A Semantic Framework for Scheduling Parallel Programs

Daniel Spoonhower, Guy E. Blelloch, and Robert Harper

Carnegie Mellon University, Pittsburgh, PA 15213, USA,
{spoons,blelloch,rwh}@cs.cmu.edu

Abstract. Declarative parallel programs offer deterministic results, allowing the language implementation to schedule parallel tasks in any order. However, program performance hinges crucially on the way that these tasks are scheduled. In this work, we use formal language semantics to express different scheduling policies. These semantics enable us to compare different policies and to understand their effects on the use of space. We offer several example programs to demonstrate that scheduling policy can have a dramatic, and even asymptotic, effect on space usage. To predict performance, programmers require a means to understand the effects of scheduling. We define a cost semantics that allows programmers to reason about how space is used by parallel declarative programs. At the same time, these costs provide a specification for how implementations should behave.

1 Introduction

Parallel architectures are now ubiquitous, and the number of processor cores is quickly overtaking the number of applications typically run by users. To keep pace with the number of cores, applications must process the increasingly large amounts data stored on personal computers *in parallel*.

Declarative parallel languages [McGraw, 1982, Arvind et al., 1989, Feo et al., 1990, Blelloch et al., 1994, Aditya et al., 1995, Chakravarty and Keller, 2000] allow parallel algorithms to be expressed concisely and in a platform-independent manner. Declarative parallel programs are deterministic, meaning that they will always yield the same results, regardless of how they are evaluated. Using declarative languages is important in ensuring that programs take advantage of many different parallel platforms and that they continue to scale with advances in hardware technology: these languages encourage programmers to express massive amounts of parallelism, far greater than the number of processors or cores available today.

While it is easy to reason about the results of declarative parallel programs, their performance depends heavily on many aspects of the language implementation, including control of task granularity, communication, and scheduling policy. In this work, we focus on scheduling policy and develop a formal model to reason about its effect on performance. Through a simple example, we show how the choice of scheduling policy can have asymptotic effects on space use.

Assessing and improving the performance of declarative programs is still a black art, especially with respect to their use of space. Current practice relies on

runtime space profiling [Runciman and Wakeling, 1993]. While useful in many cases, space profilers make assumptions about how programs are compiled, and the results depend on runtime factors including hardware configuration, garbage collector implementation, and (most relevant to the current work) scheduling policy.

To reason formally about scheduling, we express each policy not in prose or as source code, but as a dynamic semantics for our language. An important class of policies is also described using a similar semantics. This framework allows us to analyze and compare policies using induction and other familiar techniques.

We also present an abstract *cost semantics* [Blelloch and Greiner, 1996], which assigns a time and space cost measure to each program. Our costs include enough information that they can be used to understand the performance of different scheduling policies. In the context of these costs, each semantics implementing a scheduling policy will play the role of a *provable implementation*, one in which we can guarantee the asymptotic space use matches that predicted by the cost semantics.

Our approach offers several benefits. A high-level presentation of scheduling policies offers programmers a mechanism for understanding performance independently of a particular implementation. This includes different implementations of the same policy. By constraining the intensional behavior of programs, our cost semantics also provides a specification and guide for language implementers. Finally, formally specifying the dynamic behavior of a language is the first step in reasoning statically about programs, including their performance.

2 Related Work

Interest in side effect-free parallel programming began decades ago with languages such as Val [McGraw, 1982], Id [Arvind et al., 1989], and Sisal [Feo et al., 1990]. These researchers also argue that a deterministic semantics is essential in writing correct parallel programs and that a language without side effects is a natural means to achieve determinism.

Though we use a small, call-by-value language in our analysis, our results are applicable to larger languages such as NESL [Blelloch et al., 1994], pH [Aditya et al., 1995], and Nepal [Chakravarty and Keller, 2000]. Although these languages require the ordering of parallel tasks to be determined dynamically, none of them directly address how this is to be accomplished.

Various work has described operational semantics for parallel execution (*e.g.*, Hudak and Anderson [1987], Roe [1991], Aditya et al. [1995]). None of it has attempted to capture possible schedules at a high level. Evaluation strategies [Trinder et al., 1998] allow the programmer to explicitly control the structure parallel evaluation (*e.g.*, divide-and-conquer, data-parallelism). Other than the work mentioned in the introduction [Blelloch and Greiner, 1996] none of this work has tried to capture space usage.

Within the algorithms community there has been significant research on the effects of scheduling policy on memory usage [Blumofe and Leiserson, 1998, Blelloch et al., 1999, Narlikar and Blelloch, 1999]. In that work, it has been shown that different schedules can have an exponential difference in memory usage with

$$
\begin{array}{lrcl}
\text{(expressions)} & e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid (e_1, e_2) \mid \pi_i\ e \mid \mathsf{let\ par}\ d\ \mathsf{in}\ e \mid v \\
\text{(values)} & v & ::= & \langle x.e \rangle^{\ell} \mid \langle v_1, v_2 \rangle^{\ell} \\
\text{(locations)} & \ell & \in & L \\
\text{(declarations)} & d & ::= & x = e \mid d_1\ \mathsf{and}\ d_2 \\
\text{(value declarations)} & \delta & ::= & x = v \mid \delta_1\ \mathsf{and}\ \delta_2
\end{array}
$$

Fig. 1. Language Syntax. We extend the lambda calculus with declarations and values. Declarations and the parallel let construct allow us to express the evaluation of parallel expressions in progress. Declarations within a let par may step in parallel. Annotated values capture sharing explicitly.

only two processors. These works consider an abstract computation model based on threads instead of a language-based model as used in this work.

Previous work on cost semantics for parallel languages [Blelloch and Greiner, 1996] has shown that size and depth of a program execution (when presented as a directed graph) can be used to predict an upper bound on the time and space requirements when run on a parallel machine. Blelloch and Greiner assumed a *fixed* scheduling policy. We model the behavior of the scheduler explicitly in our semantics and refine their cost model so that we can reason precisely about the behavior of different scheduling policies.

Cost semantics have also been used to reason about the performance of sequential programs. Non-strict, purely functional programs can be evaluated in many different ways, and it is widely understood that different strategies for evaluation can yield wildly different performance results. Ennals [2004] uses a cost semantics to compare the work performed by a range of sequential evaluation strategies, ranging from lazy to eager. Gustavsson and Sands [1999] give a semantic definition of what it means for a program transformation to be "safe for space." They provide several laws to help prove that a given transformation does not asymptotically increase the space usage of call-by-need programs. Unlike these previous works, we consider a strict language. Thus the variations in performance are not due to whether or not a particular expression is evaluated (since all scheduling policies evaluate exactly the same set of expressions), but the *order* in which those expressions are evaluated.

3 Parallel Semantics

As a demonstration of our technique, we consider a call-by-value functional language extended with parallel pairs (Figure 1). Although pairs allow only a bounded number of new parallel tasks to be introduced in each step, this will be sufficient to demonstrate that scheduling policy can have an asymptotic effect on performance.

We use a purely functional language, but concerns about scheduling and the use of space arise in any declarative parallel language, functional or imperative, where there are more parallel tasks than processors.

In this section, we define several policies, each as a transition (small-step) semantics for this language. The first is a non-deterministic semantics that

allows unbounded parallelism. Though this semantics does not provide a useful implementation, it will serve as the definition of all possible parallel evaluation strategies. Subsequent semantics will each define the behavior of a particular scheduling policy or a class of related policies.

The language we use is described in Figure 1. We extend the usual syntax of expressions with a parallel let construct. This construct is used to denote expressions whose parallel evaluation has begun but not yet finished. Declarations within a let par may step in parallel, depending on the constraints enforced by one of the transition semantics below. let par expressions and declarations reify a stack of expression contexts such as those that appear in many abstract machines. Unlike a stack, which has exactly one topmost element, there are many "leaves" in our syntax that may evaluate in parallel.

We also include a separate syntactic class of values. Values are annotated with locations so that sharing can be made explicit without resorting to an explicit heap: the syntax distinguishes between a pair whose components occupy the same space and one whose components do not. This is critical, as we will take the set of labels appearing in an expression as a measure of its space use.

Though we use recursive functions in our examples, we omit them in this presentation for the sake of space and clarity. We stress that our framework can easily accommodate recursive functions and therefore possibly non-terminating programs. As it is a dynamic analysis, however, it does not offer any information about individual program instances that diverge. While this is certainly a limitation of our framework, it is also a limitation of current practice using heap profilers [Runciman and Wakeling, 1993]. Furthermore, this is not a significant issue with respect to the implementation of parallel algorithms (as opposed to web servers and other interactive applications) since we are only interested in instances of program execution that actually yield results.

To facilitate the definition of several different parallel semantics, we first factor out those parts of the implementation that are common to all variations. These primitive sequential transitions are defined by the following judgment.

$$e \longrightarrow e'$$

This judgment represents the step taken by a single processor in one time step. A selection of the axioms that define this judgment appear in Figure 2 and the remainder are shown in Appendix A.1. These axioms *limit* where parallel evaluation may occur by defining the intermediate form for the evaluation of pairs. Where parallel evaluation *actually* occurs is defined by the scheduling semantics, as given in the remainder of this section.

3.1 Non-Deterministic Scheduling

The non-deterministic (ND) transition semantics defines all possible parallel executions. It is given by a pair of judgments

$$e \overset{\text{nd}}{\longmapsto} e' \qquad\qquad d \overset{\text{nd}}{\longmapsto} d'$$

that state, *expression e takes a single parallel step to e'* and, similarly, *declaration d takes a single parallel step to d'*. This semantics allows unbounded parallelism: it

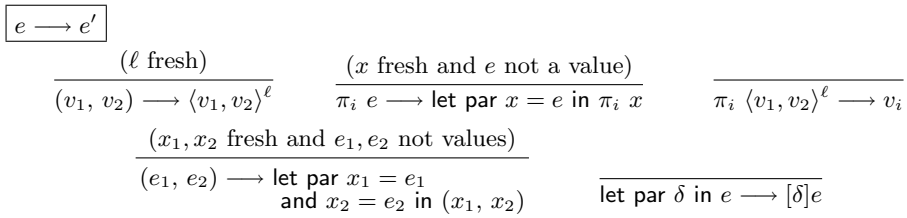


Fig. 2. Selected Primitive Transitions. These rules encode transitions where no parallel execution is possible. They will be used in each of the different scheduling semantics that follow in this section. The substitution of a value declaration into an expression $[\delta]e$ simultaneously substitutes for all the variables appearing in δ. The remaining rules appear in Appendix A.1.

models execution on a parallel machine with an unbounded number of processors. It is defined by the rules in Figure 3.

Most of the ND rules are straightforward. The only non-determinism lies in the application of the rule ND-IDLE. In a sense, this rule is complemented by ND-BRANCH: the latter says that all branches may be executed in parallel, but the former allows any sub-expression to sit idle during a given parallel step.

Extensional Behavior Though this implementation is non-deterministic in *how* it schedules parallel evaluation, the *result* of ND evaluation will always be the same, no matter which expressions evaluate in parallel. This statement is formalized in the following theorem. In this and other results below, analogous facts hold for declarations. In addition, we always consider equality of expressions and declarations up to the renaming of locations.

Theorem 1 (Confluence). *If* $e \overset{nd}{\longmapsto}{}^* e'$ *and* $e \overset{nd}{\longmapsto}{}^* e''$ *then there exists an expression* e''' *such that* $e' \overset{nd}{\longmapsto}{}^* e'''$ *and* $e'' \overset{nd}{\longmapsto}{}^* e'''$.

Proof. This proof follows from the "diamond" property as shown in Lemma 1.

Lemma 1. *If* $e \overset{nd}{\longmapsto} e'$ *and* $e \overset{nd}{\longmapsto} e''$, *then there exists an expression* e''' *such that* $e' \overset{nd}{\longmapsto} e'''$ *and* $e'' \overset{nd}{\longmapsto} e'''$.

Proof. By induction on the derivation of $e \overset{\mathrm{nd}}{\longmapsto} e'$.

Case ND-IDLE**:** In this case $e' = e$. Assume that $e \overset{\mathrm{nd}}{\longmapsto} e''$ was derived using rule R. Let $e''' = e''$. Then we have $e \overset{\mathrm{nd}}{\longmapsto} e''$ (by applying R) and $e'' \overset{\mathrm{nd}}{\longmapsto} e''$ (by ND-IDLE), as required.

As all of the non-determinism in this semantics is concentrated in the use of the ND-IDLE rule, the remaining cases follow from the immediate application of the induction hypothesis.

$\boxed{e \overset{\text{nd}}{\longmapsto} e'}$

$$\frac{d \overset{\text{nd}}{\longmapsto} d'}{\mathsf{let\ par}\ d\ \mathsf{in}\ e \overset{\text{nd}}{\longmapsto} \mathsf{let\ par}\ d'\ \mathsf{in}\ e}\ (\text{ND-Let})$$

$$\frac{}{e \overset{\text{nd}}{\longmapsto} e}\ (\text{ND-Idle}) \qquad \frac{e \longrightarrow e'}{e \overset{\text{nd}}{\longmapsto} e'}\ (\text{ND-Prim})$$

$\boxed{d \overset{\text{nd}}{\longmapsto} d'}$

$$\frac{e \overset{\text{nd}}{\longmapsto} e'}{x = e \overset{\text{nd}}{\longmapsto} x = e'}\ (\text{ND-Leaf}) \qquad \frac{d_1 \overset{\text{nd}}{\longmapsto} d_1' \quad d_2 \overset{\text{nd}}{\longmapsto} d_2'}{d_1\ \mathsf{and}\ d_2 \overset{\text{nd}}{\longmapsto} d_1'\ \mathsf{and}\ d_2'}\ (\text{ND-Branch})$$

Fig. 3. Non-Deterministic Parallel Transition Semantics. This semantics defines all possible parallel transitions of an expression, including those that take an arbitrary number of primitive steps in parallel. Parallelism is isolated within expressions of the form let par. Declarations step in parallel using ND-Let. Note that expressions (or portions thereof) may remain unchanged using the rule ND-Idle.

We also prove several properties relating the behavior of this parallel semantics to that of a sequential semantics. We write $e \Downarrow v$ for the standard call-by-value sequential semantics. The first such property (Completeness) states that any result obtained using an ordinary sequential semantics can also be obtained using the ND implementation.

Theorem 2 (ND Completeness). *If* $e \Downarrow v$ *then* $e \overset{nd}{\longmapsto}{}^* v$.

The proof is carried out by induction on the derivation of $e \Downarrow v$.

The following theorem (Soundness) ensures that any result obtained by the ND semantics can also be derived using the sequential semantics.

Theorem 3 (ND Soundness). *If* $e \overset{nd}{\longmapsto}{}^* v$, *then* $e \Downarrow v$.

The proof follows by induction on the number of steps n in the sequence of transitions. The base case is given by Lemma 2 as shown in the Appendix.

Summary These theorems establish that our parallel language is deterministic and will yield the same (extensional) result regardless which parallel execution is chosen. Programmers may therefore use the sequential semantics to reason about program correctness.

3.2 Depth-First Scheduling

We now define an alternative transition semantics that is deterministic and implements a depth-first schedule. Depth-first (DF) schedules prioritize the leftmost sub-expressions of a program and always complete the evaluation of these leftmost sub-expressions before progressing to sub-expressions on the right. The semantics

in this section implements a p-depth-first (p-DF) schedule, *i.e.*, a depth-first schedule that uses at most p processors. The p-DF transition semantics is defined by the following pair of judgments.

$$p; e \overset{\text{df}}{\longmapsto} p'; e' \qquad p; d \overset{\text{df}}{\longmapsto} p'; d'$$

These judgments define a single parallel step of an expression or declaration. Configurations $p; e$ and $p; d$ describe an expression or declaration together with a non-negative integer p that indicates the number of processors that have not yet been assigned a task in this parallel step. On the right-hand side, p indicates the number of processors that remain unused.

The p-DF transition semantics is defined by the rules given in Figure 4. Most notable is the DF-Branch rule. It states that a parallel declaration may take a step if any available processors are used first on the left sub-declaration and then any remaining available processors are used on the right. Like the ND semantics above, the p-DF transition semantics relies on the primitive transitions given in Figure 2. In rule DF-Prim, one processor is consumed when a primitive transition is applied.

$$\boxed{p; e \overset{\text{df}}{\longmapsto} p'; e'}$$

$$\frac{}{p; v \overset{\text{df}}{\longmapsto} p; v}\ (\text{DF-Val}) \qquad \frac{p; d \overset{\text{df}}{\longmapsto} p'; d'}{p; \mathsf{let\ par}\ d\ \mathsf{in}\ e \overset{\text{df}}{\longmapsto} p'; \mathsf{let\ par}\ d'\ \mathsf{in}\ e}\ (\text{DF-Let})$$

$$\frac{}{0; e \overset{\text{df}}{\longmapsto} 0; e}\ (\text{DF-None}) \qquad \frac{e \longrightarrow e'}{p+1; e \overset{\text{df}}{\longmapsto} p; e'}\ (\text{DF-Prim})$$

$$\boxed{p; d \overset{\text{df}}{\longmapsto} p'; d'}$$

$$\frac{p; e \overset{\text{df}}{\longmapsto} p'; e'}{p; x = e \overset{\text{df}}{\longmapsto} p'; x = e'}\ (\text{DF-Leaf}) \qquad \frac{p; d_1 \overset{\text{df}}{\longmapsto} p'; d_1' \quad p'; d_2 \overset{\text{df}}{\longmapsto} p''; d_2'}{p; d_1\ \mathsf{and}\ d_2 \overset{\text{df}}{\longmapsto} p''; d_1'\ \mathsf{and}\ d_2'}\ (\text{DF-Branch})$$

Fig. 4. p-Depth-First Parallel Transition Semantics. This deterministic semantics defines a single parallel step for left-to-right depth-first schedule using at most p processors. Configurations $p; e$ and $p; d$ describe expressions and declarations with p unused processors remaining in this time step.

For the DF semantics, we must reset the number of available processors after each parallel step. To do so, we define a "top-level" transition judgment for DF evaluation with p processors. This judgment is defined by exactly one rule, shown below. Note that the number of processors remaining idle p' remains unconstrained.

$$\frac{p; e \overset{\text{df}}{\longmapsto} p'; e'}{e \overset{\text{p-df}}{\longmapsto} e'}$$

The complete evaluation of an expression, as for the non-deterministic semantics, is given by the reflexive, transitive closure of the transition relation $\overset{\text{p-df}}{\longmapsto}{}^*$.

We can easily show the DF semantics is correct with respect to a sequential semantics by showing that its behavior is contained within that of the ND semantics. Informally, this theorem states, "the DF semantics is a valid parallel implementation."

Theorem 4 (DF Soundness). *If* $p; e \overset{df}{\longmapsto} p'; e'$ *then* $e \overset{nd}{\longmapsto} e'$.

Proof. By induction on the derivation of $p; e \overset{\text{df}}{\longmapsto} p'; e'$. The case for derivations ending with rule DF-LET follows immediately from an appeal to the induction hypothesis and the analogous rule in the ND semantics. DF-PRIM also follows from its analogue. Rules DF-NONE and DF-VAL are both restrictions of ND-IDLE.

It follows immediately that $e \overset{\text{p-df}}{\longmapsto} e'$ implies $e \overset{\text{nd}}{\longmapsto} e'$. This result shows the benefit of defining a non-deterministic semantics: once we have shown the soundness of an implementation with respect to the non-deterministic semantics, we get soundness with respect to the sequential semantics for free.

While this implementation says nothing of the queues and locks that would appear in a more concrete implementation of a DF scheduler, it offers a clear presentation that is easy to understand and compare with other implementations. Moreover, we believe it mirrors the space use (up to a constant factor) of a concrete implementation.

3.3 Greedy Scheduling

A *p-greedy* scheduler is any scheduler that will take exactly p primitive steps in parallel if p or more steps can be taken; if there are less than p such steps, all available primitive steps will be taken. The p-DF scheduler is one example of a greedy scheduler. Many policies fall into this category, and many important results (*e.g.*, Blelloch et al. [1999]) describe the behavior of these polices as a group.

We can describe this class of policies using another parallel semantics. This greedy semantics lies at an intermediate point between the ND and DF semantics. It is still non-deterministic, but like the DF semantics, does not have an unrestricted "idle" rule. Instead, it includes VAL, NONE, and PRIM rules like the DF semantics, as well as the following rule constraining parallel execution.

$$\frac{p_1; d_1 \overset{\text{gr}}{\longmapsto} p_1'; d_1' \quad p_2; d_2 \overset{\text{gr}}{\longmapsto} p_2'; d_2'}{p_1 + p_2; d_1 \ \mathsf{and}\ d_2 \overset{\text{gr}}{\longmapsto} p_1' + p_2'; d_1' \ \mathsf{and}\ d_2'} \ (\text{GR-BRANCH})$$

This rule requires that all available processors be divided between the two branches. If we take this semantics as the definition of greedy, then to show that a particular policy is greedy, we need only to show that any derivation in that semantics gives rise to a similar derivation in the greedy semantics.

3.4 Breadth-First Scheduling

Just as the semantics above captured the behavior corresponding to a depth-first traversal of the computation graph, we can also give an implementation

corresponding to a breadth-first (BF) traversal. This is the most "fair" schedule in the sense that it distributes computational resources evenly across parallel tasks. For example, given four parallel tasks, a 1-BF scheduler alternates round-robin between the four. A 2-BF scheduler takes one step for each of the first two tasks in parallel, followed by one step for the second two, followed again by the first two, and so on.

We omit the presentation of this semantics due to space limitations, and merely state that, like the depth-first semantics, the breadth-first policy depends on restrictions of the (ND-IDLE) and (ND-BRANCH) rules. Like the DF semantics, the breadth-first semantics is greedy, as can be shown by proving that whenever a BF rule can be applied, a greedy rule can also be applied to achieve the same result.

4 Cost Semantics

The cost semantics (Figure 5) for our language is an evaluation semantics that computes both a result and an abstract cost reflecting *how* that result is obtained. It assigns a single cost to each (closed) program that allows us to construct a model of parallel execution and reason about the behavior of different scheduling policies.

A cost semantics is critical for any asymptotic analysis of running time or space use. For sequential implementations, there is an obvious cost semantics that nearly all programmers understand implicitly. Moreover, because we are familiar with sequential implementations, we can easily move between costs at the level of source code and those of compiled code.

In this work, we give a cost semantics that serves as a basis for the asymptotic analysis of parallel programs, including their use of space. We believe that it is especially important that this semantics assigns costs to source-level programs as programmers are far less familiar with parallel implementations than their sequential counterparts. While the implementations in the previous section are relatively abstract, they serve as a bridge between our cost semantics and lower-level implementations.

The cost semantics is defined by the following judgment, which is read, *expression e evaluates to value v with computation graph g and heap graph h.*

$$e \Downarrow v; g; h$$

The extensional portions of this judgment are standard in the way that they relate expressions to values. As discussed below, edges in a computation graph represent control dependencies in the execution of a program, while edges in a heap graph represent dependencies on and between heap values. We omit declarations and values in the cost semantics as we treat these as intermediate forms and limit programmers to the source-level terms that appear in Figure 5.

As in the work of Blelloch and Greiner [1996], computation graphs g are directed, acyclic graphs with exactly one source node (with in-degree 0) and one sink node (with out-degree 0) (*i.e.* directed series-parallel graphs). Nodes are denoted ℓ and n (and variants). Edges in the computation graph point forward

$\boxed{e \Downarrow v; g; h}$

$$\frac{(\ell \text{ fresh})}{\lambda x.e \Downarrow \langle x.e \rangle^{\ell}; [\ell]; \{(\ell, \ell')\}_{\ell' \in \text{locs}(x.e)}} \text{ (E-Fn)}$$

$$\frac{e_1 \Downarrow \langle x.e_3 \rangle^{\ell_1}; g_1; h_1 \quad e_2 \Downarrow v_2; g_2; h_2 \quad [v_2/x]e_3 \Downarrow v_3; g_3; h_3 \quad (n \text{ fresh})}{e_1\ e_2 \Downarrow v_3; g_1 \oplus g_2 \oplus [n] \oplus g_3; h_1 \cup h_2 \cup h_3 \cup \{(n, \ell_1), (n, \text{loc}(v_2))\}} \text{ (E-App)}$$

$$\frac{e_1 \Downarrow v_1; g_1; h_1 \quad e_2 \Downarrow v_2; g_2; h_2 \quad (\ell \text{ fresh})}{(e_1, e_2) \Downarrow \langle v_1, v_2 \rangle^{\ell}; g_1 \otimes g_2 \oplus [\ell]; h_1 \cup h_2 \cup \{(\ell, \text{loc}(v_1)), (\ell, \text{loc}(v_2))\}} \text{ (E-Pair)}$$

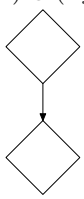
$$\frac{e \Downarrow \langle v_1, v_2 \rangle^{\ell}; g; h \quad (n \text{ fresh})}{\pi_i\ e \Downarrow v_i; g \oplus [n]; h \cup \{(n, \ell)\}} \text{ (E-Proj}_i\text{)}$$

Fig. 5. Profiling Cost Semantics. This semantics yields two graphs that can be used to reason about the parallel performance of programs. Computation graphs g record dependencies in time while heap graphs h record dependencies among values. The location of a value $\text{loc}(v)$ is the outermost label of a value and is defined in Appendix A.3.

in time. An edge from n_1 to n_2 indicates that n_1 must be executed before n_2. Each computation graph consists of a single-node, or of the sequential or parallel composition of smaller graphs. Graphs are written as tuples such as $(n_s; n_e; E)$ where n_s is the source or *start* node, n_e is the sink or *end* node, and E is a list of edges. The remaining nodes of the graph are implicitly defined by the edge list. A graph consisting of a single node n is written $(n, n, \emptyset)$ or simply $[n]$. Graph operations are defined below. (Here, a diamond stands for an arbitrary subgraph.)

Serial Composition

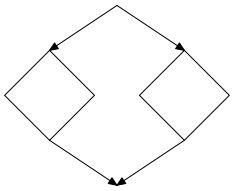
$(n_s; n_e; E) \oplus (n'_s; n'_e; E')$



$(n_s; n'_e; E, E', (n_e, n'_s))$

Parallel Composition

$(n_s; n_e; E) \otimes (n'_s; n'_e; E')$



$(n; n'; E, E', (n, n_s), (n, n'_s), (n_e, n'), (n'_e, n'))$

n, n' fresh

We extend the work of Blelloch and Greiner [1996] with heap graphs. Heap graphs are also directed, acyclic graphs but do not have distinguished start or end nodes. Each heap graph shares nodes with the computation graph arising from the same execution. In a sense, computation and heap graphs may be considered as two sets of edges on a shared set of nodes. As above, the nodes of heap graphs are left implicit.

While edges in the computation graph point forward in time, edges in the heap graph point backward in time. If there is an edge from n to ℓ then n depends on the value at location ℓ. It follows that any space associated with ℓ cannot be reclaimed until after n has executed and any space associated with n has also been reclaimed.

Edges in the heap graph record both the dependencies among heap values and dependencies on heap values by other parts of the program state. As an example of the first case, in the evaluation rule for pairs, two edges are added to the heap graph to represent the dependencies of the pair on each of its components. Thus, if the pair is reachable, so is each component. In the evaluation of a function application, however, two edges are added to express the *use* of heap values. The first such edge marks a use of the function. The second edge denotes a possible last use of the argument. For strict functions, this second edge is redundant: there will be another edge leading to the argument when it is used. However, for non-strict functions, this is the first point at which the garbage collector might reclaim the space associated with the argument.

This semantics does not capture exact memory behavior but instead asymptotic behavior. For example, it assumes that all values require the same amount of space in the heap. In addition, the analyses in this section, like the policies described above, assume fine-grained scheduling mechanism. Despite this, we believe that our results are equally applicable to more coarse-grained implementations with only a constant factor difference in the performance results.

4.1 Schedules

Together, the computation and heap graphs allow a programmer to analyze the behavior of her program under a variety of hardware and scheduling configurations. For a given program, each configuration will give rise to a particular parallel execution that is determined by the constraints described by the computation graph g.

Definition 1 (Schedule). *A* schedule *of a graph $g = (n_s; n_e; E)$ is a sequence of sets of nodes $N_0, \ldots, N_k$ such that $n_s \notin N_0$, $n_e \in N_k$, and for all $i \in [0, k)$,*

- *$N_i \subseteq N_{i+1}$, and*
- *for all $n \in N_{i+1}$, $\text{pred}_g(n) \subseteq N_i$.*

Here $\text{pred}_g(n)$ is the set of nodes n' such that there is an edge (n', n) in g.

4.2 Roots

To understand the use of space, the programmer must also account for the structure of the heap graph h. Given a schedule $N_0, \ldots, N_k$ for a graph g, consider the moment of time represented by some N_i. Because N_i contains all previously executed nodes and because edges in h point backward in time, each edge (n, ℓ) in h will fall into one of the following three categories.

- Both $n, \ell \notin N_i$. As the value associated with ℓ has not yet been allocated, the edge (n, ℓ) does not contribute to the use of space at time i.

– Both $n, \ell \in N_i$. While the value associated with ℓ has been allocated, the use of this value represented by this edge is also in the past. Again, the edge (n, ℓ) does not contribute to the use of space at time i.
– $\ell \in N_i$, but $n \notin N_i$. The value associated with ℓ has already been allocated, and n represents at one possible use in the future. Here, the edge (n, ℓ) *does* contribute to the use of space at time i.

This leads us to the following definition.

Definition 2 (Roots). *The* roots *of a heap graph h with respect to a location ℓ after evaluation of the nodes in N, written* $\text{roots}_{\ell,h}(N)$*, is the set of nodes ℓ' in N where $\ell' = \ell$ or h contains an edge leading from outside N to ℓ'. Symbolically,*

$$\text{roots}_{\ell,h}(N) = \{\ell' \in N \mid \ell' = \ell \vee (\exists n.(n, \ell') \in h \wedge n \notin N)\}$$

We use the term *roots* to evoke a related concept from work in garbage collection. For the reader who is most comfortable thinking in terms of an implementation, the roots might correspond to those memory locations that are reachable directly from the processor registers or the call stack. This is only one possible implementation, however: the heap graph constrains an implementation only in how that implementation manages space.

To relate an analysis of these cost graphs to the intensional behavior of the ND semantics, we state the following theorem.

Theorem 5 (Cost Completeness). *If $e \Downarrow v; g; h$ and $N_0, \ldots, N_k$ is a schedule of g then there exists a sequence of expressions $e_0, \ldots, e_k$ such that $e_0 = e$ and $e_k = v$ and for all $i \in [0, k)$, $e_i \overset{nd}{\longmapsto}{}^* e_{i+1}$ and $\text{locs}(e_i) \subseteq \text{roots}_{v,h}(N_i)$.*

Here we write $\text{roots}_{v,h}(N)$ as an abbreviation for $\text{roots}_{\text{loc}(v),h}(N)$. The locations of an expression $\text{locs}(e)$ are simply the labels of any values embedded in that expression as defined in Appendix A.3.

The final condition of the theorem states that the use of space in the parallel semantics, as determined by $\mathsf{locs}()$, is approximated by the measure of space in the cost graphs, as given by $\mathsf{roots}()$. The proof hinges on the fact that a schedule for a parallel (respectively, serial) graph can be decomposed into two schedules that are run in parallel (serially).

With theorems of this form, each semantics described in the previous section becomes (in the terminology of Blelloch and Greiner) a *provable implementation*. With a suitable simulation technique (as used in the examples below) or visualization tool, a programmer need not understand the parallel semantics in the previous section: once an implementation is proved correct, we are free to reason about performance using only the cost semantics.

4.3 Examples

We have implemented our cost semantics as an interpreter in Standard ML. This interpreter yields exactly the cost graphs described by the cost semantics (Figure 5). Graphs may then be analyzed to produce numerical results as shown in Figure 6, which are explained below.

Quicksort Consider the following implementation of quicksort where the input list is partitioned and each half is sorted in parallel. Recall that the components of a pair (e_1, e_2) may be evaluated in parallel.

```
fun qsort xs =
  case xs of nil ⇒ nil
           | x::xs ⇒ append (qsort (filter (le x) xs), x::(qsort (filter (gt x) xs)))
```

Figure 6(a) shows an upper bound on the use of space as a function of the input size. By creating cost graphs for each input size we can generate the data shown in the figure. Each point is derived by computing the roots for depth- and breadth-first schedules and then finding all nodes in the heap graph that are reachable from these roots. The largest such set, over all points in time, constitutes the space high-water mark.

The four configurations described in the figure are (from top to bottom) a 1-DF (*i.e.* sequential) schedule, a 2-DF schedule, a 3-DF schedule, and p-BF schedules for $p \leq 3$. While the second requires less space than the first, both require space that is polynomial in the input size. The space used in the third and fourth configurations is a linear function of the input size. In all cases, we are considering the worst-case behavior for this example: sorting a list whose elements are in reverse order. We emphasize that these plots all represent the execution of the *same* program, each with a different scheduling policy.

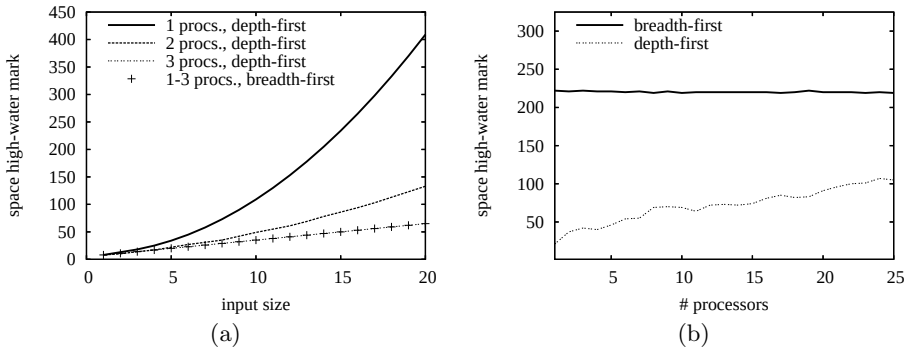


Fig. 6. Space Use. Part (a) shows an upper bound on the space required to sort a list as a function of input size. Each configuration differs in the number of processors or the scheduling policy used to prioritize parallel tasks. Part (b) shows an upper bound on space required to approximate an integral a function of the number of processors.

The first two policies require more space because they start to partition many lists (one for each recursive call) before the first such partitions are completed. In the worst case, each of these in-progress partitions requires $O(N)$ space, and there are N such partitions.

The p-BF schedules avoid this asymptotic blowup in space use by completing all partitions at a given recursive depth before advancing to the next level of

recursion. This implies that no matter how many partitions are in-progress, only $O(N)$ elements will appear in these lists. The p-DF schedules for $p > 2$ achieve the same performance by simulating the behavior of the breadth-first schedules.

Numerical Integration In our second example, we approximate an integral using the adaptive rectangle rule: this algorithm computes the area under a polynomial function as a series of rectangles, using narrower rectangles in regions where the function seems to be undergoing the most change. Whenever the algorithm splits an interval, the approximation of each half is computed in parallel. Note that the parallel structure of this example is determined not by the size or shape of a data structure, but instead by the behavior of the polynomial.

Figure 6(b) shows an upper bound on the space required by this program as a function of the number of processors. Unlike the quicksort example, the depth-first schedule requires far less space for a small number of processors since it quickly reduces intermediate results to a single scalar value. Even as we increase the number of processors, the depth-first scheduler only gradually increases its use of space. In contrast, the breadth-first scheduler requires more space because it determines (and stores) nearly all of the intervals to approximate before aggregating any of the results. Given enough processors, the depth-first scheduler will eventually emulate the behavior (and performance) of the breadth-first scheduler.

5 Conclusion and Future Work

Expressing parallel scheduling policies explicitly in the language semantics offers a powerful mechanism for modeling their behavior. It allows us to compare different policies, for example, showing the correctness of an implementation by proving that its behavior is contained within that of the non-deterministic (ND) semantics. It also allows us to prove a tight correspondence between the behavior of scheduling policies and results of a profiling analysis. This is important as the choice of policy has significant effects on performance.

We briefly draw a connection with another important component of a language implementation, the garbage collector. Programmers who use garbage-collected languages must consider the effect of the collector on application performance. Switching between collector algorithms or changing the configuration of a given algorithm can have significant impact on end-to-end performance. No single collector algorithm is appropriate for all applications. In the same way, we argue that programmers building parallel applications must understand the performance impact of the scheduler: no single scheduling policy is best for all applications. However, in both cases, it sufficient to understand an abstract model of the implementation in order to reason about performance.

One direction for future work lies in building a more concrete implementation of our language: an implementation that runs on parallel hardware. One challenge is understanding the impact of compiler optimizations (such as inlining) on performance. We are currently working toward such an implementation with the goal of validating the predictions made by our framework.

Bibliography

Shail Aditya, Arvind, Jan-Willem Maessen, and Lennart Augustsson. Semantics of pH: A parallel dialect of Haskell. Technical Report Computation Structures Group Memo 377-1, MIT, June 1995.

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, 1989. ISSN 0164-0925.

Guy Blelloch, Phil Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, 1999.

Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

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

R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal of Computing*, 27(1):202–229, 1998.

Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. In *ACM SIGPLAN International Conference on Functional Programming*, pages 94–105, New York, NY, USA, 2000. ACM Press.

Robert Ennals. *Adaptive Evaluation of Non-Strict Programs*. PhD thesis, University of Cambridge, 2004.

John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *J. Parallel Distrib. Comput.*, 10(4):349–366, 1990. ISSN 0743-7315.

Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. *Electr. Notes Theor. Comput. Sci.*, 26, 1999.

Paul Hudak and Steve Anderson. Pomset interpretations of parallel function programs. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 234–256, London, UK, 1987. Springer-Verlag.

James R. McGraw. The VAL language: Description and analysis. *ACM Trans. Program. Lang. Syst.*, 4(1):44–82, 1982. ISSN 0164-0925.

G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Trans. on Programming Languages and Systems*, 21(1):138–173, 1999.

P. Roe. *Parallel Programming Using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 1991.

Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *J. Funct. Program.*, 3(2):217–245, 1993.

Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Funct. Program.*, 8(1):23–60, January 1998.

SUPPLEMENTARY MATERIAL FOLLOWS
(for referees only)

A Appendix

A.1 Primitive Transitions

The full set of primitive transitions for our language appears below.

$$\boxed{e \longrightarrow e'}$$

$$\frac{(\ell \text{ fresh})}{\lambda x.e \longrightarrow \langle x.e\rangle^{\ell}} \qquad \frac{(x_1, x_2 \text{ fresh and } e_1, e_2 \text{ not values})}{e_1\ e_2 \longrightarrow \begin{array}{l}\mathsf{let\ par}\ x_1 = e_1\ \mathsf{in} \\ \quad \mathsf{let\ par}\ x_2 = e_2\ \mathsf{in}\ x_1\ x_2\end{array}} \qquad \frac{}{\langle x.e\rangle^{\ell}\ v_2 \longrightarrow [v_2/x]e}$$

$$\frac{(\ell \text{ fresh})}{(v_1,\, v_2) \longrightarrow \langle v_1, v_2\rangle^{\ell}} \qquad \frac{(x \text{ fresh and } e \text{ not a value})}{\pi_i\ e \longrightarrow \mathsf{let\ par}\ x = e\ \mathsf{in}\ \pi_i\ x} \qquad \frac{}{\pi_i\ \langle v_1, v_2\rangle^{\ell} \longrightarrow v_i}$$

$$\frac{}{\mathsf{let\ par}\ \delta\ \mathsf{in}\ e \longrightarrow [\delta]e} \qquad \frac{(x_1, x_2 \text{ fresh and } e_1, e_2 \text{ not values})}{(e_1,\, e_2) \longrightarrow \begin{array}{l}\mathsf{let\ par}\ x_1 = e_1 \\ \quad \mathsf{and}\ x_2 = e_2\ \mathsf{in}\ (x_1,\, x_2)\end{array}}$$

A.2 Soundness of ND Semantics

The following lemma is used in the proof of soundness for the non-deterministic semantics.

Lemma 2. *If* $e \overset{nd}{\longmapsto} e'$, $e' \overset{nd}{\longmapsto}{}^{*} v$, *and* $e' \Downarrow v$, *then* $e \Downarrow v$.

The proof follows by induction on the derivation of $e \overset{\mathrm{nd}}{\longmapsto} e'$.

A.3 Locations

The location of a value is the outermost label of that value. The locations of an expression are the labels of any values that appear in that expression. Similarly for declarations.

$$\begin{array}{ll}
\mathrm{loc}(\langle x.e\rangle^{\ell}) & = \ell \\
\mathrm{loc}(\langle v_1, v_2\rangle^{\ell}) & = \ell \\
 & \\
\mathrm{locs}(\lambda x.e) & = \mathrm{locs}(e) \\
\mathrm{locs}(e_1\ e_2) & = \mathrm{locs}(e_1) \cup \mathrm{locs}(e_2) \\
\mathrm{locs}((e_1, e_2)) & = \mathrm{locs}(e_1) \cup \mathrm{locs}(e_2) \\
\mathrm{locs}(\pi_i\ e) & = \mathrm{locs}(e) \\
\mathrm{locs}(\mathsf{let\ par}\ d\ \mathsf{in}\ e) & = \mathrm{locs}(d) \cup \mathrm{locs}(e) \\
 & \\
\mathrm{locs}(x = e) & = \mathrm{locs}(e) \\
\mathrm{locs}(d_1\ \mathsf{and}\ d_2) & = \mathrm{locs}(d_1) \cup \mathrm{locs}(d_2)
\end{array}$$