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 task-creation overheads can be overwhelming. If large tasks are executed sequentially, processors may spin idle. This granularity problem, however well known, is not well understood: broadly applicable solutions remain elusive.

We propose techniques for controlling granularity in implicitly parallel programming languages. Using a cost semantics for a general-purpose language in the style of the lambda calculus with support for parallelism, we show that task-creation overheads can indeed slow down parallel execution by a multiplicative factor. We then propose *oracle scheduling*, a technique for reducing these overheads, which bases granularity decisions on estimates of task-execution times. We prove that, for a class of computations, oracle scheduling can reduce task creation overheads to a small fraction of the work without adversely affecting available parallelism, thereby leading to efficient parallel executions.

We realize oracle scheduling in practice by a combination of static and dynamic techniques. We require the programmer to provide the asymptotic complexity of every function and use run-time profiling to determine the implicit, architecture-specific constant factors. In our experiments, we were able to reduce overheads of parallelism down to between 3 and 13 percent, while achieving 6- to 10-fold speedups.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Algorithms, Experimentation, Languages

Keywords Scheduling, Granularity Control, Work Stealing

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 as 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 [12], Manticore [16, 17], Multilisp [22], and NESL [8]. These languages enable the programmer to express opportunities for parallelism via language constructs, *e.g.*, parallel sequences, parallel arrays, and parallel tuples. This implicit approach enables a declarative programming style by delegating the task of utilizing the parallelism exposed by the program to the compiler and the run-time system. As an implicit parallel program executes, it exposes opportunities for parallelism (as indicated by the parallel constructs) and the language run-time system creates parallel tasks as needed. To execute parallel tasks efficiently, implicit programming languages use a scheduler to load balance, *i.e.*, distribute parallel tasks among processors. Various scheduling techniques and practical schedulers have been developed, including work-stealing schedulers [1, 3, 11] and depth-first-search schedulers [7].

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 tasks to execute sequentially. This problem, often referred to as the *granularity problem*, is important because creating a parallel task requires additional overhead. If the task granularity is not handled effectively, task-creation overheads can easily obliterate the benefit of parallelism.

Many parallel programs are characterized by parallel slackness [41], a property which indicates that the program exposes many more opportunities for parallelism than the number of available processors. In such programs, effective granularity control is crucial because the program typically creates many small tasks, thereby ensuring significant scheduling overhead.

No known broadly applicable solution to the granularity problem exists. Theoretical analyses often ignore task-creation overheads, yielding 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. Since the running time of a task depends on the hardware, such manual control of granularity is difficult and bound to yield suboptimal results and/or non-portable code [40].

In this paper, we propose theoretical and practical techniques for the granularity problem in implicit parallel-programming languages. Our results include theorems that characterize how parallel run time is affected by task-creation overheads, which we show to be significant (Section 3). To reduce these overheads, we consider a granularity control technique that relies on an oracle for determining the run-time of parallel tasks (Section 2). 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 (Section 3). Based on this result, we describe how oracles can be realized in practice; we call this technique *oracle scheduling* because it relies on an oracle to estimate task sizes and because it can be used in conjunction with practically any other scheduler (Section 4). 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 4). We present an implementation and evaluation of the proposed approach by extending a subset of the Caml language (Sections 5 and 6).

Brent's theorem [13], commonly called the work-time principle, characterizes what is arguably the most important benefit of parallel programs, which is that a parallel program can be executed on a multiprocessor to obtain near linear speedups. For a computation, let *raw work*, written w, refer to the total number of executed instructions, and let *raw depth*, written d, refer to the length longest dependent chain of executed instructions. Brent's theorem shows that we can execute a computation with w raw work and d raw depth in no more than $w/P + d$ steps on P processors using any *greedy scheduler*. [1] A greedy scheduler is a scheduler that can find available work immediately. This assumption is reasonably realistic, as practical multiprocessor scheduling algorithms, such as work-stealing, can match Brent's bound asymptotically for certain relatively large classes of computations, *e.g.*, fork-join and nested data-parallel computations.

In the execution model with raw work and raw depth, each instruction implicitly is assigned unit cost. Unfortunately, this model does not direcly account for task-creation overheads. To assess the significance of these overheads in implicitly parallel programs, we consider a lambda calculus with parallel tuples and present a cost-semantics for evaluating expression of this language (Section 2). The cost semantics accounts for task-creation overheads by assigning non-unit costs to the operations generating such overheads. In addition to raw work and raw depth, the cost semantics yield total work, total depth of each evaluated expression. We define *total work*, written $\mathcal{W}$, as the total cost of the evaluated instructions, and *total depth*, written $\mathcal{D}$, as the total cost of the most expensive dependent chain of evaluated instructions—total work and total depth include task-creation overheads.

Using this cost semantics, we show that task creation overheads can be a significant multiplicative factor of the raw work. To understand the understand the impact of the overheads, we adapt Brent's theorem to take them into account (Section 2). Specifically, we show that parallel computations with total work $\mathcal{W}$ and total depth $\mathcal{D}$ can be executed in no more than $\mathcal{W}/P+\mathcal{D}$ steps. Intuitively, this bound shows that task-creation overheads contribute directly to the parallel run time just like any other work. Combined with the result that task-creation overheads can increase total work by a multiplicative factor, the generalized Brent's theorem implies that the overheads slow down parallel run time by a multiplicative factor.

To reduce task-creation overheads, we propose an alternative *oracle semantics* that capture a well-known principle for avoiding the task-creation overheads. We evaluate a task in parallel only if its is sufficiently large, *i.e.*, greater than some *cutoff* 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 (Sections 2 and 3). These bounds suggest that we can reduce the task-creation overheads significantly, if we can realize the semantics in practice. This unfortunately is impossible because it requires determining a priori task-creation 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. We show that if we have prior knowledge of the raw work and the raw depth of a computation, then we can pick the optimal cutoff constant κ that yields the fastest parallel run time for a class of computations. We also show that, under some assumptions, there exists a constant κ that reduces the task creation overheads to a small constant ratio of the raw work, without

[1] Note that the bound is tight within a factor of two.

increasing the depth of the computation in a way that would significantly affect the run time.

To realize the oracle semantics in practice, we describe a scheduling technique that we call *oracle scheduling* (Section 4). Oracle scheduling relies on a *task-size estimator* that can estimate the actual run time of parallel tasks in constant-time within a constant factor of accuracy, and a conventional greedy scheduling algorithm, *e.g.*, work-stealing, or a parallel depth-first scheduler. Oracle schedulers perform efficient parallel task creation 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 (asymptotic complexity 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 are expressed without hardware-dependent constants, and profiling techniques, which can be used to determine these constants precisely.

We present a prototype implementation of the proposed approach (Section 5) by extending the OCAML language to support parallel tuples and complexity functions. The implementation translates programs written in this extended language to the PML (Parallel ML) language [17]. Although our implementation requires the programmer to enter the complexity information, this information could also be inferred in some cases via static analysis (*e.g.*, [25] and references therein). In our implementation, for simplicity 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 extend the Manticore compiler for PML to support oracle scheduling and use it to compile generated PML programs. Our experiments (Section 6) show that oracle implementation can reduce the overheads of a single processor parallel execution to between 3 and 13 percent of the sequential time. When using 16 processors, we achieve 7- to 15-fold speedups on an AMD machine and 6- to 10-fold speedups on an Intel machine.

2. Source Language

To give an accurate account of the cost of task creation, and to specify precisely our compilation strategy, we consider a source language in the style of the λ-calculus and present a dynamic cost semantics for it. The semantics and the costs are parameterized by τ and ϕ, which represent the cost of creating a parallel task and the cost of consulting an external oracle for predicting the sizes of its two branches respectively. By using a known proof technique, we generalize Brent's theorem to take task-creation overheads into account.

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

Figure 1. Abstract syntax of the source language

2.1 Cost semantics

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. For simplicity of exposition, we consider parallel tuples of arity two only. Parallel tuples of higher arity can be easily represented with those of arity two.

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 1 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 include references; for the purposes of this paper, however, they add no additional insight and thus are omitted for clarity.

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

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

In addition to an evaluating expression, the dynamic semantics also returns cost measures including *raw work* and *raw depth* denoted by w and d (and variants), and *total work* and *total depth*, denoted by $\mathcal{W}$ and $\mathcal{D}$ (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 $\mathcal{D}$.

Figure 2 shows the complete inductive definition of the dynamic cost semantics judgment $e \Downarrow^{\alpha} v, (w, d), (\mathcal{W}, \mathcal{D})$. When evaluating any expression that is not a parallel tuple,

(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)

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

(case-right)

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

(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)

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

(ptuple-orc-sequentialize)

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

Figure 2. Dynamic cost semantics

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

Evaluation of parallel tuples vary depending on the mode.

- **Sequential mode.** Parallel tuples are treated exactly like sequential tuples: evaluating a parallel tuple simply contributes 1 to the raw and the total work (depth), which are computed as the sum of the work (depth) of the two branches plus 1. In the sequential mode, raw and total work (depth) are the same.
- **Parallel mode.** The evaluation of parallel tuples induces an additional constant cost τ. The depth is computed as

the maximum of the depths of the two branches of the parallel tuple plus 1, and work is computed as the sum of the work of the two branches plus τ. In the oracle mode, there are two cases. If the parallel tuple is scheduled sequentially, then its costs 1 unit. Raw/total work and depth are both calculated as the sum of the depth of the branches plus one. If the parallel tuple is evaluated in parallel, then an extra cost τ is included in the total work and depth and the depth is computed as the maximum of the depth of the two branches.

- **Oracle mode.** The scheduling of a parallel tuple depends on the amount of raw work involved in the two branches. If the raw work of each branch is more than κ, then the tuple is evaluated in parallel in the oracle mode. Otherwise, the raw work of at least one branch is less than κ, and the tuple is executed sequentially. When evaluating a parallel tuple sequentially, the mode in which each branch is evaluated depends on the work involved in the branch. If a branch contains more than κ units of raw work, then it is evaluated in oracle mode, otherwise it is evaluated in sequential mode. This switching to sequential mode on small tasks is needed for ensuring that the oracle is not called too often during the evaluation of a program.

2.2 Generalized Brent's theorem

In order to relate the total work and total depth of a program with its execution time, we rely on Brent's theorem. This theorem is usually formulated in terms of *computation DAGs*. A computation DAG is a directed acyclic graph that represents a parallel computation. 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 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 3. A node is said to be *ready* if all the nodes that points to it have already been executed.

Brent's theorem gives a bound on the time required for executing all the tasks in a computation DAG with a greedy scheduler, assuming that each node takes a unit of time to execute. A scheduler is said to be *greedy* if it never stays idle unnecessarily, i.e., when there exists a ready node the scheduler finds it at no cost and executes it. Typical proofs of Brent's theorem assume a unit cost model where each instruction costs a unit cost to execute and construct a "level-by-level" execution schedule.

One way to extend the Brent's theorem to include task-creation overheads is to assign a weight to each node. Proving such a generalization directly, however, turns out to be highly nontrivial and in our attempts resulted in relatively

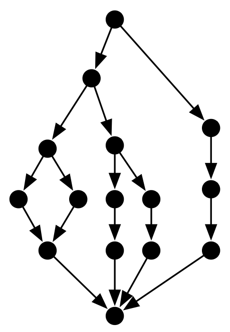
Figure 3. An example computation DAG.

complex proofs. Another approach is to represent non-unit cost tasks with a sequence of unit tasks, e.g., we can replace a task with weight three with a sequence of three unit-cost tasks. Since overheads are non-divisible work, we would require that such tasks execute on the same processors back to back without interleaving with other tasks. With this approach, typical proofs of Brent's theorem, which assume a "level-by-level" execution schedule, do not work because they break up sequences. Fortunately, we have found that Arora et al's proof [3] can be adapted easily for this purpose, because it makes no assumption about ordering of ready nodes, directly allowing us to generalize Brent's theorem to include task-creation overheads.

Theorem 2.1 (Brent's theorem for computation DAGs) *Let G be a computation DAG made of W nodes and whose longest path has length D. Any greedy scheduler can execute this computation DAG in no more than $\frac{W}{P} + D$ steps on P processors.*

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

Consider a given time step. If all processors are executing then the idle bucket receives zero tokens. Otherwise, a number of processors are idle. In this case, the idle bucket receives between one and $P - 1$ tokens. We can bound the number of time steps at which this situation happens, as follows. If one or more processors are idle, it means that those processors cannot find a ready task to execute. Because the scheduler is assumed to be greedy, it must be the case that all the ready tasks are currently executing. Therefore, at such a time step, the maximal length of a path in the computation DAG starting from a ready node decreases by one unit. Because the maximal length of a path in the computation DAG is initially D, there can be at most D time steps at which not all processors are executing. It follows that the final number of tokens in the idle bucket does not exceeed $(P-1)D$. This result entails the inequality $I \leq PD$. □

Observe that the proof does not impose any constraint on the order in which the ready tasks should be executed by the processors. So, if one processor starts working on a sequence of several nodes, then it can execute all the nodes in the sequence before looking for other ready tasks. Therefore, the proof accepts computation DAGs that encode non-unit tasks as sequences of unit tasks. We will make use of such an encoding in the proof of our next theorem, which relates our cost semantics to the computation DAG model.

Theorem 2.2 (Brent's theorem for the cost semantics) *Assume $e \Downarrow^{orc} v, (w, d), (\mathcal{W}, \mathcal{D})$ to hold for some v, w and d. Any greedy scheduler can execute the expression e in no more than $\frac{\mathcal{W}}{P} + \mathcal{D}$ computations steps on P processors.*

Proof In order to invoke the version of Brent's theorem that applies to computation DAGs, we build the computation DAG associated with the execution of the expression e, including nodes that represent the cost of scheduling. To that end, we describe a recursive algorithm for turning an expression e with total work $\mathcal{W}$ and total depth $\mathcal{D}$ into a corresponding computation DAG containing $\mathcal{W}$ nodes and whose longest path has length $\mathcal{D}$. The algorithm follows the structure of the derivation that e has total work $\mathcal{W}$ and total depth $\mathcal{D}$.

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

It is straightforward to check that, in each case, $\mathcal{W}$ and $\mathcal{D}$ match the number of nodes and the total depth of the DAG being produced. □

3. Analysis

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

3.1 Ideal oracle

We quantify the relationships between raw work, raw depth and total work, total depth for each mode.

Theorem 3.1 (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	$\mathcal{W} \leq (1 + \frac{\tau}{2})\, w$	$\mathcal{D} \leq (1 + \tau)\, d$
orc	$\mathcal{W} \leq (1 + \frac{\tau}{\kappa+1})\, w$	$\mathcal{D} \leq (1 + \texttt{max}\,(\tau, \kappa))\, d$

Proof The equations concerning the sequential semantics follow by inspection of the semantics of the source language (Figure 2). The inequalities for the parallel and the oracle modes follow directly by our more general bounds presented later (Theorems 3.2 and 3.3). To prove that the inequalities for the parallel and the oracles modes are tight, we give example computation that achieve the bounds.

- **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 $n + (n-1)(1+\tau)$. We therefore have $\mathcal{W} = (1 + \frac{n\tau}{2n+1})w \leq \left(1 + \frac{\tau}{2}\right) w$. As n increases, the bound approaches $\left(1 + \frac{\tau}{2}\right) w$ and thus the bound on the total work is tight. To see that the depth bound is also tight, note that each parallel tuple adds 1 to the raw depth and $1 + \tau$ to the total depth. The total depth therefore can be as much as $1 + \tau$ times greater than the raw depth.

- **Oracle mode.** Consider an expression with n nested parallel tuples, where tuples are always nested in the right branch of their parent tuple. The tuples are built on top of expressions that involve κ units of work. In the oracle semantics, all the tuples are executed in parallel. Thus the raw work w is $n+(n+1)\kappa$, the total work $\mathcal{W}$ is $n(1+\tau)+(n+1)\kappa$, and $\mathcal{W} = w\left(1 + \frac{n\tau}{n(\kappa+1)+\kappa}\right) \leq w\left(1 + \frac{\tau}{\kappa+1}\right)$. As n increases, the bound approaches $\left(1 + \frac{\tau}{\kappa+1}\right)w$ and thus the bound on the total work is tight.
For the depth bound, we consider two cases. In the first case, we have $\tau \geq \kappa$. Using the same example, the raw depth is $d = n+1$, the total depth is $\mathcal{D} = n(1+\tau)+\kappa$, and $\mathcal{D} = \left(1 + \frac{n\tau+\kappa-1}{n+1}\right) d \leq (1+\tau)\,d$. As n increases, $\mathcal{D}$ approaches $(1+\tau)\,d$ and thus the bound is tight.
For the second case when $\kappa \geq \tau$, we change the example slightly by reducing the amount of raw work in each leaf to just under κ. This will cause all the parallel tuples to be evaluated sequentially; the raw depth is $d = n+\kappa$ and the total depth is equal to the total work, *i.e.*, $\mathcal{D} = n + (n+1)\kappa \leq \left(1 + \frac{n\kappa}{n+\kappa}\right) d$. As n increases, $\mathcal{D}$ approaches $(1+\kappa)\,d$ and thus the bound is tight. □

This theorem leads to some important conclusions. First, the theorem shows that task creation (scheduling) costs matter a great deal. In a parallel evaluation, the total work and total depth can be as much as τ times larger than the raw depth and raw work. This essentially implies that a parallel program can be significantly slower than its sequential counterpart. If τ is large compared to the number of processors, then even in the ideal setting, where the number of parallel processors is small relative to τ, we may observe no speedups. In fact, it is not uncommon to hear anecdotal evidence of this kind of slowdown in modern computer systems.

Second, the theorem shows that evaluation of a program with an ideal oracle can require as much as $\frac{\kappa}{2}$ less work than in the parallel mode. This comes at a cost of increasing the depth by a factor of $\frac{\kappa}{\tau}$. Increasing the depth of a computation can hurt parallel execution times because many parallel schedulers rely on the availability of large degree of parallelism to achieve optimal speedups. Unless done carefully, increasing the depth can dramatically reduce parallel slackness. In the common case, however, where there is plenty of parallelism, i.e., when $\frac{w}{P}$ is far greater than d, we can safely increase depth by a factor of $\frac{\kappa}{\tau}$ to reduce the task-creation overheads. Concretely, if parallel slackness is high and κ is not too large, then κd remains small compared to $\frac{w}{P}$, and $\frac{\tau}{\kappa}\frac{w}{P}$ becomes much smaller than $\frac{\tau}{2}\frac{w}{P}$, dramatically reducing task-creation overheads without harming parallel speedups.

3.2 Realistic oracles

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

We show that even with a realistic oracle, we can reduce task creation overheads. We start with bounding the depth; the result implies that the total depth is no larger than $\mu\kappa$ times the raw depth when κ is large compared to τ and ϕ. Since with the ideal oracle this factor was κ, the bound implies that the imprecision of the oracle can be influenced by changing the constant multiplicative factor.

Theorem 3.2 (Depth with a realistic oracle)

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

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

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

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

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

• Now, consider the case of a parallel tuple. First, assume that the two branches of this tuple are predicted to be large. In this case, the tuple is executed in parallel and the branches are executed in oracle mode. We exploit the induction hypotheses $\mathcal{D}_1 \leq \rho d_1$ and $\mathcal{D}_2 \leq \rho d_2$ to conclude as follows:

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

• 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 $w_1 \leq \mu\kappa$ and $w_2 \leq \mu\kappa$. 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 $\mathcal{D}_1 \leq \mu\kappa$ and $\mathcal{D}_2 < \mu\kappa$. Below, we also exploit the fact that $\mathtt{max}\,(d_1,d_2) \geq 1$, 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 leading to n parallel tuples generating $2n$ values as leaves. The raw work is equal to $(n-1)+n+2n$, and the total work is $(n-1)+n\phi+2n$. Thus, $\mathcal{W} \le \frac{\phi}{4}w$ and this is tight for large values of n. This means that a program executed according to the oracle semantics can slow down by as much as $\phi/4$.

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

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

Definition 3.1 (Balanced programs) *For $\gamma \ge 1$, a program or expression e is γ-balanced if evaluating e in the oracle mode invokes the oracle only for subexpressions whose raw work is no less than $\frac{\kappa}{\mu\gamma}$.*

Note that if a program is γ-balanced and if $\gamma < \gamma'$, then this program is also γ'-balanced. We will later give a sufficient condition for proving that particular programs are balanced (§3.4).

Theorem 3.3 (Work with a realistic oracle) *Assume $e \Downarrow^{orc} v, (w, d), (\mathcal{W}, \mathcal{D})$ where e is a γ-balanced program.*

$$\mathcal{W} \quad \le \quad \left(1 + \frac{\mu(\tau + \gamma\phi)}{\kappa + 1}\right) w.$$

Proof We establish the following slightly tighter inequality.

$$\mathcal{W} \quad \le \quad \left(1 + \frac{\tau}{\kappa/\mu \ + \ 1} + \frac{\phi}{\kappa/(\mu\gamma) \ + \ 1}\right) w.$$

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

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

This is indeed a strengthened result because we have:

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

$$\text{and} \qquad \phi \left\lfloor \tfrac{(w-\kappa'')^+}{\kappa''+1} \right\rfloor \le \phi \tfrac{w}{\kappa''+1} \le \tfrac{\phi}{\kappa/(\mu\gamma)\,+\,1}\, w$$

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

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

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

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

So, we can easily derive the conclusion:

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

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

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

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

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

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

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

• Assume now that the two branches are predicted to be less than the cutoff. This implies $w_1 \leq \kappa'$ and $w_2 \leq \kappa'$. Both these tasks are executed sequentially, so $\mathcal{W}_1 = w_1$ and $\mathcal{W}_2 = w_2$. Since the program is γ-balanced, we have $w_1 \geq \kappa''$ and $w_2 \geq \kappa''$. 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 $w_1 + w_2 + 1 - \kappa'' \geq \kappa'' + 1$, we know that $\left\lfloor \frac{w_1+w_2+1-\kappa''}{\kappa''+1} \right\rfloor \geq 1$. Therefore:

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

• 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 $w_1 \geq \kappa'$ and $w_2 \leq \kappa'$. 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 $\mathcal{W}_1 \leq w_1 + \tau \left\lfloor \frac{w_1-\kappa'}{\kappa'+1} \right\rfloor + \phi \left\lfloor \frac{w_1-\kappa''}{\kappa''+1} \right\rfloor$. For the second task, which is executed sequentially, we have $\mathcal{W}_2 = w_2$. Moreover, the regularity hypothesis gives us $w_2 \geq \kappa''$. Hence, we have $\left\lfloor \frac{w_2+1}{\kappa''+1} \right\rfloor \geq 1$. We conclude as follows:

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

□

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

Theorem 3.4 (Execution time with a realistic oracle) *Assume an oracle that costs ϕ and makes an error by a factor not exceeding μ. Assume $\kappa > \tau$, which is always the case in practice. The execution time of a parallel γ-balanced program on a machine with P processors under the oracle semantics with a greedy scheduler does not exceed the value*

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

Proof The bound follows by the version of Brent's theorem adpated to our cost semantics (Theorem 2.2), and by the bounds established in Theorem 3.3 and Theorem 3.2. For simplicity, we have replaced the denominator $\kappa + 1$ with κ. This change does not loosen the bound significantly because κ is usually very large in front of a unit cost. □

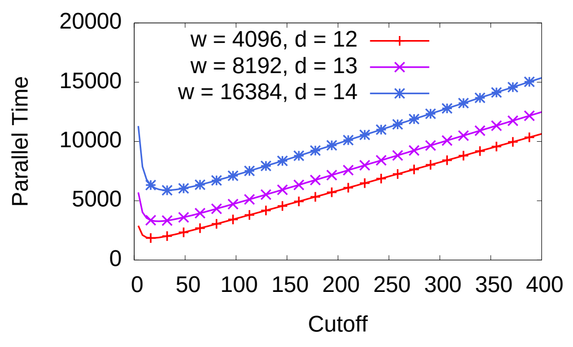

Figure 4. An illustration of the run-time function $\left(1+\frac{\mu(\tau+\gamma\phi)}{\kappa}\right)\frac{w}{P} + (\kappa\mu+\phi+1)\,d$ on $P = 4$ processors with constants $\mu = 1$, $\tau = 5$, $\gamma = 1$, and $\phi = 2$, and different work and depth values.

3.3 Choice of the cutoff

Theorem 3.4 shows that the running time of a parallel program can be controlled by changing the constant κ; the formula, however, reveals an interesting tradeoff: we can reduce task-creation overheads but this comes at the cost of increasing the depth. To see this connection better, consider the bound that appears in the statement of Theorem 3.4 and notice that as κ increases the work (first) term decreases but the depth (second) term increases. Figure 4 illustrates a concrete instance of the bound for a hypothetical computation for fixed constants but different raw work and raw depth. The exact values of the constant and the raw work and depth are not relevant to our discussion; constants are fixed at some reasonable values consistent with our experimental observations. The work and depth are consistent with a program whose raw work is linear in the input size and whose raw depth is logarithmic in the input size.

As Figure 4 illustrates, the parallel run time decreases as we increase κ up to some inflection point and then starts increasing. We compute the optimal value for κ by solving for the root of the derivative. We obtain:

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

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

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

Theorem 3.5 (Run time with fixed κ) *Consider an oracle with ϕ cost and μ error. For any $\gamma \geq 1$ and for any constant r such that $0 < r < 1$, there exists a constant κ and a constant c such that the evaluation with the oracle semantics of a γ-balanced program reduces task creation overheads to a fraction r of the raw work, while in the same time increasing the total depth by no more than a factor $\frac{c}{r}$. With a greedy scheduler, the total parallel run time on P processors of such a program therefore does not exceed $(1+r)\frac{w}{P} + \frac{c}{r}\,d$.*

Proof Consider a particular γ-balanced program with raw work w and raw depth d, and consider its evaluation under the oracle semantics. By Theorem 3.3 we know that total work does not exceed

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

To achieve the desired bound on execution time, we take $\kappa = \frac{\mu(\tau+\gamma\phi)}{r}$. Plugging this value of κ into the formula yields $(1+r)\,w$ for total work, showing that task creation overheads are reduced to a fraction r of the raw work.

Furthermore, by Theorem 3.2 we know that the total depth is bounded by $(\mathtt{max}\,(\tau,\mu\kappa)+\phi+1)\,d$. Plugging in the same value for κ yields the following bound on total depth:

$$\mathcal{D} \leq \left(\mathtt{max}\left(\tau, \frac{\mu^2(\tau+\gamma\phi)}{r}\right)+\phi+1\right)d.$$

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

$$\mathcal{D} \leq \left(\frac{\mu^2(\tau+\gamma\phi)}{r}+\frac{\phi+1}{r}\right)d.$$

Choosing $c = \mu^2(\tau+\gamma\phi)+\phi+1$ therefore ensures that the total depth does not exceed the desired bound $\frac{c}{r}\,d$. The run-time bound follows by an application of Brent's theorem (Theorem 2.2). $\square$

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

3.4 Balanced programs

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

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

Definition 3.2 (Domination of a parallel branch) *A branch e of a parallel tuple is said to be* dominated *by the branch e_i of another parallel tuple $(|e_1, e_2|)$ if the expression e is involved in the execution of the branch e_i.*

Definition 3.3 (Regularity of a parallel program) *A program is said to be γ-regular if, for any parallel branch involving, say, w units of raw work, either w is 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. Observe that every program is ∞-regular.

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

The following lemma explains how the regularity assumption can be exploited to ensure that the oracle is never invoked on 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 3.1 (From regularity to balanced)
If a program is γ-regular then it is γ-balanced.

Proof We have to show that, during the execution of a γ-regular program according to oracle semantics, the oracle is never invoked on subexpressions involving less than $\kappa/(\mu\gamma)$ raw work. Consider a particular subexpression e involving w units of raw work, and assume that the oracle is invoked on this subexpression. Because the oracle is being invoked, e must correspond to the branch of a parallel tuple. By the regularity assumption, either w is very large compared with $\kappa/(\mu\gamma)$, in which case the conclusion holds immediately, or the branch e is dominated by a branch e_i that involves that involves w' units of work, with $w' \leq \gamma w$. For the latter case, we need to establish $w \geq \kappa/(\mu\gamma)$. To that end, it suffices to prove that $w' \geq \kappa/\mu$, which amounts to showing that the amount of raw work associated with the dominating branch e_i contains at least κ/μ raw work.

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

```
type cost
type estimator
val create: unit -> estimator
val report: estimator× cost× float -> unit
val predict: estimator× cost-> float
```

Figure 5. The signature of the estimator data structure

4. Oracle Scheduling

As we describe in this section, we can realize the oracle semantics 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*.

Run-time estimators. To realize the oracle semantics, we require the user to provide a *cost function* for each function in the program and rely on an *estimator* for estimating actual work using the user-provided cost information. When applied to an argument `v`, a cost function of `f` returns the abstract cost of the application of `f` to `v`. The cost is passed to the estimator, which uses the cost to compute an estimate of the actual execution time, that is, the raw work, of the application. Figure 5 shows a signature for the estimator. To perform accurate estimates, the estimator utilizes profiling data obtained from actual execution times. The sampling operation `report (t, c, e)` adds a cost `c` and an execution time `e` to the set of samples in an estimator `t`. An estimate of the actual execution time is obtained by calling `predict`. Given an estimator `t` and cost `c`, the call `predict (t, c)` returns a predicted execution time.

Compilation. To support oracle scheduling with estimators, we need compilation support to associate an estimator with each function defined in the program code, to derive a sequential and an oracle version for each function, and to evaluate tuples sequentially or in parallel depending on the approximations performed by the estimator.

For simplicity, we assume that constituents of parallel tuples are function applications, *i.e.*, they are of the form

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

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

The second pass of our compilation scheme uses the allocated estimators to approximate the actual raw work of function applications and relies on an MakeBranch function to determine whether an application should be run in the oracle or in the sequential mode. Figure 6 defines more precisely the second pass. We write $[\![v]\!]$ for the translation of a value v, and we write $[\![e]\!]^\alpha$ for the translation of the expression e according to the semantics α, which can be either seq or orc. When specifying the translation, we use triples, quadruples, projections, sequence, if-then-else statements, and unit value; these constructions can all be easily defined in our core programming language.

Translation of values other than functions does not depend on the mode and is relatively straightforward. We translate functions, which are of the form "fun $f.x.e_b$ $[e_c|r]$", into a quadruple consisting of the estimator r, a sequential cost function, the sequential version of the function, and the oracle versions of the function. Translation of a function application depends on the mode. In the sequential mode, the sequential version of the function is selected (by projecting the third component of the function) and used in the application. Similarly, in the oracle mode, the oracle version of the function is selected and used in the application. To translate a tuple, we recursively translate the subexpression, while preserving the mode. Similarly, translation of the let, projections, and case constructs are entirely structural.

In the sequential mode, a parallel tuple is turned into a simple tuple. In the oracle mode, the translation applies the oracle-based scheduling policy with the aid of the meta-function MakeBranch. This meta-function, shown in Figure 7, describes the template of the code generated for preparing the execution of a parallel tuple. MakeBranch expects a (translated) function f and its (translated) argument v, and it returns a boolean b indicating whether the application of f to v is expected to take more or less time than the cutoff κ, and a thunk t to execute this application. On the

```
MakeBranch $(f, v)$ $\equiv$
    let $r$ = proj$^1$ $f$ in
    let $m$ = proj$^2$ $f$ $v$ in
    let $b$ = predict$(r, m) > \kappa$ in
    let fun $k_{\mathsf{seq}}$ $()$ = proj$^3$ $f$ $v$ in
    let fun $k'_{\mathsf{seq}}$ $()$ = MeasuredRun$(r, m, k_{\mathsf{seq}})$ in
    let fun $k_{\mathsf{orc}}$ $()$ = 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
    report $(r, m, (t' - t))$;
    $v$
```

Figure 7. Auxiliary meta-functions used for compilation.

one hand, if the application is predicted to take more time than the cutoff (in which case b is true), then the thunk t corresponds to the application of the oracle-semantics version of the function f. On the other hand, if the application is predicted to take less time than the cutoff (in which case b is false), then the thunk t corresponds to the application of the sequential-semantics version of the function f. Moreover, in the latter case, the time taken to execute the application sequentially is measured. This time measure is reported to the estimator by the auxiliary meta-function MeasuredRun (Figure 7), so as to enable its approximations.

Observe that the translation introduces many quadruples and applications of projection functions. However, in practice, the quadruples typically get inlined so most of the projections can be computed at compile time. Observe also that the compilation scheme involves some code duplication, because every function is translated once for the sequential mode and once for the oracle mode. In theory, the code could grow exponentially when the code involves functions defined inside the body of other functions. In practice, the code the growth is limited because functions are rarely deeply nested. If code duplication was a problem, then we can use flattening to eliminate deep nesting of local functions, or pass the mode α as an extra argument to functions.

Cost as complexity functions. The techniques described in this section require the programmer to annotate each function defined in the program with a cost function that, when applied to the argument, returns an abstract cost value. This abstract cost value is then used by an estimator, which is also left abstract, to approximate the actual raw work of a task. For our bounds to apply, complexity expressions should require constant time to evaluate.

Predicting the raw work is only needed for sequential tasks, so the estimator actually needs to return an approximation of the actual run time of a sequential task. A cru-

$$
\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 \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}\} \\
\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 & \left\{ \begin{array}{l} \texttt{let}\ (b_1, k_1)\ =\ \texttt{MakeBranch}(\llbracket f_1 \rrbracket, \llbracket v_1 \rrbracket)\ \texttt{in} \\ \texttt{let}\ (b_2, k_2)\ =\ \texttt{MakeBranch}(\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{array} \right.
\end{array}
$$

Figure 6. Translation for oracle scheduling.

cial property of the abstract cost is that it should be abstract enough that the programmer can write the cost functions without necessarily knowing the details of the hardware that the programs will be executed on. Yet, abstract costs should provide sufficient information to estimate the actual run times.

Asymptotic complexity specifications serve as a natural cost function by satisfying both of these properties. Since they eliminate hardware specific constants, they can be specified easily. Using complexity functions, we can approximate the actual run time of sequentially executed functions by simply determining the constants hidden by the asymptotic complexity notation. Such an approximation can be performed by using the least squares method or similar techniques for data fitting from known samples.

In our implementation described in Section 5, we implement an approach based on complexity functions. We define `cost` as an integer, which represents the application of the complexity function applied to the input size. We approximate the actual run time by calculating a single constant, assuming that the constants in all terms of the asymptotic complexity are the same. Although assuming a single constant can decrease the precision of the approximations, we believe that it suffices because we only have to compute lower bounds for our functions; *i.e.*, we only need to determine whether they are "big enough" for parallel execution.

5. 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 [26], which is a strict functional language. Our Caml dialect corresponds to the core Caml language extended with syntax for parallel pairs and complexity annotations. Figure 8 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.

```
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 8. An example parallel program.

We use the Caml type checker to obtain a typed syntax tree, on which we perform the oracle-scheduling translation defined in Figure 6. We then produce code in the syntax of Parallel ML (PML) [17], 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 run-time 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 [16]. 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.

Run-time 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 of 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 estimator data structure. In this section, we discuss the pratical implementation of the evaluation of constant factors.

In order for the measurement of the constant to be lightweight, we simply compute average values of the constant. The constant might evolve over time, for example if the current program is sharing the machine with another program, a series of memory reads by the other program may slow down the current program. For this reason, we do not just compute the average across the entire history, but instead maintain a moving average, that is, an average of the values gathered across a certain number of runs.

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 estimator contains the estimated value for the constant that is read by all the processors when they need to predict execution times. The local data structures are used to accumulate statistics on the value of the constant. Those statistics are reported on a regular basis to the shared memory cell, by computing a weighted mean between the value previously stored in the shared memory cell and the value obtained out of the local data structure. We treat initializations somewhat specially: for the first few measures, a processor always begins by reporting its current average to the shared memory cell. This 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 detecting these outliers to be relatively easy because the measured times 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 more accurate techniques would be required, 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 [11]. 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 [34] adopts a code-specialization scheme, called clone translation, taken from Cilk-5's implementation [19].[2] With clone translation, each parallel-pair expression is compiled into two versions: the fast clone and the slow clone. The purpose of a fast clone is to optimize the code that corresponds to evaluating on the local processor, whereas the slow clone is used when the second branch of a parallel-pair is migrated to another processor. A common aspect of between clone translation and our oracle translation (Figure 6) 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

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

it involves inter-processor communication. It is well established, both through analysis and experimentation, that (with high probability) no more than $O(P\mathcal{D})$ steals occur during the evaluation [11]. So, for programs that exhibit parallel slackness ($\mathcal{W} \gg P\mathcal{D}$), we do not need to take into account the cost of slow clones because there are relatively few of them. We focus only on the cost of creating fast clones, which correspond to the cost τ. A fast clone needs to packages a task, push it onto the deque and later pop it from the deque. So, a fast clone is not quite as fast as the corresponding sequential code. The exact slowdown depend on the implementation, but in our case we have observed that a fast clone is 3 to 5 times slower than a simple function call.

6. 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 with 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, and 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 core 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 task-creation overheads for each of our test machines. To estimate, we use a synthetic benchmark expression e whose evaluation sums integers between zero and 30 million using a parallel divide-and-conquer computation. We chose this particular expression because most of its evaluation time is spent evaluating parallel pairs.

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

We then define the work-stealing overhead $c_w = \frac{w_w}{w_s}$. We estimate the cost τ of creating a parallel 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 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 §3.2. Recall that r is the targette overhead for scheduling costs. We aim for $r = 10\%$. Our oracle appears to be always accurate within a factor 2, so we set $\mu = 2$. 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 and $60\mu s$ 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, we used $\kappa = 26\mu s$ on the AMD machine and $\kappa = 61\mu s$ on the Intel machine.

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

The Quicksort benchmark sorts a sequence of 2 million integers. Our program is adapted from a functional, tree-based algorithm [6]. The algorithm runs with $O(n \log n)$ raw work and $O(\log^2 n)$ raw depth, where n is the length of the sequence. Sequences of integers are represented as binary trees in which sequence elements are stored at leaf nodes and each internal node caches the number of leaves contained in its subtree.

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

The Barnes-Hut benchmark is an n-body simulation that calculates the gravitational forces between n particles as they move through 2-d space [4]. The Barnes-Hut compu-

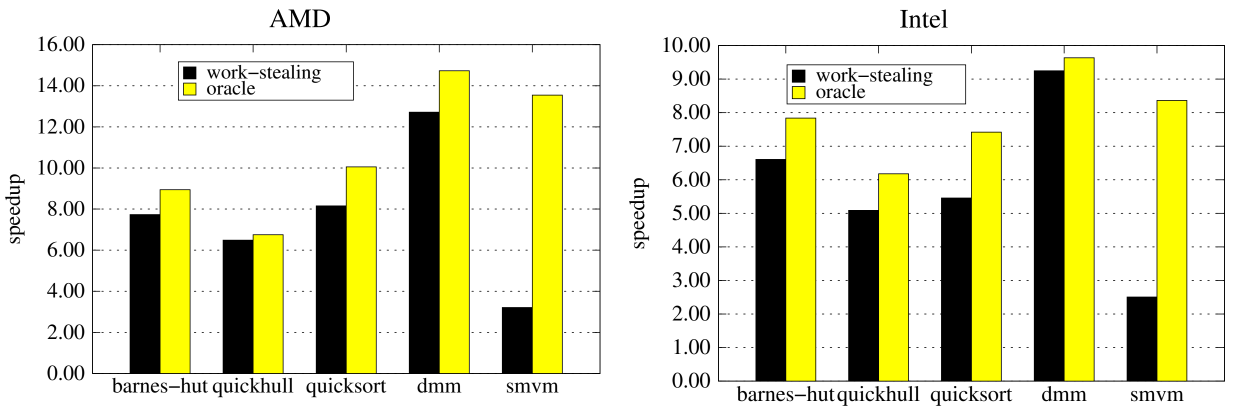

Figure 9. Comparison of the speedup on sixteen processors. Higher bars are better.

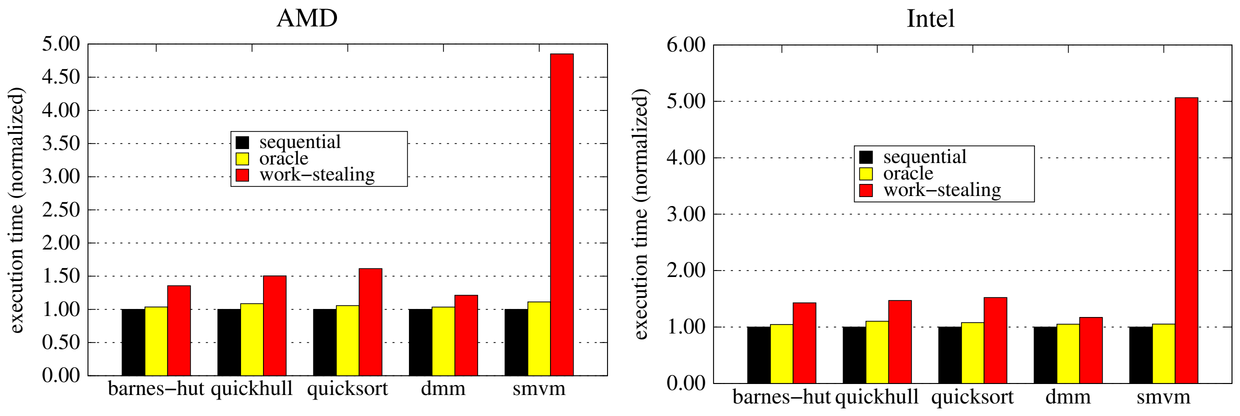

Figure 10. Comparison of execution times (normalized) on a single processor. Lower bars are better.

tation consists of two phases. In the first, the simulation volume is divided into square cells via a quadtree, so that only particles from nearby cells need to be handled individually and particles from distant cells can be grouped together and treated as large particles. The second phase calculates gravitational forces using the quadtree to accelerate the computation. The algorithm runs with $O(n \log n)$ raw work and $O(\log n)$ raw depth. Our benchmark runs 10 iterations over 100,000 particles generated from a random Plummer distribution [33]. The program is adapted from a Data-Parallel Haskell program [31]. 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 [18]. We have recursion go down to scalar elements. The algorithm runs with $O(n^3)$ raw work and $O(\log n)$ raw depth. We selected $n = 512$.

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 [9]. 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 thus the complexity function 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.

$$\left([8,3,9,2,3,1,4,5],\quad \begin{tikzcd}[column sep=small] & & & 7 \arrow[dll, dash] \arrow[drr, dash] \\ & 4 \arrow[dl, dash] \arrow[dr, dash] & & & & 3 \arrow[dl, dash] \arrow[dr, dash] \\ 3 & & 1 & & 2 & & 1 \end{tikzcd}\right)$$

To summarize, in order to write a constant-time complexity function, we changed 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 task-creation overheads. T_{par}^{P} denotes the execution time with work stealing on P processors. T_{orc}^{P} 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 9 shows the speedup on sixteen processors for each of our benchmarks, that is, the values $T_{\text{par}}^{16}/T_{\text{seq}}$ and $T_{\text{orc}}^{16}/T_{\text{seq}}$. 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 10. This plot shows execution time for one processor, normalized with respect to the sequential execution times. In other

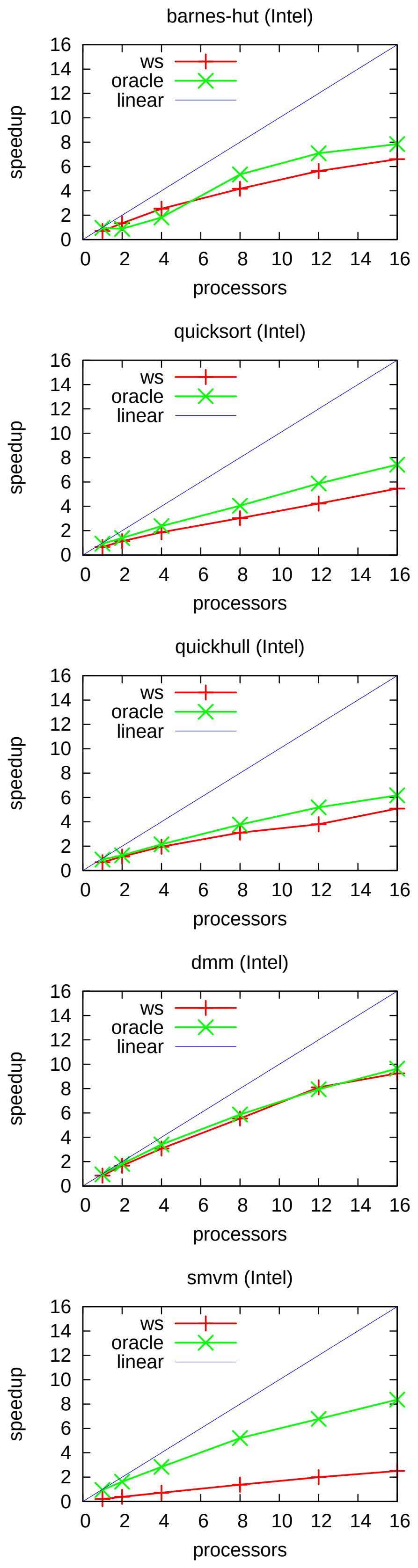

Figure 11. Comparison between work stealing and oracle.

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 task-creation overheads in the oracle implementation do not exceed 13% of the raw work in any benchmark. The cases where we observe large improvements in speedup are the same cases where there is a large difference 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 11 shows speedup curves for each of our experiments, that is, values of $T^P_{\mathsf{par}}/T_{\mathsf{seq}}$ and $T^P_{\mathsf{orc}}/T_{\mathsf{seq}}$ against the number of processors P on our Intel machine; the measurements on the AMD machine show similar trends but quantitatively better results for the oracle versions.

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, which is probably due to a lack of parallelism in the program. Notice, however, that our scheduler does not fall below work stealing.

7. Related Work

Cutting off excess parallelism. This study is not the first to propose using cost prediction to determine when to cut off parallelism. One approach, developed in early work in functional programing, uses list size to determine cut offs [24]. Using list size alone is limited, because the technique assumes linear work complexity for every parallel operation.

Another way to handle cost prediction is to use the depth and height of the recursion tree [30, 42]. But depth and height are not, in general, the most direct means to predict the execution time of subcomputations. In our oracle scheduling, we ask for either the programmer or compiler to provide for each function a cost function that expresses the asymptotic cost of applying the function.

Lopez *et. al.* take this approach as well, but in the context of logic programming [27]. On the surface, their technique is similar to our oracle scheduling, except that their cost estimators do not utilize profiling to estimate constant factors. An approach without constant-factor estimation is overly simplistic for modern processors, because it relies on complexity function predicting execution time exactly. On modern processors, execution time depends heavily on factors such as caching, pipelining, *etc.* and it is not feasible in general to predict execution time from a complexity function alone.

Reducing per-task costs. One approach to the granularity problem is to focus on reducing 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 [15, 19, 23, 28, 34, 36]. 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 [40]. 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 [5]. 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 [2], 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 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 interprocessor 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 [29]. 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 [10]. Implementations of flattening include NESL [8] and Data Parallel Haskell [32]. 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 [38]. 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 [7], as a consequence of the transformation making changes to data structures defined in the program. Our transformation involves no such changes.

The NESL [8] and Data Parallel Haskell [32] 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 [6, 7, 39]. The idea of instrumenting evaluations to generate cost information goes back to the early 90s [35, 37].

Inferring Complexity Bounds. Our implementation of oracle scheduling requires the programmer to enter complexity bounds for all parallel tasks. In some cases, these bounds can be inferred by various static analyses, for example, using type-based and other static analyses (e.g., [14, 25]), symbolic techniques (e.g., [20, 21]). Our approach can benefit from these approaches by reducing the programmer burden, making it ultimately easier to use the proposed techniques in practice.

8. Conclusion

In this paper, we propose a solution to the granularity-control problem. We prove that an oracle that can approximate the sizes of parallel tasks in constant time within a constant factor of accuracy can be used to reduce the task creation overheads to any desired constant fraction for a reasonably broad class of computations. We describe how such an oracle can be integrated with any scheduler to support what we call oracle scheduling. We realize oracle scheduling in practice by requiring the programmer to enter asymptotic complexity annotations for parallel tasks and by judicious use of runtime profiling. Consistently with our theoretical analysis, our experiments show that oracle scheduling can reduce task creation overheads to a small fraction of the sequential time without hurting parallel scalability.

References

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

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] Karl Crary and Stephnie Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '00, pages 184–198. ACM, 2000.

[15] 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.

[16] 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. ACM, 2008.

[17] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011.

[18] 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.

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

[20] Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. Measuring empirical computational complexity. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, pages 395–404, 2007.

[21] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 127–139, 2009.

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

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

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

[30] Joseph Pehoushek and Joseph Weening. Low-cost process creation and dynamic partitioning in qlisp. In Takayasu Ito and Robert Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 182–199. Springer Berlin / Heidelberg, 1990.

[31] 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, 2008.

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

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

[34] Mike Rainey. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. PhD thesis, University of Chicago, August 2010.

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

[36] 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.

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

[38] Daniel Spoonhower. *Scheduling Deterministic Parallel Programs*. Ph. D. dissertation, Carnegie Mellon University, Pittsburg, PA, USA, 2009.

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

[40] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *Proceedings of the 2010 ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*. ACM Press, February 2010.

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

[42] Joseph S. Weening. *Parallel Execution of Lisp Programs*. PhD thesis, Stanford University, 1989. Computer Science Technical Report STAN-CS-89-1265.