_k \Rightarrow \exists j.\ j \prec i \wedge v_k \in \Delta_j$$

To prove this, consider the set of tasks that i depends on, and add i itself to that set. Select from this set the subset of tasks that starts after v_k. Call j' the task among these that has the minimal starting time (i.e. $t_{j'}$ minimal). Now, consider all the tasks that j' depends on. Due to the minimality of $t_{j'}$, all those tasks must start before v_k. If none of those task is executing at the time v_k, then it means that the task j' could have been scheduled to start just before v_k. Indeed, there was a free scheduler at this point because the interval $[u_k, v_k]$ corresponds to a nonempty period of time where not all processors are busy. So, there must exists at least one task j that executes at time v_k and such that $j \prec j'$. We therefore have $j \prec i$ and $v_k \in \Delta_j$.

• **Observation B:** If we have a task i_1 that executes at time v_k then we can find a sequence of tasks $i_N \prec \ldots \prec i_1$ such that these tasks entirely cover the interval $[u_k, v_k]$.

Formally,

$$\forall i_1 k.\ v_k \in \Delta_{i_1} \Rightarrow \exists i_2 \dots i_N. \left\{ \begin{array}{l} i_N \prec \dots \prec i_1 \\ u_k \in \Delta_{i_N} \\ v_k - u_k = \sum_{n=1}^{N} ||\Delta_{i_n} \cap [u_k, v_k]|| \end{array} \right.$$

Above, the expression $||\Delta_{i_n} \cap [u_k, v_k]||$ corresponds to the aomunt of time that the task i spent being executed inside the time interval $[u_k, v_k]$. We construct the sequence (i_n) iteratively in such a way that the tasks are adjacents to each others. Technically, we have $t_{i_n} = t_{i_{n+1}} + w_{i_{n+1}}$ for $n \in [1, N-1]$. Initially, we only have i_1. At a given point in the construction, the sequence built up to index n. There are two cases. If $u_k \in \Delta_{i_n}$, then we are done ($N = n$). Otherwise, the task i_n must depend on a task that ends at time t_{i_n} If this was not the case then the task i_n could have been scheduled earlier. Indeed, we have $u_k < t_{i_n} \leq v_k$ so there is at least one processor available just before time t_{i_n}). We call i_{n+1} the task that preceeds i_n, and we then repeat the process. Since there are only a finite number of tasks, the process must end after finitely many iterations. Note that we must reach the date u_k at some point, because the last task that can be considered is the task that is scheduled at time 0, which is earlier than u_k.

• **Main induction:** we construct a sequence of tasks that belong to a same precedence path and that covers all the periods of time where not all processors are busy. To build this sequence, we exploit observation A to traverse periods of full activity and exploit observation B to cover periods of partial activity. More precisely, we prove by induction that, for any $L \in [1, K]$, there exists a path π such that the task at the head of the path π is running at time v_L and such that the sum of the execution time of the tasks involved in the path π, counting only the execution occuring in the interval $[u_L, u_K]$, is greater than the sum of the width of the intervals of the form $[u_k, v_k]$ for $k \geq L$. Formally,

$$\forall L.\ \exists \pi.\ u_L \in \Delta_{\mathsf{hd}(\pi)} \wedge \sum_{k=1}^{L} (v_k - u_k) \leq \sum_{n=1}^{|\pi|} ||\Delta_{\pi_n} \cap [u_L, u_K]||$$

The base case is $L = K$. In this case, we define π as the singleton path made of the tasks that depends on all the others. This task is executed at time u_K (which is equal to T), so we have $u_K \in \Delta_{\mathsf{hd}(\pi)}$. Since $v_K = u_K$, the two sums involved are both equal to zero, so we are done for the base case.

Now, assume the result true for L, and let us establish it for $L+1$. By induction hypothesis, there exists a path π such that $u_L \in \Delta_i$, where i denotes the head of the path π, and such that $\sum_{k=1}^{L} (v_k - u_k) \leq \sum_{n=1}^{|\pi|} ||\Delta_{\pi_n} \cap [u_L, u_K]||$. The first step consists in extending the path π into a path π' whose head task, call it j_1, is executing at time v_{L+1}. There are two cases, if $t_i \leq v_{L+1}$, then we can simply define $\pi' = \pi$ and we have $j_1 = i$. Otherwise, $t_i > v_{L+1}$, so we can apply observation A to get a task j_1 such that $j_1 \prec i$ and $v_{L+1} \in \Delta_{j_1}$, and we then define $\pi' = j_1 \cdot \pi$. Now, we apply observation B, which asserts the existence of a sequence of tasks $j_N \prec \dots \prec j_2 \prec j_1$ such that $u_{K+1} \in \Delta_{j_N}$ and $v_{L+1} - u_{L+1} = \sum_{n=1}^{N} ||\Delta_{j_n} \cap [u_k, v_k]||$. The path π'' defined as $j_N \cdot \dots \cdot j_2 \cdot j_1 \cdot \pi'$ covers the time interval $[u_{L+1}, u_K]$. This path can be used to conclude. First, the head of the path π'' is the task j_N, which satisfies $u_{K+1} \in \Delta_{j_N}$ as required. Second, the required inequality is as shown next.

$$\begin{array}{ll} & \sum_{n=1}^{|\pi''|} ||\Delta_{\pi''_n} \cap [u_{L+1}, u_K]|| \\ \geq & \quad \sum_{n=1}^{N} ||\Delta_{j_n} \cap [u_{L+1}, u_{L+1}]|| \\ & + \sum_{n=1}^{|\pi|} ||\Delta_{\pi_n} \cap [u_L, u_K]|| \\ \geq & (v_{L+1} - u_{L+1}) + \sum_{k=1}^{L} (v_k - u_k) \\ \geq & \sum_{k=1}^{L+1} (v_k - u_k) \end{array}$$

The case where $j_1 = i$ is a bit delicate. This case occurs when the execution of task i intersects with several periods of time during which not all processors are working. In this case, we also have $\pi''_N = i$, so a part of the task i appears as $||\Delta_{\pi''_N} \cap [u_{L+1}, u_K]||$ and another part appears as $||\Delta_{j_1} \cap [u_{L+1}, u_{L+1}]||$.

• **Conclusion:** We construct a path π by applying the result from the main induction with $L = 1$. We can then establish the inequality $T_{partial} \leq \mathcal{D}$ as follows.

$$\begin{array}{l} T_{partial} = \sum_{k=1}^{K} (v_k - u_k) \leq \sum_{n=1}^{|\pi|} ||\Delta_{\pi_n} \cap [u_1, u_K]|| \\ \leq \sum_{n=1}^{|\pi|} \Delta_{\pi_n} \leq ||\pi|| \leq \max_{\pi'} ||\pi'|| = \mathcal{D} \end{array}$$

□

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

Figure 2. Abstract syntax of the source language

3. Source language

To give an accurate account of cost of task creation, and to specify precisely 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 task and the cost of consulting an external oracle.

Syntax The source language includes recursive functions, pairs, sum types, and parallel tuples. Parallel tuples enable expressing computations that can be performed in parallel, similar to the fork-join or nested data parallel computations. Note that we only consider parallel tuples of arity two. Parallel tuples of higher arity can be easily represented using

(value)

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

(let)

$$\frac{e_1 \Downarrow^{\alpha} v_1, (w_1, d_1), (\mathcal{W}_1, \mathcal{D}_1) \qquad e_2[v_1/x] \Downarrow^{\alpha} v, (w_2, d_2), (\mathcal{W}_2, \mathcal{D}_2)}{(\texttt{let}\, x \;=\; e_1 \,\texttt{in}\, e_2) \Downarrow^{\alpha} v, (w_1 + w_2 + 1, d_1 + d_2 + 1), (\mathcal{W}_1 + \mathcal{W}_2 + 1, \mathcal{D}_1 + \mathcal{D}_2 + 1)}$$

(app)

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

(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)

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

(case-right)

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

(tuple)

$$\frac{e_1 \Downarrow^{\alpha} v_1, (w_1, d_1), (\mathcal{W}_1, \mathcal{D}_1) \qquad e_2 \Downarrow^{\alpha} v_2, (w_2, d_2), (\mathcal{W}_2, \mathcal{D}_2)}{(e_1, e_2) \Downarrow^{\alpha} (v_1, v_2), (w_1 + w_2 + 1, d_1 + d_2 + 1), (\mathcal{W}_1 + \mathcal{W}_2 + 1, \mathcal{D}_1 + \mathcal{D}_2 + 1)}$$

(ptuple-seq)

$$\frac{e_1 \Downarrow^{\mathsf{seq}} v_1, (w_1, d_1), (\mathcal{W}_1, \mathcal{D}_1) \qquad e_2 \Downarrow^{\mathsf{seq}} v_2, (w_2, d_2), (\mathcal{W}_2, \mathcal{D}_2)}{(|e_1, e_2|) \Downarrow^{\mathsf{seq}} (v_1, v_2), (w_1 + w_2 + 1, d_1 + d_2 + 1), (\mathcal{W}_1 + \mathcal{W}_2 + 1, \mathcal{D}_1 + \mathcal{D}_2 + 1)}$$

(ptuple-par)

$$\frac{e_1 \Downarrow^{\mathsf{par}} v_1, (w_1, d_1), (\mathcal{W}_1, \mathcal{D}_1) \qquad e_2 \Downarrow^{\mathsf{par}} v_2, (w_2, d_2), (\mathcal{W}_2, \mathcal{D}_2)}{(|e_1, e_2|) \Downarrow^{\mathsf{par}} (v_1, v_2), (w_1 + w_2 + 1, \texttt{max}\,(d_1, d_2) + 1), (\mathcal{W}_1 + \mathcal{W}_2 + 1 + \tau, \texttt{max}\,(\mathcal{D}_1, \mathcal{D}_2) + 1 + \tau)}$$

(ptuple-orc-parallelize)

$$\frac{w_1 > \kappa \wedge w_2 > \kappa \qquad e_1 \Downarrow^{\mathsf{orc}} v_1, (w_1, d_1), (\mathcal{W}_1, \mathcal{D}_1) \qquad e_2 \Downarrow^{\mathsf{orc}} v_2, (w_2, d_2), (\mathcal{W}_2, \mathcal{D}_2)}{(|e_1, e_2|) \Downarrow^{\mathsf{orc}} (v_1, v_2), (w_1 + w_2 + 1, \texttt{max}\,(d_1, d_2) + 1), (\mathcal{W}_1 + \mathcal{W}_2 + 1 + \tau + \phi, \texttt{max}\,(\mathcal{D}_1, \mathcal{D}_2) + 1 + \tau + \phi)}$$

(ptuple-orc-sequentialize)

$$\frac{\begin{array}{c} w_1 \le \kappa \vee w_2 \le \kappa \\ e_1 \Downarrow^{(\mathsf{if}\; w_1 \le \kappa\; \mathsf{then\; seq\; else\; orc})} v_1, (w_1, d_1), (\mathcal{W}_1, \mathcal{D}_1) \quad e_2 \Downarrow^{(\mathsf{if}\; w_2 \le \kappa\; \mathsf{then\; seq\; else\; orc})} v_2, (w_2, d_2), (\mathcal{W}_2, \mathcal{D}_2) \end{array}}{(|e_1, e_2|) \Downarrow^{\mathsf{orc}} (v_1, v_2), (w_1 + w_2 + 1, d_1 + d_2 + 1), (\mathcal{W}_1 + \mathcal{W}_2 + 1 + \phi, \mathcal{D}_1 + \mathcal{D}_2 + 1 + \phi)}$$

Figure 3. Dynamic cost semantics

those of arity two. (We leave to future work the investigation of an optimized treatment of n-ary parallel tuples.)

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 references; in this case, however, they contribute no additional insight and thus are omitted for clarity.

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. 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 parallel tuples are always evaluated in parallel regardless of their size. The *mode* of an evaluation is one of: sequential (written seq), parallel (written par), or oracle (written orc). We let α range over modes. In summary, we have:

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

In addition to evaluating expression, the dynamic semantics also returns cost measures including *raw work* and *raw depth* written by w and d (and variants), and *total work* and *total depth*, written by W and D (and variants). Dynamic semantics is presented in the style of a natural (big-step) semantics and consists of evaluation judgments of the form

$$e \Downarrow^{\alpha} v, (w, d), (\mathcal{W}, \mathcal{D}).$$

This judgment states that evaluating expression e in mode α yields value v resulting in raw-work of w and raw-depth of d and total work of $\mathcal{W}$ and total depth of $\mathcal{D}$.

In the sequential mode, parallel tuples are treated exactly like sequential tuples: evaluating a parallel tuple thus simply contributes 1 to the total work and depth. The depth is in this mode computed as one plus the sum of the depths of the two branches. In the parallel mode, the evaluation of parallel tuples induce an additional constant cost τ. The depth is in this mode computed as one plus the maximum of the depths of the two branches. For the oracle mode, there are two cases. If the parallel tuple is scheduled sequentially, then its cost is only 1 and the depth is computed as the sum of the depth of the branches. If the parallel tuple is scheduled in parallel, then an extra cost τ is involved and the depth is computed as the maximum of the depth of the two branches.

In the 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 of the two both branches is more than κ, then the tuple is executed in parallel and both branches are executed according to the oracle mode. Otherwise, if the raw work of either of the two branches is less than κ, then the tuple is executed sequentially. The mode in which each branch is executed then depends on the work involved in the branch. If a branch contains more than κ units of raw work, then it is executed in oracle mode, otherwise it is scheduled in sequential mode. This switching to sequential mode on small tasks is needed for ensuring that the oracle is not called too often during the execution of a program.

For all expressions other than parallel tuples, the (raw/total) work and the (raw/total) depth are computed by summing up those of the premises and adding one unit. The complete inductive definition of the dynamic cost semantics judgment $e \Downarrow^{\alpha} v, (w, d), (\mathcal{W}, \mathcal{D})$ appears in Figure 3. Note that the rules concerning the oracle mode involve a cost ϕ that will be used to take into account the cost of invoking the oracle. For the time being, consider that ϕ is equal to zero.

4. Analysis

Based on our source language and its cost semantics, we start by showing bounds on the raw/total work and the raw/total depth of computations in different execution modes. For the time being the assumption of a *ideal oracle*, that is, an oracle that always makes perfectly-accurate predictions without any overhead (i.e., $\phi = 0$). Theorem 4.1 shows the relationships between raw-work/raw-depth and total-work/total-depth for the tree possible modes.

Theorem 4.1 (Work and depth) *Consider an expression e such that $e \Downarrow^{\alpha} v, (w, d), (\mathcal{W}, \mathcal{D})$. Assume $\phi = 0$. The following tight bounds can be obtained for total work and total depth, on a machine with P processors where the cost of creating parallel tasks is τ.*

α	*Bound on total work*	*Bound on total depth*
seq	$\mathcal{W} = w$	$\mathcal{D} = d = w$
par	$\mathcal{W} \le (1 + \frac{\tau}{2})\, w$	$\mathcal{D} \le (1 + \tau)\, d$
orc	$\mathcal{W} \le (1 + \frac{\tau}{\kappa+1})\, w$	$\mathcal{D} \le (1 + \mathtt{max}\,(\tau, \kappa))\, d$

Proof The result about 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 (Theorem 4.2 and Theorem 4.3). 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 $\mathcal{W}$ is equal to $n + (n-1)(1+\tau)$. The ratio $\mathcal{W}/w$ can be rewritten as $1+\frac{n\tau}{2n+1}$, which tends to $1+\frac{\tau}{2}$ as n grows.

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

• 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 just over 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 $\mathcal{W}$ is equal to $n(1+\tau) + (n+1)\kappa$. The ratio $\mathcal{W}/w$ 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 depth in oracle mode, in the case $\tau \ge \kappa$, we use the same example as for the work. The raw depth is $n+1$ and the total depth is $n(1+\tau)+\kappa$. The ratio $\mathcal{D}/d$ is equal to $1 + \frac{n\tau+\kappa-1}{n+1}$, which approaches $1+\tau$ as n grows.

• For the depth in oracle mode, in the case $\kappa \ge \tau$, 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 depth is $n+\kappa$ and the total depth is equal to the total work, which is $n+(n+1)\kappa$. The ratio $\mathcal{D}/d$ can be expressed as $1 + \frac{n\kappa}{n+\kappa}$, which approaches $1+\kappa$ as n grows. □

The first important result coming out of the above theorem is that scheduling costs really matter. the total work and total depth can be as much as τ time larger than the raw depth and raw work. For example, consider a program that involves only parallel tuples, in a perfectly balanced tree with n leaves. This program involves about $2n$ raw work, so a sequential run requires $2n$ units of time. A parallel run of this program can take time $\frac{2n\tau}{P}$, which is only $\frac{\tau}{P}$ faster than the sequential execution. In fact, the parallel run can even be slower if the relative cost of creating a parallel task is greater than the number of processors.

The second important result concerns the oracle mode. The theorem suggests that the execution of a computation with an ideal oracle can be as much as $\frac{\kappa}{\tau}$ times faster than that of the parallel mode. This comes at a cost of increasing the depth by a factor of $\frac{\kappa}{\tau}$. In general, it can be very harmful to increase the depth. However, for programs that exhibit a lot of parallel slackness, this is not an issue. Indeed, when $\frac{w}{P}$ is a lot greater than d, we can safely multiply the depth and thereby reduce the scheduling overheads involved. In other words, if parallel slackness is high and κ is not too large, then τd remains small in front of $\frac{w}{P}$, so the depth is not a limitation, however $\frac{\tau}{\kappa}\frac{w}{P}$ becomes much smaller than $\tau\frac{w}{P}$, meaning that the scheduling overheads are dramatically reduced.

Realistic oracles Realizing a perfectly-accurate oracle in practice is impossible. In fact, even predicting execution time with 5 percent accuracy can be tremendously hard. So, we generalize our analysis by allowing by considering an oracle that is allowed to make error by up to a multiplicative factor μ (for example a factor 3). This means that a task that executes sequentially in time w should be predicted by the oracle to take a time between $\frac{w}{3}$ and $3w$. We moreover allow the oracle to take some constant time to provide its answer, and we call this constant ϕ. In what follows, we show that even with such a realistic oracle we are able to reduce the overheads, at leasts for a relatively broad range of programs that we call *regular*. We start with studying the depth and give a result that does not depend on regularity. In the particular case where κ to be large in front of τ and ϕ, this result implies that the depth is no larger than $\mu\kappa$ times the raw depth. (Recall that with the ideal oracle this factor was κ.)

Theorem 4.2 (Depth with realistic oracle)

$$e \Downarrow^{orc} v, (w,d), (\mathcal{W},\mathcal{D}) \quad \Rightarrow \quad \mathcal{D} \le (1+\mathtt{max}\,(\tau,\mu\kappa)+\phi)\, d$$

Proof We write ρ as a shorthand for $1 + \mathtt{max}\,(\tau,\mu\kappa) + \phi$. So, the goal is to prove $\mathcal{D} \le \rho d$. The proof is by induction on the derivation of the hypothesis.

• For a rule with zero premise, we have $\mathcal{D} = d = 1$. Because $\rho \ge 1$, it follows that $\mathcal{D} \le \rho d$.

• For a rule with one premise, we know by induction hypothesis that $\mathcal{D} \le \rho d$. Using again the fact that $\rho \ge 1$, we can deduce the inequality $\mathcal{D}+1 \le \rho(d+1)$.

• For a rule with two premises, we can similarly establish the conclusion $\mathcal{D}_1 + \mathcal{D}_2 + 1 \le \rho(d_1 + d_2 + 1)$ using the induction hypotheses $\mathcal{D}_1 \le \rho d_1$ and $\mathcal{D}_2 \le \rho d_2$.

• Now, consider the case of a parallel tuple. First, assume that the two branches of this tuple is 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 $\mathcal{D}_1 \le \rho d_1$ and $\mathcal{D}_2 \le \rho d_2$ to conclude as follows:

$$\begin{aligned}\mathcal{D} &= \mathtt{max}\,(\mathcal{D}_1,\mathcal{D}_2) + 1 + \tau + \phi \\ &\le \mathtt{max}\,(\rho d_1, \rho d_2) + 1 + \mathtt{max}\,(\tau,\mu\kappa) + \phi \\ &\le \mathtt{max}\,(\rho d_1, \rho d_2) + \rho \\ &\le \rho\,(\mathtt{max}\,(d_1,d_2)+1) \\ &\le \rho d\end{aligned}$$

• 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 $w_1 \le \mu\kappa$ and $w_2 \le \mu\kappa$. Moreover, both branches are executed according to the sequential mode, so we have $\mathcal{D}_1 = w_1$ and $\mathcal{D}_2 = w_2$. It follows that $\mathcal{D}_1 \le \mu\kappa$ and $\mathcal{D}_2 < \mu\kappa$. Below, we also exploit the fact that $\mathtt{max}\,(d_1,d_2) \ge 1$, which comes from the fact that raw depth is at least one unit. We conclude as follows:

$$\begin{aligned}\mathcal{D} &= \mathcal{D}_1 + \mathcal{D}_2 + 1 + \phi \\ &\le \mu\kappa + \mu\kappa + 1 + \phi \\ &\le (1 + \mu\kappa + \phi) * 2 \\ &\le (1 + \mathtt{max}\,(\tau,\mu\kappa) + \phi)\cdot(\mathtt{max}\,(d_1,d_2)+1) \\ &\le \rho d\end{aligned}$$

• It remains to consider the case where one branch is predicted to be smaller than the cutoff 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 $w_2 \le \mu\kappa$. This first branch is thus executed according to the sequential mode, so we have $\mathcal{D}_2 = d_2 = w_2$. It follows that $\mathcal{D}_2 \le \mu\kappa$. For the first branch, which is executed according to the oracle mode, we can exploit the induction hypothesis which is $\mathcal{D}_1 \le \rho d_1$. We conclude as follows:

$$\begin{aligned}\mathcal{D} &= \mathcal{D}_1 + \mathcal{D}_2 + 1 + \phi \\ &\le \rho d_1 + \mu\kappa + 1 + \phi \\ &\le \rho d_1 + (1 + \mathtt{max}\,(\tau,\mu\kappa) + \phi) \\ &\le \rho\,(d_1 + 1) \\ &\le \rho\,(\mathtt{max}\,(d_1,d_2)+1) \\ &\le \rho d\end{aligned}$$

□

This ends our analysis of the depth. Now, let us look at the work. The fact that every call to the oracle can induce a cost ϕ can lead the work to be multiplied by ϕ. For example, consider a program made of a complete tree built using $n-1$ sequential tuples, and where the n leaves are parallel tuples made of atomic values. If n is the number of parallel tuples, the raw work is equal to $(n-1)+n+2n$, and the total work is $(n-1)+n\phi+2n$. So, the ratio $\mathcal{W}/w$ tends to $\phi/4$ when n gets larged. This means that a program executed according to the oracle semantics can get as much as $\phi/4$ times slower than its sequential counterpart.

Fortunately, most programs do not exhibit this pathological behavior. However, to prove a better bound we need to make further assumptions about the structure of the program. It turns out that a sufficient condition for establishing an interesting bound is to ensure that the oracle is never called on too small tasks. This property is hard to capture in a hardware-independent way. However, we can devise a sufficient condition, called *regularity*, which is hardware-independent and ensures the desired property. Intuitively, a program is γ-regular if the ratio between the work involved in a recursive call and the work involved in the next recursive call does not exceed γ. Divide-and-conquer algorithm typically satisfy the regularity condition. We next formalize the definition of regularity.

Definition 4.1 (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 $(|e_1, e_2|)$ if the expression e is involved in the execution of the branch e_i.*

Definition 4.2 (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 very large compared with $\kappa/(\mu\gamma)$ or this branch is dominated by another parallel branch that involves less than γw units of work.*

The condition "w is very large compared with $\kappa/(\mu\gamma)$" is 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. Note 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, it has size at least 3. The next recursive call has raw work $\left\lfloor\frac{n-1}{2}\right\rfloor c$, The ratio between those two values is equal $n/\left\lfloor\frac{n-1}{2}\right\rfloor$ is always less than 3 when $n \geq 3$. So, the algorithm is 3-regular.

The following lemma explains how the regularity assumption can be exploited to ensure that the oracle is never invoked on tasks 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 4.1 (Oracle invocation in regular programs) *Consider a γ-regular program being executed according to oracle semantics. If the oracle is invoked during the execution of the program on an expression e, then e involves at least $\kappa/(\mu\gamma)$ units of work.*

Proof Assume the expression involves w units of work. By the regularity assumption, either w is very large compared with $\kappa/(\mu\gamma)$, in which case the conclusion holds, or it is dominated by a branch that involves that involves w' units of work, with $w' \leq \gamma w$. Because the oracle is being invoked, it means that the evaluation mode is orc. Thus, the dominating branch must have been predicted to be bigger than κ for not being sequentialize. So the dominating branch must be at least of size κ/γ. Combining the inequality $\gamma w \geq w'$ and $w' \geq \kappa/\mu$, we get $w \geq \kappa/(\mu\gamma)$. $\square$

Theorem 4.3 (Work with realistic oracle, with regularity) *Assume $e \Downarrow^{orc} v, (w,d), (\mathcal{W},\mathcal{D})$ where e is γ-regular.*

$$\mathcal{W} \quad \leq \quad (1+\frac{\tau}{\kappa/\mu\ +\ 1}+\frac{\phi}{\kappa/(\mu\gamma)\ +\ 1})\,w$$

Proof Define κ' as a shorthand for κ/μ and κ'' as a shorthand for $\kappa/(\mu\gamma)$. Note that, because $\gamma \geq 1$, we have $\kappa'' \leq \kappa'$. Let x^+ be defined as the value x when x is nonnegative and as zero otherwise. We prove by induction that:

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

This is indeed a strengthened result because we have:

$$\tau\left\lfloor\frac{(w-\kappa')^+}{\kappa'+1}\right\rfloor \;\leq\; \tau\frac{w}{\kappa'+1} \;\leq\; \frac{\tau}{\kappa/\mu+1}\,w$$

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

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

• For a rule with zero premise describing an atomic operation, we have $\mathcal{W}=w=1$, so the conclusion is satisfied.

• For a rule with a single premise, the induction hypothesis is:

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

So, we can easily derive the conclusion:

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

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

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

To conclude, we need to establish the following two mathematical inequality:

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

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 nonnegative, and we are left to check the inequality $w_1 - \kappa' + w_2 - \kappa' \le w_1 + w_2 + 1 - \kappa'$. Third, if w_1 is greater than κ' and w_2 is smaller than κ', then the inequality becomes $(w_1 - \kappa')^+ \le ((w_1 - \kappa') + (w_2 + 1))^+$, which is clearly true. The case $w_1 \ge \kappa'$ 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 $w_1 > \kappa'$ and $w_2 > \kappa'$. In this case, a parallel task is created. Note that, because $\kappa'' \le \kappa'$, we also have $w_1 > \kappa''$ and $w_2 > \kappa''$. So, all the values involved in the following computations are nonnegative. Using the induction hypotheses, we have:

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

• Assume now that the two branches are predicted to be less than the cutoff. This implies $w_1 \le \kappa'$ and $w_2 \le \kappa'$. Both these tasks are executed sequentially, so $\mathcal{W}_1 = w_1$ and $\mathcal{W}_2 = w_2$. By lemma Lemma 4.1, the regularity assumption ensures that we have $w_1 \ge \kappa''$ and $w_2 \ge \kappa''$. 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 $w_1 + w_2 + 1 - \kappa'' \ge \kappa'' + 1$, we know that $\left\lfloor \frac{w_1+w_2+1-\kappa''}{\kappa''+1} \right\rfloor \ge 1$. Therefore:

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

• It remains to consider the case where one branch is predicted to be bigger than the cutoff while the other is predicted to be smaller than the cutoff. For example, assume $w_1 > \kappa'$ and $w_2 \le \kappa'$. The parallel tuple is thus executed as a sequential tuple. The first task is executed in oracle mode, whereas the second task is executed in the sequential mode. For the first task, we can invoke the induction hypothesis $\mathcal{W}_1 \le w_1 + \tau \left\lfloor \frac{w_1-\kappa'}{\kappa'+1} \right\rfloor + \phi \left\lfloor \frac{w_1-\kappa''}{\kappa''+1} \right\rfloor$. For the second task, which is executed sequentially, we have $\mathcal{W}_2 = w_2$. Moreover, the regularity hypothesis gives us $w_2 \ge \kappa''$. Hence, we have $\left\lfloor \frac{w_2+1}{\kappa''+1} \right\rfloor \ge 1$. We conclude as follows:

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

□

Observe that the above proof does not exploit the regularity assumption directly but only through the application of Lemma 4.1. In fact, the proof treats the value $\kappa/(\mu\gamma)$, called κ'' in the proof, as an abstract value with the only assumption that it is smaller than κ/μ. So, even though we have been using regularity as a sufficient condition for ensuring that the oracle does not get invoked on small expressions, there might be other sufficient conditions that could be used. For example, if we observe that the oracle never gets called in practice on tasks that take more than a time t to execute (note that if the oracle satisfies its specification, we should have $t \ge \kappa/(\mu\gamma)$), then we can replace the factor $\frac{\phi}{\kappa/(\mu\gamma)+1}$ from the statement of Theorem 4.3 with the smaller factor $\frac{\phi}{t+1}$.

Using these bounds on work and depth, we bound the time for executing a γ-regular parallel program with a realistic oracle (Theorem 4.4). This bound relies directly on our generalization of Brent's theorem, which as the original theorem assumes a greedy scheduler that can perform load balancing without any overheads; this is of course unrealistic but not too far away from it as it turns out (Section 5).

Theorem 4.4 (Execution time with realistic oracle) *Assume an oracle that costs ϕ and makes error by a factor not exceeding μ. Then, the execution time of a parallel γ-regular program on a machine with P processors under the oracle semantics with a greedy scheduler is*

$$\left(1 + \frac{\tau}{\kappa/\mu+1} + \frac{\phi}{\kappa/(\mu\gamma)+1}\right)\frac{w}{P} \;+\; (1 + \mathtt{max}\,(\tau, \kappa\mu) + \phi)\, d$$

Proof By the generalized version of Brent's theorem (Theorem 2.3), using the bounds provided by Theorem 4.3 and Theorem 4.2. $\square$

Choosing a value for the cutoff Assume that κ is relatively large, in the sense that $\kappa/(\mu\gamma) \gg 1$ and $\kappa \gg \tau$ and $\kappa \gg \phi$. Then, we can rewrite the bound of the theorem in a simpler way:

$$\left(1 + \frac{\mu(\tau+\gamma\phi)}{\kappa}\right)\frac{w}{P} \;+\; \kappa\mu d$$

It now appears clearly that a small value for κ increases the total work whereas a large value for κ increases the total depth. We are thus interested in computing the optimal value for κ, that is, the value of κ that minimizes the above expression. To that end, we compute the derivative of this expression with respect to κ. We get:

$$-\frac{\mu(\tau+\gamma\phi)}{\kappa^2}\frac{w}{P} + \mu d$$

The optimal value for κ is that for which the derivative is zero. This happens for:

$$\kappa = \sqrt{\tau+\gamma\phi}\cdot\sqrt{\frac{w}{Pd}}$$

This suggests that κ should never exceed a constant factor times the square root of the *degree of parallel slackness*, which is the ratio between $\frac{w}{P}$ and d. (Intuitively, the parallel slackness assumption asserts that this ratio is "large", however to compute the optimal value for κ we quantify how large it is.) The fact that κ cannot be taken very large, because it is bounded by a square root, is not necessarily a problem. Indeed, we would be satisfied if the ratio $\frac{\mu(\tau+\gamma\phi)}{\kappa}$ was equal to any value less than 10 percent, because this would ensure that the scheduling overheads do not exceed 10 percent of the total work.

We can now ask the question the other way around: what is the minimal degree of parallel slackness that is required for ensuring that the ratio $\frac{\mu(\tau+\gamma\phi)}{\kappa}$ is less than r, for some r? To that end, we need to satisfy both $(\frac{\mu(\tau+\gamma\phi)}{r})^2 < \kappa^2$ and $\kappa^2 < (\tau+\gamma\phi)\frac{w}{Pd}$. Eliminating κ^2, we obtain the inequality:

$$\frac{w}{Pd} > \frac{\mu^2(\tau+\gamma\phi)}{r^2}$$

This result implies that, in order to be able to bound the scheduling overheads by r, parallel slackness needs to exceed a constant factor times $1/r^2$. For example, we can take $r = 10\%$. For program with small depth (e.g., logarithmic depth), this result implies that the scheduling overheads do not exceed a factor r of the raw work when w is larger than $r^2 * \mu^2(\tau+\gamma\phi)Pd$. Thus, by taking $\kappa = \frac{\mu(\tau+\gamma\phi)}{r}$, we ensure that, for any program with small depth, the scheduling overheads do not exceed a fraction r of the raw work whenever the input data given to the program is large enough.

Improving the bound on total depth The bound that we have established concerning the total depth of an expression did not use the μ-regularity assumption. This assumption alone does not suffice to obtain a better bound than that of Theorem 4.2. However, if we make further assumptions we may improve the bound.

Consider for example a simple recursive function, which does not make calls to other parallel functions, and whose body involves a single parallel tuple. Now, consider an execution path in the computation DAG of a run of this program. All the parallel tuples at the beginning of this path contain a lot of work so they are scheduled in parallel. Only the parallel tuples near the end of the path may get sequentialized.

More precisely, as long as the raw work involved in a parallel tuple exceeds $\gamma\mu\kappa$ units of raw work, we know by the regularity assumption that both branches of this tuple involve more than $\mu\kappa$ work. So, the oracle must predict those branches to involve more than κ work, leading the parallel tuple to be executed in parallel. This observation implies that we can never sequentialize more than $\gamma\mu\kappa$ units of work on a given execution path.

In summary, for simple recursive functions containing only one parallel tuple, we obtain the following bound on the total depth:

$$\mathcal{D} \;\leq\; (1+\tau+\phi)\, d \;+\; \mu\gamma\kappa$$

This bound significantly improves that of Theorem 4.2, because κ no longer appears multiplicatively in front of the raw depth, but only as an extra additive factor. We leave to future work a formal investigation of the class of functions for which the factor κ appears only additively and not multiplicatively in the total depth.

5. Oracle scheduling with complexity functions

The original theorem of Brent as well as our generalization assume a greedy scheduler that can find available work (parallel tasks to execute) immediately with no overhead. This is unrealistic of course in a literal sense but many schedulers can achieve similar bounds asymptotically for a reasonably broad class of computations. For example, a work-stealing scheduler can execute a fully-strict computations with $\mathcal{W}$ work and $\mathcal{D}$ depth on P processors in $O(\mathcal{W}/P + \mathcal{D})$ expected time [10]. The class of fully-strict computations (*i.e.*, series-parallel computations) include all fork-join programs, nested-parallel computations, and specifically computations with parallel tuples, our focus here.

Since the oracle semantics that we have presented makes no assumptions about a scheduler (it simply creates parallel tasks), the created parallel tasks can be scheduled by using a scheduler, *e.g.*, a work-stealing scheduler, to execute them on a parallel machine. The oracle semantics itself can be realized by using a (ϕ, μ)*-estimator* that requires ϕ time to estimate actual run-time of parallel tasks within a factor of no more than μ. We refer to the combination of an estimator with a parallel scheduler as an (ϕ, μ)*-oracle-scheduler*.

In the rest of this section, we describe how to obtain an estimator suitable to be used as part of an oracle scheduler by using asymptotic complexity annotations entered by the user. We use an implementation of this estimator combined with a work-stealing scheduler in our experiments (Sections 6 and 7).

Asymptotic complexity annotations. To obtain an effective estimator as part of an oracle, we rely on asymptotic complexity annotations decorating function calls. For the purpose of this paper, we assume that these annotations are entered by the programmer but in some cases they can also be approximated by some other technique such as a static analysis (e.g., [24]). The basic idea behind our approach is to combine these combine annotations with judiciously timed runtime measurements to determine efficiently the constant factors that are hidden by the complexity functions. Here, we describe a compilation strategy for complexity annotations to implement this idea.

To take advantage of complexity functions, we restrict parallel tuples of the form $(|e_1, e_2|)$, to that the expressions are function applications, *i.e.*, $(|f_1\, v_1, f_2\, v_2|)$. Note that this syntactic restriction does not reduce 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. We require user to provide complexity function for every defined function and our compilation technique propagates these annotations to the applications sites.

We write "$\texttt{fun}\, f.x.e_b\ [e_c]$" to denote a function "$\texttt{fun}\, f.x.e_b$" for which the complexity 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 an integer value. For our bounds to apply, complexity expressions should require constant time to evaluate.

To predict accurately actual execution times of a parallel tuple, the oracle also needs to determine the constant factors hidden in the complexity functions. We estimate these constants factors by collecting statistical information on execution times of parallel tuples. Our compilation strategy introduce the code for collecting this statistical information. Since we are interested only in the sequential run-time of parallel tasks, we only need to measure parallel tasks that are sequentially executed; this both decreases the overhead of measurements and increases their accuracy.

We use a "Constant-Estimator Data Structure" (CED) to collect the statistical information needed to determine the constant factors. The first pass of our compiler simply in associating a CED with every function. For example, if the source code contains a function of the form "$\texttt{fun}\, f.x.e_b\ [e_c]$", then our compiler allocates a CED specific to that function definition. For example, if the variable r refers to this particular CED, then the function becomes annotated with r. We then write it "$\texttt{fun}\, f.x.e_b\ [e_c|r]$". The second pass uses the CED to estimate the constant factors hidden in the complexity function.

For the time being, we leave the implementation of the constant-estimator datastructure (CED) abstract and only describe its interface, which comprises three functions:

$$\begin{array}{ll}\textsf{type ced} & \\ \textsf{val}\ \mathit{ced\_initialize} & : \textsf{unit} \rightarrow \textsf{ced} \\ \textsf{val}\ \mathit{ced\_estimate} & : \textsf{ced} \times \textsf{int} \rightarrow \textsf{float} \\ \textsf{val}\ \mathit{ced\_measured} & : \textsf{ced} \times \textsf{int} \times \textsf{float} \rightarrow \textsf{unit}\end{array}$$

The function $\mathit{ced\_initialize}$ allocates a new CED. The function $\mathit{ced\_estimate}$ takes a pointer to a CED and the value returned by a call to the user-provided complexity function, and then returns the amount of time predicted for the execution of the function. The function $\mathit{ced\_measured}$ is used to report to a CED the time actually measured by the execution of the function. It takes as argument a pointer to the CED, the value returned by the complexity function and the time measured, and it then updates the data stored internally in the CED.

In summary, to predict the execution time of a function call of the form "$\texttt{fun}\, f.x.e_b\ [e_c|r]$" on an argument v, the oracle first executes the complexity expression $e_c[v/x]$, which produces an integer value, call it m. The prediction of the oracle is then obtained by evaluating the expression $\mathit{ced\_estimate}(r, m)$. Typically, this would compute the product of m by the current estimate of the constant factor stored in r.

Compilation. We now describe how to translate a source code with annotated functions of the form "$\texttt{fun}\, f.x.e_b\ [e_c|r]$" into a standard piece of code by replacing these annotations to operations on CEDs. This translation performs three tasks. First, it packs every function with its CED and its complexity function. Second, it adds code to make runtime decisions about the scheduling of parallel tuples. Third, it instruments the code so as to measure execution time for branches of parallel tuples that are executed according to the sequential semantics.

To describe the translation implementing oracle scheduling, we write $[\![v]\!]$ the translation of a value v, and we write $[\![e]\!]^{\alpha}$ the translation of the expression e according to the semantics α, which can be either seq or orc. Observe that the translation of values, contrary to the translation of expressions, does not depend on the mode. The compilation scheme appears in Figure 4 and it is describe next.

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

Figure 4. Translation implementing oracle scheduling

Note that the figure makes use of triples, quadruples, projections, sequence, if-then-else statements, and unit value; these constructions can all be easily defined in our core programming language. An annotated function of the form "$\texttt{fun}\ f.x.e_b\ [e_c|r]$" is translated to a quadruple of the form

$$(r,\ \texttt{fun}\ \_.x.e_c,\ \texttt{fun}\ f.x.\llbracket e_b \rrbracket^{\mathsf{seq}},\ \texttt{fun}\ f.x.\llbracket e_b \rrbracket^{\mathsf{orc}})$$

which is made of the CED associated with the function, the complexity function associated with the function, the sequential-mode version of the function, and the oracle-mode version of the function. An application of a function f to an argument v is mapped to the application of either the third or the fourth projection of f to the value v, depending on whether the function should be executed in the sequential mode or in the oracle mode. A parallel tuple is turned into a simple tuple if it is executed in the sequential mode, otherwise it is scheduled using our oracle-based scheduling policy. This policy is implemented with help of a function called Oracle, which is described next. The translation of other language constructs is entirely structural.

The meta-function Oracle, shown in Figure 5, describes the template of the code generated for preparing the execution of a parallel tuple. Oracle 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 cutoff, and a continuation k to execute this application. On the one hand, if the application is predicted to take more time than the cutoff (in which case b is true), then the continuation k corresponds to the oracle-semantics version of the function f. On the other hand, if the application is predicated to take less time than the cutoff (in which case b is false), then the continuation k corresponds to 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 CED so that it can take it into account for future predictions. The reporting is performed by the auxiliary meta-function $\mathit{MeasuredRun}$, whose code also appears in Figure 5. This completes the description of our translation.

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 the body of other functions. In practice, the code grows only reasonably since functions are rarely deeply nested. If code duplication was a problem, then one could either use flattening to eliminate deep nesting of local functions, or make functions take the parameter α as extra argument.

6. Implementation

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 [25], which is a strict functional language. Our Caml dialect corresponds to the core Caml language extended with syntax for parallel pairs and com-

```
$Oracle\,(f, v) \;\equiv$
    let $r \;=\;$ proj$^1\, f$ in
    let $m \;=\;$ proj$^2\, f\, v$ in
    let $b \;=\; ced\_estimate(r, m) > \kappa$ in
    let $k_{\mathsf{seq}}\,() \;=\;$ proj$^3\, f\, v$ in
    let $k'_{\mathsf{seq}}\,() \;=\; MeasuredRun(r, m, k_{\mathsf{seq}})$ in
    let $k_{\mathsf{orc}}\,() \;=\;$ proj$^4\, f\, v$ in
    let $k \;=\;$ if $b$ then $k_{\mathsf{orc}}$ else $k'_{\mathsf{seq}}$ in
    $(b, k)$
$MeasuredRun\,(r, m, k) \;\equiv$
    let $t \;=\; get\_time\,()$ in
    let $v \;=\; k\,()$ in
    let $t' \;=\; get\_time\,()$ in
    $ced\_measured\,(r, m, (t' - t))$;
    $v$
```

Figure 5. Auxiliary meta-functions used for compilation

```
type tree =
  | Leaf of int
  | Node of int * tree * tree

let size = function
  | Leaf _ -> 1
  | Size (s,_,_) -> s

let rec sum t = Oracle.complexity (size t);
  match t with
  | Leaf n -> n
  | Node (size,t1,t2) ->
     let (n1,n2) = (| sum t1, sum t2 |) in
     n1 + n2
```

Figure 6. An example parallel program.

plexity annotations. Figure 6 shows a program implemented in our Caml dialect. This recursive program traverse a binary tree to compute the sum of the values stored in the leaves.

We use the Caml type checker to obtain a typed syntax tree, on which we perform the oracle-scheduling translation defined in Figure 4. We then produce code in the syntax of Parallel ML (PML) [15], 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 Manticore runtime system provides a parallel, generational garbage collector that is crucial for scaling to more than four processors, because functional programs, such as the ones we consider, often involve heavy garbage-collection loads. Further details on Manticore can be found elsewhere [14]. In the rest of this section, we explain how we compute the constant factors, and we also give a high-level description of the particular work-stealing scheduler on top of which we are building the implementation of our oracle scheduler.

Runtime estimation of constants. The goal of the oracle is to make relatively accurate execution time predictions at little cost. Our approach to implementing the oracle consists in evaluating a user-provided asymptotic complexity function, and then multiplying the result by an appropriate constant factor. Every function has its own constant factor, and the value for this constant factor is stored in the constant-estimate data-structure (CED). In this section, we discuss the pratical implementation of the evaluation of constant factors.

In order for the measurement of constant to be lightweight, we simply compute average values of the constant. The constant might evolve through 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 all history, but instead maintain a moving average, that is, an average of the values gathered across a certain number of runs.

Maintaining averages is not entirely straightforward. One the one hand, storing data 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 CED contains the estimated value for the constant that is read by all the processors when they want to 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 into 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 use a special treatment for the initialization of the constants: for its first few measures, a processor always report immediately its current average to the shared memory cell. This ensures a fast propagation of the information gathered from the first runs, so as to quickly improve the accuracy of the predictions.

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 the detection of these outliers to be relatively easy because the time measured are at least one or two orders of magnitude greater than the cutoff value. Third, the default system function that reports the time is only accurate by one microsecond. This is good enough when the cutoff is greater than 10 microseconds. However, if one were to aim for a smaller cutoff, which could be useful for programs exhibiting only a limited amount of parallelism, then it would be required to use more accurate techniques, for example using the specific processor instructions for counting the number of processor cycles.

Work stealing. We implement our oracle scheme on top of the work stealing scheduler [10]. In this section we outline the particular implementation of work stealing that we selected from the Manticore system. Our purpose is to understand what exactly contributes to the scheduling cost τ in our system.

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 tasks represented as thunks. Processors treat their own deques like call stacks. When a processor starts to evaluate a parallel-pair expression, it creates a task for the second subexpression of the pair and pushes the task onto the bottom of the deque. Processors that have no work left try to *steal* tasks from others. More precisely, they repeatedly select a random processor and try to pop a task from this processor's deque.

Manticore's implementation of work stealing [31] adopts a code-specialization scheme, called clone translation, taken from Cilk-5's implementation [18].[1] 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 doing the evaluation 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 of between clone translation and our oracle translation (Figure 4) 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 spawns subtasks.

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 $O(Pd)$ steals occur during the evaluation [10]. So, for programs that exhibit parallel slackness ($w \gg Pd$), 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 thus correspond to the cost τ. A fast clone needs to packages a task, push it onto the deque and later pop it from the deque. So, a fast clone is not quite as fast than the corresponding sequential code. The exact slowdown depend on the implementation, but in our case we have observed that a fast clone is 3 to 5 times slower than a simple function call.

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

7. 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 two machines with different architectures. 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 up to sixteen processors.

Machines. Our AMD machine has four quad-core AMD Opteron 8380 processors running at 2.5GHz. Each core has 64Kb each of L1 instruction and data cache, a 512Kb L2 cache. Each processor has a 6Mb L3 cache that is shared with the four cores of the processor. The system has 32Gb of RAM and runs Debian Linux (kernel version 2.6.31.6-amd64).

Our Intel machine has four eight-core Intel Xeon X7550 processors running at 2.0GHz. Each processor has 32Kb each of L1 instruction and data cache and 256 Kb of L2 cache. Each processor has an 18Mb L3 cache that is shared by all eight cores. The system has 1Tb of RAM and runs Debian Linux (kernel version 2.6.32.22.1.amd64-smp). For uniformity, we consider results from just sixteen out of the thirty-two cores of the Intel machine.

Measuring scheduling costs. We report estimates of the scheduling overheads for each of our test machines. To estimate, we use a synthetic benchmark expression e whose evaluation sums integers between zero and 30 millions using a parallel divide-and-conquer computation. We choose 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 evaluated 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 use call the oracle to predict the time and measure the time. 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 task 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 and making a time measure by computing $\frac{w_o - w_s}{m}$, where m is the number of times the oracle is invoked. Our measures are as follows.

Machine	c_w	τ (μs)	ϕ (μs)
AMD	4.86	0.09	0.18
Intel	3.90	0.18	0.94

The first column 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. Column two indicates that the cost of creating parallel task τ is significant, taking roughly between 200 and 350 processor cycles. The last column suggests that the oracle cost ϕ is of the same order of magnitude (ϕ is 2 to 5 times larger than τ).

To determine a value for κ, we use the formula $\frac{\mu(\tau+\gamma\phi)}{r}$ from §4. Recall that r is the targetted overhead for scheduling costs. We aim for $r = 10\%$. Our oracle appears to be always accurate within a factor 2, so we take $\mu = 2$. Our benchmark programs are fairly regular, so we take $\gamma = 3$. We then use the values for τ and ϕ specific to the machine and evaluate the formula $\frac{\mu(\tau+\gamma\phi)}{r}$. We obtain $13\mu s$ for the AMD machine $60\mu s$ for the Intel machine. However, we were not able to use a cutoff as small as $13\mu s$ because the time function that we are using is only accurate up to $1\mu s$. For this reason, we doubled the value to $26\mu s$. (One possibility to achieve greater accuracy would be to use architecture-specific registers that are able to report on the number of processor cycles involved in the execution of a task.)

In our experiments, wehave used $\kappa = 26\mu s$ on the AMD machine and $\kappa = 61\mu s$ on the Intel machine.

Benchmarks. We use 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 [5]. 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 algorithm that calculates the gravitational forces between n particles as they move through 2-d space [2]. The Barnes-Hut computation consists of two phases. In the first, a quadtree is constructed from the sequence of particles. Using this tree, the second phase computes this tree to accelerate the computation of the gravitational force for each of the n particles. 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 [30]. The program is adapted from a Data-Parallel Haskell program [23]. 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 nonzero values.

The DMM benchmark multiplies two dense, square $n \times n$ matrices using the recursive divide-and-conquer algorithm of Frens and Wise [17]. We have recursion go down to scalar elements. The algorithm runs with $O(n^3)$ raw work and $O(\log n)$ raw depth. We select $n = 512$.

Implementing complexity functions. Our aim is to make complexity functions fast, ideally constant time, so that we can keep oracle costs low. But observe that, in order to complete in constant time, the complexity function needs access to the input size in constant time. For four of our benchmark programs, no modifications to the algorithm is necessary, because the relevant data structures are already decorated with sufficient size information. The only one for which we make special provisions is SMVM. The issue concerns a subproblem of SMVM called segmented sums [8]. In segmented sums, our input is an array of arrays of scalars, *e.g.*,

$$[[8, 3, 9], [2], [3, 1][5]]$$

whose underlying representation is in segmented format. The segmented format consists of a pair of arrays, where the first array contains all the elements of the subarrays and second contains the lengths of the subarrays.

$$([8, 3, 9, 2, 3, 1, 5], [3, 1, 2, 1])$$

The second array is called the segment descriptor. The objective is to compute the sum of each subarray,

$$[20, 2, 4, 5],$$

There are two sources of parallelism in segmented sums: (1) within the summation of each subarray and (2) between different subarray sums. We use divide-and-conquer algorithms to solve each case. In the first case, our algorithm is just an array summation, and correspondingly, the complexity function for which is straightforward to compute in constant time from the segment descriptor. The second case is where we make the special provisions. For this case, we use a parallel array-map algorithm to compute all the subarray sums in parallel. The issue is that the complexity of performing a group of subarray sums is proportional to the sum of the sizes of those subarrays. So, to obtain this size information in constant time, we modify our segmented-array representation slightly so that we store a cached tree of subarray sizes rather than just a flat array of subarray sizes.

$$\begin{tikzcd}
([8,3,9,2,3,1,4,5], & & & 7 \arrow[dl, no head] \arrow[dr, no head] & & &) \\
& & 4 \arrow[dl, no head] \arrow[d, no head] & & 3 \arrow[d, no head] \arrow[dr, no head] & & \\
& 3 & 1 & & 2 & 1 &
\end{tikzcd}$$

To summarize, in order to write a constant-time complexity function, we change the existing SMVM program to use a tree data structure, where originally there was an array data structure. Building the tree can be done in parallel, and the cost of building can be amortized away by reusing the sparse matrix multiple times, as is typically done in iterative solvers.

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 scheduling 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 sixteen 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 sixteen 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 (with an average of 1.07), indicating that the scheduling 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 bewteen 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 processor P. Observe that on the Intel machine there is super-linear scaling, but not on the AMD machine. We attribute this behavior to cache effects. The curves show that our oracle implementation generally scales well up to sixteen processors.

There is one exception, which is the quickhull benchmark on the AMD machine. For this benchmark, the curve tails off after reaching twelve processors. We need to conduct further experiments to understand the cause of this tailing off, which is probably due to a lack of parallelism in the program. Notice, however, that our scheduler does not fall below work stealing.

8. Related work

Cutting off excess parallelism. Early work on granularity control for functional programs focuses on using list size to determine when to cutoff parallelism [22]. This approach is limited because it assumes linear time for parallel subcomputations.

Lopez *et al.* study the granularity problem for logic programs [26]. Their approach uses complexity functions to guide parallelism. On the surface, this is similar to our oracle approach, except that their cost estimators do not take use 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.

Reducing per-task costs. One approach to the granularity problem is to focus on reducing the the costs associated with tasks, rather than limiting how many tasks get created. This approach is taken by implementations of work stealing with lazy task creation [13, 18, 21, 27, 31, 33]. In lazy task 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 task creation, there is still a non-negligable scheduling cost for each implicit thread.

Lazy Binary Splitting (LBS) is an improvement to lazy task creation that applies to parallel loops [36]. The crucial optimization comes from extending the representation of a task so that multiple loop iterations can be packed into a single task. This representation enables the scheduler to both avoid creating closures and executing deque operations for most iterations. A limitation of LBS is that it addresses only parallel loops whose iteration space is over integers. Lazy Tree Splitting (LTS) generalizes LBS to handle parallel aggregate operations that produce and consume trees, such as map and reduce [3]. LTS is limited, however, by the fact that it requires a special cursor data structure to be defined for each tree data structure.

Amortizing per-task costs. Feitelson *et al.* study the granularity problem in the setting of distributed computing [1], where the crucial issue is how to minimize the cost of interprocessor communication. In their setting, the granularity problem is modeled as a staging problem, in which there

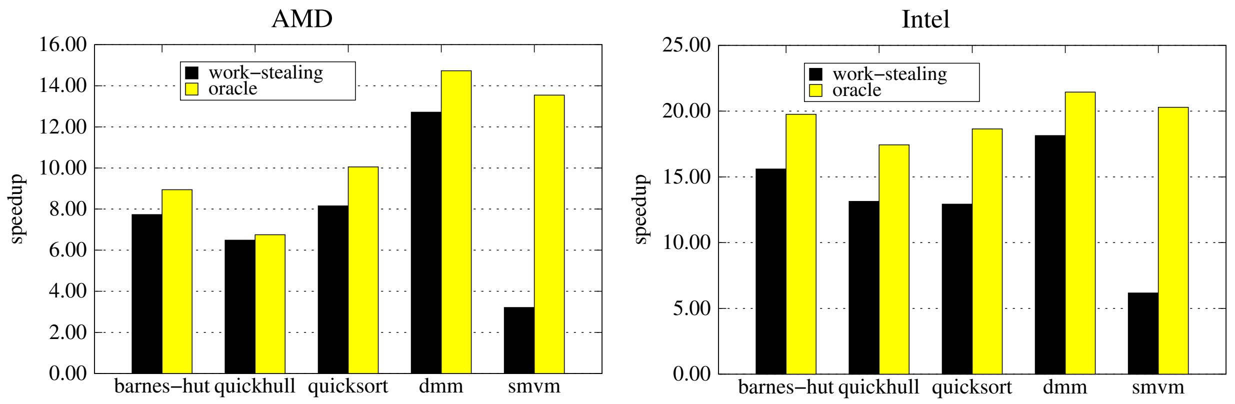

Figure 7. Comparison of the speedup on sixteen processors. Higher bars are better.

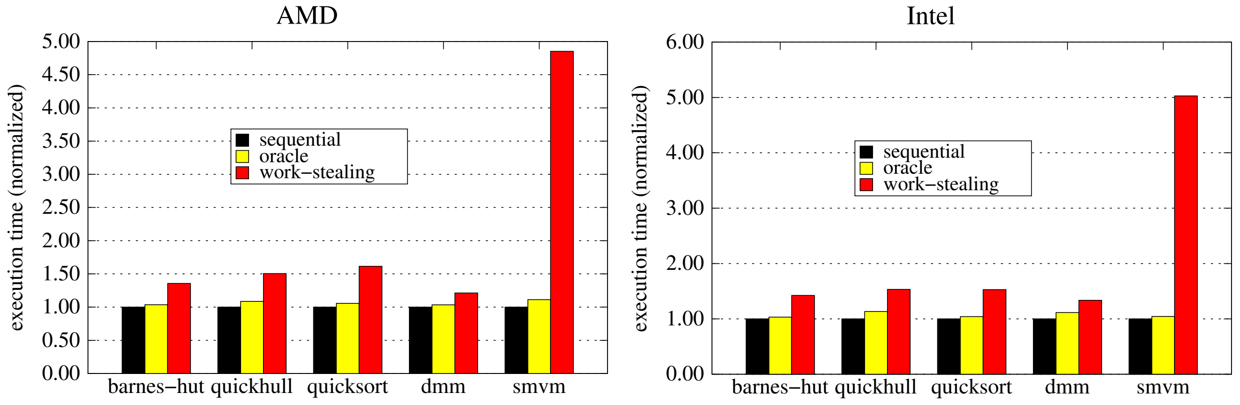

Figure 8. Comparison of execution times (normalized) on a single processor. Lower bars are better.

are two stages. The first stage consists of a set of processor-local task pools and the second stage consists of a global task pool. Moving a task to the global task pool requires inter-processor communication. The crucial decision is how often each processor should promote tasks from its local task pool to the global task 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 Feitelson *et al.* is based on an online algorithm called CG. In this approach, it is assumed that the cost of moving a task to the global task pool is an integer constant, called g. The basic idea is to use amortization to reduce the scheduling total cost of moving tasks to the global task pool. In particular, for each task that is moved to the global task pool, CG ensures that there are at least $g + 1$ tasks added to the local task pool. Narlikar describes a similar approach based on an algorithm called DFDeques [28]. 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-negligable cost for each implicit thread.

Flattening and fusion. Flattening is a well-known program transformation for nested parallel languages [9]. Implementations of flattening include NESL [7] and Data Parallel Haskell [29]. 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. 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 [6], as a consequence of the trans-

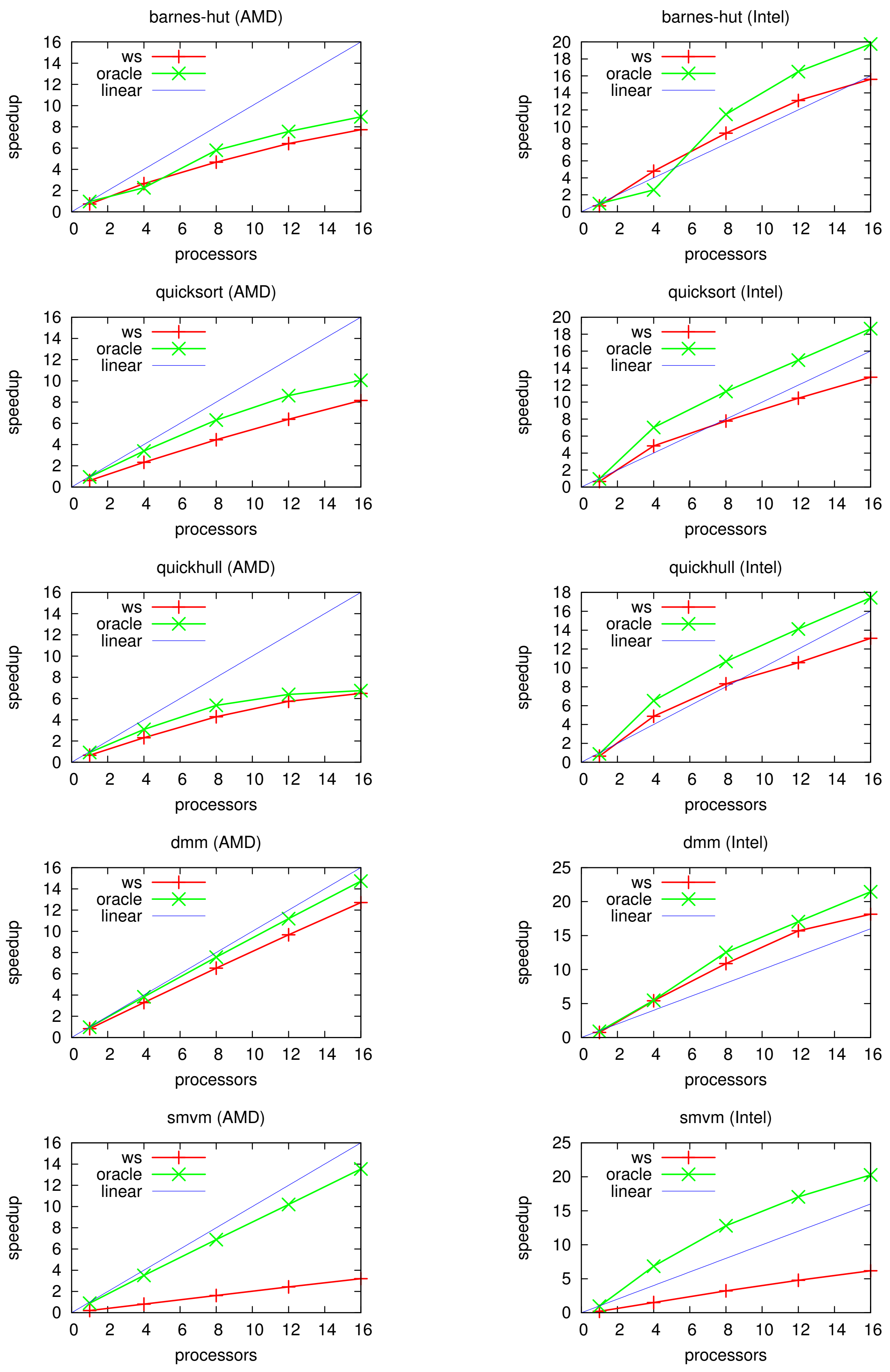

Figure 9. Comparison between work stealing and oracle.

formation making changes to data structures defined in the program. Our transformation involves no such changes.

The NESL [7] and Data Parallel Haskell [29] 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.

Cost Semantics. To give an accurate accounting of task-creation of overheads in implicitly parallel languages we use a cost semantics, where evaluation steps (derivation rules) are decorated with work and depth information or "costs". This information can then be used to 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-depth properties, some of which also make precise the relationship between cost semantics and the standard directed-acyclic-graph models [4, 6, 35]. The idea of instrumenting evaluations to generate cost information goes back to the early 90s [32, 34].

9. Future work

In this work, we have experimented our idea of oracle scheduling by implementing it on top of a work-stealing scheduler. It would be interesting to also investigate the use of other schedulers, in particular schedulers based on a shared queue. We could also try to apply this technique to a distributed setting, where spawning parallel tasks not only requires to migrate code but also to migrate data, which can be very costly.

In our implementation, we have fixed κ to be a constant value, but we could try to dynamically adjust it. However, there is a major difficulty: obtaining accurate estimates for the raw depth is much harder than for the raw work. Indeed, for raw work it suffices to measure the time taken by a sequential execution of a task. There is no such easy way to measure the raw depth of a program.

To realize the oracle, we have assumed that the programmer would provide complexity functions explicitly. This seems to be a reasonable assumption in general. However, for particular domains of application it is possible to use static analysis techniques to infer those complexity bounds. We could also use dynamic analysis, in particular machine learning techniques, for automatically guessing the shape of the complexity function.

We have also assumed that the complexity functions could be implemented in constant time. To achieve this in the context of working with functional data structures, one need to store size information in data structures. Fortunately, this can be achieved at little cost if the compiler provides specialized support for this, as done for example in [20]. This would make it possible to store sizes with little memory overhead and with little programmer intervention onto the source code.

References

[1] Gad Aharoni, Dror G. Feitelson, and Amnon Barak. A run-time algorithm for managing the granularity of parallel functional programs. *Journal of Functional Programming*, 2:387–405, 1992.

[2] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324:446–449, December 1986.

[3] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Lazy tree splitting. In *ICFP 2010*, pages 93–104, New York, NY, USA, September 2010. ACM Press.

[4] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *FPCA '95: Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 226–237, 1995.

[5] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the Symposium on Functional Programming and Computer Architecture*, pages 226–237, June 1995.

[6] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, pages 213–225. ACM, 1996.

[7] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994.

[8] Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, August 1993.

[9] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, February 1990.

[10] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. *Annual IEEE Symposium on Foundations of Computer Science*, 0:356–368, 1994.

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 1995.

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

[13] Marc Feeley. A message passing implementation of lazy task creation. In *Proceedings of the US/Japan Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, pages 94–107, London, UK, 1993. Springer-Verlag.

[14] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 241–252, New York, NY, USA, 2008. ACM.

[15] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in manticore. *Journal of Functional Programming*, FirstView:1–40, 2011.

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

[17] Jeremy D. Frens and David S. Wise. 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, pages 206–216, New York, NY, USA, 1997. ACM.

[18] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[19] Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985.

[20] M. Hermenegildo and P. López García. Efficient term size computation for granularity control. In Leon Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 647–664, Cambridge, June 13–18 1995. MIT Press.

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

[22] Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. 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, pages 79–90, 1994.

[23] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 223–236, 2010.

[25] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*, 2005.

[26] P. Lopez, M. Hermenegildo, and S. Debray. A methodology for granularity-based control of parallelism in logic programs. *Journam of Symbolic Computation*, 21:715–734, June 1996.

[27] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. 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*, pages 185–197, New York, New York, USA, June 1990. ACM Press.

[28] Girija Jayant Narlikar. *Space-efficient scheduling for parallel, multithreaded computations*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1999. AAI9950028.

[29] Simon L. Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *APLAS*, page 138, 2008.

[30] H. C. Plummer. On the problem of distribution in globular star clusters. *Monthly Notices of the Royal Astronomical Society*, 71:460–470, March 1911.

[31] Mike Rainey. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. PhD thesis, University of Chicago, August 2010. Available from `http://manticore.cs.uchicago.edu`.

[32] Mads Rosendahl. Automatic complexity analysis. In *FPCA '89: Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, 1989.

[33] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 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, pages 311–322, New York, NY, USA, 2010. ACM.

[34] David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, University of London, Imperial College, September 1990.

[35] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008.

[36] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *Symposium on the Principles and Practices of Parallel Programming*, New York, NY, USA, January 2010. ACM Press.

[37] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.