Cost Semantics for Space Usage in a Parallel Language

Daniel Spoonhower
Carnegie Mellon University
spoons@cs.cmu.edu

Abstract

We describe a framework for better understanding scheduling policies for fine-grained parallel computations and their effect on space usage. We define a profiling semantics that can be used to determine the asymptotic space taken by any schedule. A nondeterministic parallel transition semantics is used to describe all possible parallel executions. Refinements of that semantics can be used to model the behavior of particular schedulers. We use the framework to show that different schedules can lead to asymptotic differences in space usage.

1. Introduction

With the advent of multi-core processors, both application programmers and compiler writers are working to exploit the parallelism now available on most desktop and laptop computers. Many parallel applications, however, offer much more parallelism than can be executed by current hardware. It is the role of the scheduler to decide which parallel tasks should be executed first.

There are many different scheduling policies to choose from, and the choice of scheduler can have a very large effect on the performance of the application. Perhaps even more importantly than time, the scheduler can have a huge effect on memory usage.

In practice, programmers who wish to determine the performance of a parallel program must often resort to a combination of empirical methods or analysis of compiled code that are specific to a single compiler, runtime, and architecture.

In this work, we set out to capture the memory usage of particular scheduling policies in the operational semantics of a language. The goal is to supply a user with a better and consistent understanding of memory usage at a high level. The framework is set up so that it can be applied easily to different scheduling policies but abstracts away from unnecessary details of an implementation. Because of this abstraction, the framework does not capture exact memory behavior but rather captures asymptotic behavior.

2. Motivation

Consider the implementation of quicksort given in Figure 1 where the input list is partitioned and each half sorted in parallel. In this work, the components of a pair $(:\ e_1,\ e_2\ :)$ may be evaluated in parallel. Figure 2 shows an upper bound on the use of space as a function of the input size. Each line represents a variation in the number of processor cores or in the scheduling algorithm used to prioritize parallel tasks. Much like the choice of garbage collection algorithm, these runtime parameters affect the performance of a program. Configurations A and B consume space as a polynomial function of the input size, while configuration C requires only linear space. We will discuss this example in further detail below.


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

Figure 1. A Parallel Implementation of Quicksort. The components of a pair $(:\ e_1,\ e_2\ :)$ may be evaluated in parallel.

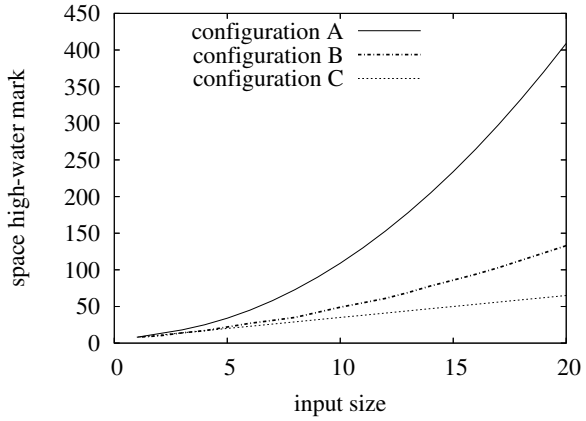

Figure 2. Space Use as a Function of Input Size. This figure shows an upper bound on the space required by the code in Figure 1 as a function of input size. Each configuration differs in the number of processor cores available or in the scheduling algorithm used to prioritize parallel tasks.

3. Cost Semantics

As a demonstration of our technique, we will consider a simple call-by-value functional language extended with parallel pairs. Other forms of expressions such as arrays, arithmetic operators, and recursion may be added, but we elide them for the sake of brevity. We include a separate syntactic class of values. Values are labeled so that sharing can be made explicit.

(expressions)	e	$::=$	$x \mid \lambda x.e \mid e_1\ e_2 \mid (:\ e_1,\ e_2\ :) \mid \pi_i\ e$
(values)	v	$::=$	$\langle \eta; x.e \rangle^{\ell} \mid \langle v_1, v_2 \rangle^{\ell}$
(labels)	ℓ	$\in$	L

Our cost semantics (Figure 3) is an evaluation semantics that computes both the result of computation and an abstract cost re-

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

$$\frac{(\ell \text{ fresh})}{\eta \triangleright \lambda x.e \Downarrow \langle \eta; x.e \rangle^{\ell}; [\ell]; \{(\ell, \ell')\}_{\ell' \in \text{labels}(\eta(x.e))}} \qquad \frac{\eta \triangleright e \Downarrow \langle v_1, v_2 \rangle^{\ell}; g; h \quad (n \text{ fresh})}{\eta \triangleright \pi_i\ e \Downarrow v_i; g \oplus [n]; h \cup \{(n, \ell)\}} \qquad \frac{(x \mapsto v) \in \eta \quad (n \text{ fresh})}{\eta \triangleright x \Downarrow v; [n]; \{(n, \text{label}(v))\}}$$

$$\frac{\eta_1 \triangleright e_1 \Downarrow \langle \eta_2; x.e_3 \rangle^{\ell_1}; g_1; h_1 \quad \eta_1 \triangleright e_2 \Downarrow v_2; g_2; h_2 \quad \eta_2, x \mapsto v_2 \triangleright e_3 \Downarrow v_3; g_3; h_3 \quad (n \text{ fresh})}{\eta_1 \triangleright e_1\ e_2 \Downarrow v_3; g_1 \oplus g_2 \oplus [n] \oplus g_3; h_1 \cup h_2 \cup h_3 \cup \{(n, \ell_1), (n, \text{label}(v_2))\}}$$

$$\frac{\eta \triangleright e_1 \Downarrow v_1; g_1; h_1 \quad \eta \triangleright e_2 \Downarrow v_2; g_2; h_2 \quad (\ell \text{ fresh})}{\eta \triangleright (:\ e_1,\ e_2\ :) \Downarrow \langle v_1, v_2 \rangle^{\ell}; g_1 \otimes g_2 \oplus [\ell]; h_1 \cup h_2 \cup \{(\ell, \text{label}(v_1)), (\ell, \text{label}(v_2))\}}$$

Figure 3. Profiling Cost Semantics. In addition to a result, this semantics also yields two graphs that can be used to reconstruct the cost of obtaining that result. Computation graphs g record control dependencies while heap graphs h record dependencies on and among values.

$$\begin{array}{rcl@{\quad}l@{\quad}l}
[n] & := & (n; & n; & \epsilon) \\
(n_s; n_e; E) & := & (n_s; & n'_e; & E, E', (n_e, n'_s)) \\
\oplus\ (n'_s; n'_e; E') & & & & \\
(n_s; n_e; E) & := & (n; & n'; & E, E', \\
\otimes\ (n'_s; n'_e; E') & & & & (n, n_s), (n, n'_s), \\
& & & & (n_e, n'), (n'_e, n')) \\
& & (n, n' \text{ fresh}) & &
\end{array}$$

Figure 4. Single-Node Graphs and Graph Operations.

flecting *how* the result was obtained. While the semantics is sequential, the cost will allow the programmer to reconstruct different parallel schedules as well as the space use of programs executing with these schedules. The judgment

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

is read, *in environment η expression e evaluates to value v with computation graph g and heap graph h*. As discussed below, edges in the computation graph represent control dependencies in the execution of a program, while edges in the heap graph represent dependencies on and between heap values.

Computation graphs g are directed, acyclic graphs. Each computation graph consists of a single-node, or of the sequential or parallel composition of smaller graphs. Nodes are denoted ℓ and n. Each graph is written as a tuple $(n_s; n_e; E)$ where n_s is the start node, n_e is the end node, and E is a list of edges. The remaining nodes of the graph are implicitly defined by the edge list. Single-node graphs and graph operations are defined in Figure 4.

Heap graphs are also directed, acyclic graphs and share nodes with the computation graph. 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.

Together, the computation and heap graphs allow a programmer to analyze the behavior of her program under a variety of hardware and scheduling configurations. A key component of this analysis is the notion of a *schedule*.

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

Where $\text{pred}_g(n)$ is the set of nodes with edges in g leading to n. Intuitively, a schedule of g is given by the sets of nodes covered by a pebbling of g.

4. Parallel Semantics

As part of an implementation of our language, we extend the syntax of expressions with a parallel let construct. It allows us to express the evaluation of parallel expressions in progress. Declarations within a let par may step in parallel. We also include values within the syntax of expressions, so that substitution may be well-defined.

$$\begin{array}{lrcl}
\text{(expressions)} & e & ::= & \ldots \mid \mathsf{let\ par}\ d\ \mathsf{in}\ e \mid v \\
\text{(declarations)} & d & ::= & x = e \mid d_1\ \mathsf{and}\ d_2
\end{array}$$

We then give a nondeterministic transition semantics that defines all possible parallel executions. This semantics is defined by a pair of judgments

$$e \longmapsto e' \qquad d \longmapsto d'$$

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 models execution on a parallel machine with an unbounded number of execution units. For example, the rule below states that both branches of a declaration may step in parallel.

$$\frac{d_1 \longmapsto d'_1 \quad d_2 \longmapsto d'_2}{(d_1\ \mathsf{and}\ d_2) \longmapsto (d'_1\ \mathsf{and}\ d'_2)}$$

The following rule gives the transition taken by a parallel pair. It states that the parallel execution of a pair begins by forking off two parallel threads (as expressed by the let par construct).

$$\frac{}{(:\ e_1,\ e_2\ :) \longmapsto \mathsf{let\ par}\ x_1 = e_1\ \mathsf{and}\ x_2 = e_2\ \mathsf{in}\ (:\ x_1,\ x_2\ :)}$$

Evaluation continues by evaluating the declarations of the let par using the rule above.

Given a transition semantics, we can state the following theorem relating schedules to transition derivations. It shows that the cost semantics accurately models the parallelism expressed by the transition semantics.

Theorem 1. *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 $i \in [0, k)$, $e_i \longmapsto^* e_{i+1}$ and $\text{labels}(e_i) \subseteq \text{roots}_{h,v}(N_i)$.*

The final condition states that the use of space in the parallel semantics, as determined by labels(), is approximated by the measure of space in the cost graphs, as given by roots().

5. Analysis

To understand how space use is captured by the cost semantics, we first consider the different sorts of edges in the heap graph.

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.

We note that each set of nodes N_i in a schedule represents the partial evaluation of an expression at time i. The space use of a program at that time can be reconstructed by considering those heap edges that cross over into N_i. Since edges in the heap graph point backward in time, an edge that points from a node $n \notin N_i$ to another node $n' \in N_i$ represents a dependency on an existing heap value. These edges are the *roots* of the heap at time i (to borrow from the terminology of garbage collection). The set of all nodes in h reachable by following these roots and other edges in h determines the total space in use at time i.

6. Scheduling

Different scheduling algorithms can be encoded directly into the operational semantics. For example, we can give a deterministic variation of the semantics from Section 4 that corresponds to a depth-first traversal of the computation graph with a finite number of execution units. We can make this notion of a bounded, depth-first traversal precise by defining a restricted form of schedule, one in which nodes on the left are always executed before those on the right. We then prove that every depth-first schedule corresponds to a sequence of depth-first transitions (analogous to the result given by Theorem 1). A similar result can also be given for breadth-first schedules. In each case, the transition semantics gives a high-level description of the behavior of a scheduling algorithm and serves as a guide for implementation.

7. Quicksort: Revisited

We briefly revisit the parallel implementation of quicksort described in Section 2. Configuration A is a single-core, depth-first execution, and configuration B is a dual-core, depth-first execution. While the second requires less space than the first, both require space that is polynomial in the input size. Configuration C represents breadth-first executions for both single- and dual-core processors; the space used in each of these cases is a linear function of the input size. Depth-first schedules fare poorly in this example because these schedules fork off many parallel threads, each of which consumes a large amount of space.

Consider an alternative implementation where the recursive case is structured as follows. In this version, we partition the list in parallel, but then synchronize before recursively sorting each sub-list.

```
...
| x::xs =>
  let val (ls, gs) = (: filter (le x) xs,
                        filter (gt x) xs :)
  in
    append (: qsort ls, x::(qsort gs) :)
  end
```

While this second version makes better use of space under a depth-first schedule, it does so at the cost of introducing more constraints on parallel execution. This example also points to another potential problem for parallel programmers: otherwise innocuous program transformations may adversely affect space usage. In this case, we note that program variables `ls` and `gs` are each used exactly once and are prime candidates for inlining. However, inlining the definitions of `ls` and `gs` produces exactly the version of the code given in Section 2. We leave as future work a characterization of program transformations (such as inlining) that may be safely performed in the context of a parallel language.

8. Related Work

Within the algorithms community there has been significant research [3, 6, 1] on the effect of scheduling policy on memory usage. In that work, it has been shown that different schedules can have an exponential difference in memory usage with only two processors.

Previous work on cost semantics for parallel languages [2] has shown that the 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.

Gustavsson and Sands [4] used a cost semantics to prove the soundness (with respect to space) of several laws describing transformations of call-by-need programs. Minamide [5] showed that a CPS transformation is space efficient using a cost semantics for a call-by-value functional language.

9. Conclusion

While functional languages avoid many problems associated with parallel programming by offering deterministic results, the performance of these programs can vary dramatically, even asymptotically, with changes in the runtime configuration.

In this work, we have described two semantics for a parallel language: a cost semantics that can be used to reconstruct the costs of evaluation, and a transition semantics that defines parallel execution. Several scheduling algorithms can be readily encoded in this second semantics. Together, these two semantics form a contract between the programmer and the language implementor: a specification of how a program can be compiled and how the result should perform.

Acknowledgments

This is joint work with Guy Blelloch and Robert Harper.

References

[1] BLELLOCH, G., GIBBONS, P., AND MATIAS, Y. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the Association for Computing Machinery 46*, 2 (1999), 281–321.

[2] BLELLOCH, G. E., AND GREINER, J. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming* (May 1996), pp. 213–225.

[3] BLUMOFE, R. D., AND LEISERSON, C. E. Space-efficient scheduling of multithreaded computations. *SIAM Journal of Computing 27*, 1 (1998), 202–229.

[4] GUSTAVSSON, J., AND SANDS, D. A foundation for space-safe transformations of call-by-need programs. In *Proceedings of Workshop on Higher Order Operational Techniques in Semantics* (September 1999), no. volume 26 of Electronic Notes in Theoretical Computer Science.

[5] MINAMIDE, Y. Space-profiling semantics of the call-by-value lambda calculus and the CPS transformation. In *The Third International Workshop on Higher Order Operational Techniques in Semantics* (1999), A. D. Gordon and A. M. Pitts, Eds., vol. 26 of *Electronic Notes in Theoretical Computer Science*, Elsevier.

[6] NARLIKAR, G. J., AND BLELLOCH, G. E. Space-efficient scheduling of nested parallelism. *ACM Trans. on Programming Languages and Systems 21*, 1 (1999), 138–173.