Algorithmic λ-Calculus for the Design, Analysis, and Implementation of Parallel Algorithms

Umut Acar, Guy Blelloch, Robert Harper **Carnegie Mellon University**
John Reppy **University of Chicago**

1 Introduction

The early 1930s were bad years for world affairs, but great years for what would eventually be called Computer Science. In 1932, Alonzo Church at Princeton described his λ-calculus as a formal system for mathematical logic [19] and in 1935 argued that any function on the natural numbers that can be effectively computed can be computed with his calculus [20]. Independently in 1935, as a master's student at Cambridge, Alan Turing was developing his machine model of computation and a year later he too argued that his model could compute all computable functions on the natural numbers and showed that his machine and the λ-calculus are equivalent [69]. The fact that two such different models of computation calculate the same functions was solid evidence that they both represented an inherent class of computable functions. From this arose the so-called Church-Turing thesis, and although the thesis by itself is one of the most important ideas in Computer Science, the influence of Church and Turing's models go far beyond the thesis itself.

Turing's machine has become the foundation for the study of algorithms and computational complexity [30]. Hartmanis and Stearns initiated the study of complexity in the early 60s, and this was quickly followed by seminal work of Blum (on an axiomatic theory), Cooke and Levin (on the question of P vs. NP), and Karp (on methodologies for proving problems are NP-complete). There has been and continues to be a huge amount of research in the area, and together the work has led to several Turing awards. All this work has been defined in terms of minor variants of the Turing machine. In the field of Algorithms other models, such at the Random Access Machine (RAM), have been developed that are closer to physical machines. Even these machines, however, were heavily influenced by the Turing machine. The practical influence of the Turing Machine and the models derived from it is huge. It forms the basis of all analysis of algorithms, the basis for the security of many protocols, and the basis for a wide variety of hardness results that have helped guide researchers away from trying to solve problems that are likely impossible.

On the other side, Church's calculus has become the foundation for the theory of programming languages [16]. Dana Scott and Christopher Strachey developed denotational semantics in the late 60s, and this was followed by the seminal work by Per Martin-Löf (on Intuitionistic Type Theory), Robin Milner (on LCF and type theory), and Jean-Yves Girard and John C. Reynolds (on data abstraction and polymorphism). Again, there has been and continues to be a huge amount of research in the area, and together the work has led to several Turing Awards. This work was almost all based on Church's calculus and logical formalism. As with Turing's machine, the theoretical and practical influence of Church's calculus and the models derived from it are huge. The calculus forms the core of mathematically-grounded functional languages such as ML and Haskell, and even plays an increasing role in less mathematically-grounded languages such as Lisp, Rust, Scala, Swift, and even C++ and Java, which have both recently added λ-abstraction. The λ-calculus also forms the core of almost all modern interactive theorem provers, including NuPRL, Coq, Isabelle, and HOL.

Despite the pervasive influence of Turing's machine in the performance analysis of programs and the pervasive influence of Church's calculus in the meaning and correctness of programs, there has been surprisingly little overlap of the work. Indeed, beyond Turing's initial paper, it is hard to find a research paper that even mentions both formalisms. As a consequence, the subfields of the theory of programming languages and the theory of algorithms and complexity seem to be as far apart as just about any two other subfields of computer science.

```
fix mergesort $A$ =
if $|A| \leq 1$ then $A$
else let $(L,R)$ = split($A$)
    in par $L'$ = mergesort($L$) and $R'$ = mergesort($R$)
        in merge($L'$,$R'$)

fix quicksort $A$ =
if $|A| \leq 1$ then $A$
else par $L$ = quicksort($\langle x \in A[0] \mid x < p \rangle$) and $G$ = quicksort($\langle x \in A[0] \mid x > p \rangle$)
    in $L + \langle x \in A[0] \mid x = p \rangle + G$
```

Figure 1: Mergesort and quicksort in λ-alg (extended with sequences). The **par .. and ..** indicates parallel calls, and the merge itself can be parallelized, as well as the comprehensions. With a parallel merge, the mergesort algorithm has work $W(n) = O(n \log n)$ and span $S(n) = O(\log^2 n)$ ($n = |A|$) [9]. Assuming the input is randomly permuted, then quicksort algorithm has work $O(kn \log n)$ and span $O(k \log^2 n)$ with probability $O(1 - 1/n^k)$. The realizability theorem for a Parallel-RAM with P processors should give a mapping to $O(W(n)/P + f(P)S(n))$ time, where $f(P)$ is a function depending on the specifics of the Parallel-RAM.

The limits of the human compiler. One might wonder how is it possible that computer science, especially the more theoretical side, has done so well while having two completely different theories for algorithms and programming languages. A key reason this separation has worked to date is because languages are simple enough that humans can effectively compile code in their head. When shown an algorithm in pseudocode, or even a concrete language such as C or Java, the algorithm designer knows, for example, that the reads and writes to array elements are constant time operations on the RAM, and also understands (at least implicitly) that recursive calls involve pushing some constant size data onto a stack and later popping it (or even that tail recursion does not require this overhead), and that memory allocation can be implemented efficiently on the RAM (at least most of the time, and perhaps ignoring the garbage collector). The mapping from the programming language to the machine model is rather direct.

Unfortunately, with the advent of commodity parallel machines, as well as the advent of commodity programming languages supporting increasingly powerful features (*e.g.*, higher-order functions, dynamic dispatch, automatic memory management) this is no longer possible because in requires the human to understand all these complicated features are compiled, and possibly interact. This is especially true when using fine-grained dynamic parallelism, where tasks can be created on the fly, and scheduler is required to map tasks onto processors. We believe that for parallelism the abstraction level presented by a machine model, such as the Parallel-RAM [44], is too low level for reasoning about and programming parallel algorithms. Therefore, no satisfactory machine model has been developed.

This Proposal. In this proposal we suggest an end-to-end approach to melding the ideas from the programming languages and algorithms community — *i.e.*, the ideas that arose from Church and Turing, respectively. By end-to-end we mean from algorithm design and analysis to efficient running code. Our thesis is that some form of melding of the ideas is necessary for a sound and practical theory of parallel algorithms. Although we also believe that melding the ideas might be useful in understanding other areas of complexity and programming, the focus of our proposal is on the design, analysis and implementation of algorithms, and in particular parallel algorithms — this is where we believe melding the ideas is needed most. For this purpose we propose to use the following framework for our research:

λ-alg. To specify an *abstract* programming language that has a rigorous *behavioral* and *cost* semantics, which defines both what programs do and the (abstract) efficiency with which they do it. By an "abstract" language we mean it should be thought of in a similar way as the RAM (Random Access Machine) model — *i.e.*, not as a particular instruction set such as the x86 instruction set nor as a particular processor family such as the Xeon E7, but as simplified abstract model. The core language-model we propose is the typed λ-calculus with a few extensions and a cost semantics based on call-by-value evaluation. Parallelism is expressed by simple fork-join construct, and costs for the core will be defined in terms of work and span (described in more detail below), as well as space. The language will be designed so that it can be reasonably easily extended with additional features. We refer to it as the Algorithmic λ-calculus, or λ-alg for short. An example of λ-alg is given in Figure 1.

Realizability theorems. To prove *realizability theorems* that relate the abstract cost specified by the high-level language to the concrete cost of more traditional machine models such as the parallel-RAM. These theorems are the part that ties together the two sides of theory. In the discussion above they can be thought of as the glue that removes the need to compile algorithms in one's head to understand how they perform on a machine model. Instead, the algorithm designer understands cost at a more abstract level and is given a translation of these costs into time (or other resources) on a machine model. It is important that the bounds produced by this relationship are useful for analyzing algorithms and comparing their asymptotic performance. As targets of the realizability theorems we will consider machine models with various levels of detail, including models that account for caching.

Resource bounded implementations. To implement a parallel ML dialect that embodies λ-alg and its extensions on shared memory multicore machines. Having realizability theorems for abstract machines is useful for relating the two sides of theory, but ultimately we have the very practical objective of being able to analyze parallel algorithms in an abstract model for the purpose of understanding performance (at least roughly) on real parallel machines. We believe that having an actual implementation that is faithful to the realizability theorem, and identifies impracticalities in the machine models themselves is very important. We fully realize that abstract machines are themselves only rough approximations (sometimes not even that) of real machines. In this implementation we will also be concerned with not just asymptotics, but minimizing constant factors in performance.

Within this framework we plan to first apply the approach to a core λ-alg, and then add various extensions (*e.g.*, arrays, futures, memoization). Each of these extensions will require additions across all three levels.

The PIs have significant experience on the topics mentioned above [9, 12, 36, 66, 28, 37, 13], and indeed as a general framework the idea of cost semantics and provably efficient implementations is not new. However previous work has never been put in a common framework, and more importantly none of the work has been accompanied by a corresponding real implementation that attempts to be faithful to the costs. We believe this framework will allow us (and others) to study many extensions of the core language, as well as considering many variants of the target machine models. We believe the implementation will be important in demonstrating that the approach is not just of theoretical interest, but can lead to a practical methodology for designing parallel algorithms. The key contributions of the proposed work are the following:

Proposed Work and Contributions.

- The definition of a single common core λ-alg model. Although many of the ideas exist, there is effort required to put the pieces in a common framework, to document it, and to formalize it in an interactive proof system such as Coq and/or Twelf.
- Realizability cost bounds for the core λ-alg on abstract (idealized) machine model, and in particular the asynchronous Parallel-RAM. Although some of these proofs has been done in "paper and pencil" we note that not even mapping the sequential part of the λ-calculus onto a sequential RAM, even ignoring memory management, has been done in a formal setting.
- Formal analysis and verification of the costs of the scheduler itself as part of the implementation. Previous work has considered idealized greedy schedulers. Practical schedulers, however, are often

based on work-stealing and we know of no work formally analyzing their cost.

- Building on previous work by the PIs on parallel implementations of ML [27, 66, 62, 28, 6, 2, 61], we will develop an implementation of λ-alg's parallelism model and prove that the implentation is faithful to our asymptotic cost bounds (to the extent that the target machine is faithful to our abstract machine model), but also to achieve practical efficiency — *e.g.*, considering constant factors.
- Extensions to the core λ-alg with a variety of features, including aggregate data types (*e.g.*, arrays, sets, maps), interactive parallelism, and other forms of parallelism such as futures and or-parallelism. These features involve extending both the behavioral and cost semantics, extending the realizability theorems to capture the costs of these extensions, and ultimately extending the compiler and runtime system.
- Extensions of the target machine models for the realizability theorems with more realistic cost assumptions that capture characteristics of modern parallel machines. For example we plan to consider memory hierarchies. Harper and Blelloch have done this for sequential lambda-calculus on a single level cache [13]. Here we propose extending this to the parallel context.
- Validation of the approach by developing a collection of algorithms expressed in the λ-alg (plus extensions). This will include asymptotic analysis of the algorithms in the λ-alg cost model, and where possible proven in a proof system. It will also include empirical measurements of runtime on our implementation on shared memory multicore machines, and validation that the analysis at least approximates measured times in terms of scalability of problem size and processor counts.

In summary the work will involve using λ-alg as a platform for an end-to-end study and testbed to demonstrate how to use language-based cost models for the design, analysis and efficient (asymptotically and practically) implementation of parallel algorithms.

λ-alg is pure, not having any side effects. We believe this is important in the context of parallelism since it guarantees determinism, and means that the behavioral semantics are sequential. Many of the extensions we will include, however, will involve benign effects. For example our work on efficient-arrays will involve effects in the implementation to achieve the required bounds. Similarly the work on memoization will involve effects in the implementation. The goal is to encapsulate the non-determinism due to effects in the implementation so that the language is itself deterministic, while getting the benefits of better performance. Our machine models will necessarily incorporate effects. λ-alg uses call-by-value (strict) instead of call-by-need (lazy) evaluation, as used in, for example, Haskell. We believe call-by-value is a much better evaluation strategy for reasoning about costs, and also is much better suited for parallelism. Indeed most of the effort in generating parallel extensions to Haskell has involved using strict subsets or strictness annotations [53, 52, 59]. It would not be hard to add a lazy application rule to λ-alg.

Although the focus of the proposed work is on research on language-based cost models, another important aspect of the work will be to develop teaching material. The research is motivated in part by a sophomore-level course that Acar, Blelloch and Harper have been teaching at CMU. The course is the introduction to data-structures and algorithms course required for all of our major and taken by about 400 students a year. It teaches parallelism from the start and uses a purely functional style—not using side effects, and makes significant use of higher-order functions. In fact the pseudocode used is effectively λ-alg although without a formal semantics. We plan to continue developing the course with the goal of making the material available to others.

2 The λ-alg Formalism

Church's legacy provides us with a model of computation based on *calculation*, achieving a unification of computation and mathematics that is absent in traditional machine-based programming. Calculation is naturally parallel, for there is no inherent ordering among the sub-calculations in a formula. For example, to evaluate a pair of expressions (e_1, e_2), we can, if we like, evaluate the e_i in either order to obtain their simplified forms v_i, then put them back together to form (v_1, v_2), the evaluated pair. But we can just as well evaluate the e_i's simultaneously to obtain their values, from which the value of the pair can be constructed. Importantly, the values of the two expressions are the same, regardless of whether they are evaluated sequentially or simultaneously. Regardless of the ordering or the simultaneity of computations, the outcome is always the same. In computer science terms the *correctness* of a parallel program is never in doubt, for it behaves the same as any chosen sequential calculation order.

Thus, in Church's framework, parallelism is not about correctness, but rather *efficiency*. Ideally, the more computing resources we have available, the more we can perform calculations simultaneously, and the faster we can compute. The only limitation is the inherent sequentiality of a computation. For example, if we are to compute $\sqrt{(3+2)+(2+2)}$, we can perform the innermost pair of additions simultaneously, but must obviously defer computing the outer addition, or the square root, until those computations complete. Parallel algorithm design is concerned with maximizing such opportunities by minimizing needless sequential dependencies. In previous work [9, 38] we have developed a framework for expressing the efficiency of functional programs using a *cost semantics* that not only defines the outcome of a computation, but also identifies the dependencies among the subcomputations in the form of a *cost graph*. The *work*, or sequential complexity, of a program is the number of nodes (atomic computation steps) in this graph; the *span*, or idealized parallel complexity, is the *depth* or *diameter*, of this graph, the longest chain of sequential dependencies among subcomputation. For example, in the case of evaluating a pair (e_1, e_2), if evaluation of e_i takes w_i steps, then evaluation of the pair takes $w_1 + w_2 + 1$ steps, reflecting the work of constructing the pair. Similarly, if the span of e_i is s_i, then the span of the pair is $\max(s_1, s_2) + 1$, reflecting the independence of the two components, and the dependence of the pair on their results.

The assignment of a cost graph to a computation is a matter of definition; the rules define the cost of evaluation of each form of expression. There is room for variation, and there is room for error, in these definitions. It is therefore important to validate the cost semantics against a variety of lower-level models by showing that it is *realizable* on these models with specified bounds. The paradigmatic result of this kind is Brent's Principle [15], which states that if an expression e has work w and span d, then it can be executed on p processors in time no greater than $w/p + s$ [9, 38]. This bound is at most twice optimal, for we can perform the work in chunks of p at a time, limited only by the span s, which measures the inherent sequentiality of the algorithm. Brent's Principle provides the link between the programmer-level understanding of the complexity of an algorithm and the compiler-level realization of that algorithm on an abstract machine. At this level of abstraction the central implementation problem is scheduling the work onto the processors, and managing the interaction between the work and the scheduler. It is here, and only here, that one must take into account questions of synchronization, and of the efficient use of machine resources. By varying the choice of machine model we may take account of other hardware characteristics such as asynchronous processors [34], memory hierarchies [11], and the nature of the interconnect [49].

Thus, the λ-alg formalism serves a number of purposes for our work. First, it provides a mathematically precise framework for reasoning about the correctness and complexity of parallel algorithms. Having a cost semantics makes it possible for the programmer or algorithm designer to think at the level of the code as written, rather than in terms of an imaginary compilation to a low-level machine model whose complexity is defined by the number of instructions executed. Abstract cost measures, such as cache locality considerations, are readily accommodated at the language level [13], rather than derived from careful analysis of machine characteristics and compilation strategy. Second, the formalism abstracts from the details of any specific programming language, allowing results to be transferred to a variety of settings. To be sure, λ-alg, being derived from the λ-calculus, is most naturally implemented by transliteration into a functional language such as ML [65, 51] or Manticore [28]. But it can also be realized in an imperative language, such as C++, with sufficient care [3]. Third, the formalism supports the concept of a Brent-style *resource-bounded implementation* on a variety of platforms in which we may prove bounds on the complexity of the compiled code as a function of the abstract complexity assigned to it by the cost semantics. Thus, *in toto*, we achieve a rigorous, practical methodology for parallel algorithm design and implementation grounded in a precisely specified semantics for λ-alg. Fourth, such precise specifications are essential for mechanization of proofs of the correctness and complexity of algorithms, and of the resource bounds on an implementation. Given the complexity of the proofs involved, we consider it essential to explore the use of formal proof development systems, such as Coq [21] or Agda [4], to verify informal arguments.

2.1 Formulation of λ-alg

To make these ideas more concrete, let us consider a simple fragment of λ-alg consisting of a purely functional core enriched with a binary parallel evaluation construct presented in the style of *Practical Foundations for Programming Languages* [38]. The syntax is specified by the following definition in which the evident binding and scope conventions are left implicit:

τ	$::=$	$\mathtt{nat}$	natural numbers
		$\tau_1 \rightharpoonup \tau_2$	partial functions
e	$::=$	x	variable
		$\mathtt{z}$	zero
		$\mathtt{s}(e)$	successor
		$\mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\}$	zero test
		$\lambda\, (x : \tau)\, e$	function
		$e_1(e_2)$	application
		$\mathtt{fix}\, x : \tau\, \mathtt{is}\, e$	recursion
		$\mathtt{par}\, x_1 = e_1\, \mathtt{and}\, x_2 = e_2\, \mathtt{in}\, e$	parallel let

The core of the language consists of a base type of natural numbers together with functions between any two types, of arbitrary order. Parallelism is supported by a binding construct for evaluating two expressions simultaneously, and substituting their values into a common “join point” expression.[1]

For the sake of brevity, we omit the static semantics (typing rules) for λ-alg, which is standard for a typed functional language. The dynamic semantics, or just dynamics for short, defines the deterministic process by which expressions are evaluated and assigns an abstract cost to evaluation that defines both the sequential complexity, or *work*, of the computation as the total number of steps and the idealized parallel complexity, or *span*, of the computation as the length of the longest chain of sequential dependencies within the computation. The span may also be thought of as the idealized parallel time complexity in that it abstracts from the number of processing elements required to complete the computation, much as the semantics abstracts from the memory capacity of the processor(s). The ratio of the work to the span defines the *parallelizability* of the computation. Intuitively, if the span is small, then the work can be done in parallel, whereas if the span is proportional to the work, there is little scope for parallelism. This ratio is a key figure of merit for algorithm design.

The costs assigned to evaluations are convenient presented by *cost graphs*, which in the present case are a class of series-parallel graphs representing the *dynamic* dependency structure of the evaluation as it unfolds. Cost graphs are defined by the grammar

$$c ::= \mathbf{0} \mid \mathbf{1} \mid c_1 \otimes c_2 \mid c_1 \oplus c_2,$$

representing the unit and zero costs, parallel and sequential combination of costs. Costs may be thought of as series-parallel dag’s in which the nodes represent units of work. Accordingly, the work, $W(c)$, of a cost graph, c, is defined to be its size, and the span, $S(c)$, is defined to be its depth or diameter.

[1]There are many alternative formulations that are substantially equivalent. See below for a brief discussion of n-ary parallelism derived from operations on sequences.

$$\frac{e \Downarrow^{c} \mathtt{z} \quad e_0 \Downarrow^{c_0} v}{\mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\} \Downarrow^{c \oplus c_0 \oplus \mathbf{1}} v} \qquad \frac{e \Downarrow^{c} \mathtt{s}(\overline{n}) \quad [\overline{n}/x]e_1 \Downarrow^{c_1} v_1}{\mathtt{ifz}\, e\, \{\mathtt{z} \hookrightarrow e_0 \mid \mathtt{s}(x) \hookrightarrow e_1\} \Downarrow^{c \oplus c_1 \oplus \mathbf{1}} v} \tag{1a}$$

$$\frac{}{\lambda\,(x:\tau_1)\, e_2 \Downarrow^{\mathbf{1}} \lambda\,(x:\tau_1)\, e_2} \qquad \frac{e_1 \Downarrow^{c_1} \lambda\,(x:\tau_2)\, e \quad e_2 \Downarrow^{c_2} v_2 \quad [v_2/x]e \Downarrow^{c} v}{e_1(e_2) \Downarrow^{(c_1 \otimes c_2) \oplus c \oplus \mathbf{1}} v} \tag{1b}$$

$$\frac{[\mathtt{fix}\, x:\tau\, \mathtt{is}\, e/x]e \Downarrow^{c} v}{\mathtt{fix}\, x:\tau\, \mathtt{is}\, e \Downarrow^{c \oplus \mathbf{1}} v} \qquad \frac{e_1 \Downarrow^{c_1} v_1 \quad e_2 \Downarrow^{c_2} v_2 \quad [v_1, v_2/x_1, x_2]e \Downarrow^{c} v}{\mathtt{par}\, x_1 = e_1\, \mathtt{and}\, x_2 = e_2\, \mathtt{in}\, e \Downarrow^{(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c} v} \tag{1c}$$

Figure 2: Dynamics of λ-alg (Selected Rules)

The cost dynamics of λ-alg is given in Figure 2. The rules therein define the *evaluation judgment* $e \Downarrow^c v$ stating that the (closed) expression e evaluates to the value v with cost c. The cost dynamics specifies evaluation in the usual way for a call-by-value (strict) functional language, but enriched with a special construct for parallel binding of two expressions for use within another. For the sake of illustration, the rule for evaluating function applications specifies that the function and argument are evaluated in parallel, and then the result of substituting the argument into the function is evaluated. The rule for parallel binding specifies that the two bound expressions are evaluated in parallel, and their values are substituted into the scope of the binding, which therefore depends on the values of the bindings. In either case the resulting cost graph is $(c_1 \otimes c_2) \oplus \mathbf{1} \oplus c$, which uses the graph combinators to express both its parallel and sequential aspects. Specifically, the product represents parallelism, the sum represents sequentiality, and the unit cost represents the step of substitution (binding values to variables).

As an example, consider a simple recursive algorithm for computing the Fibonacci numbers. Although this function is not an efficient way to compute Fibonacci numbers, it is traditionally used to illustrate a simple parallel computation. We can write this function in λ-alg (taking small liberties with operations on the natural numbers) as shown in Figure 3. Figure 4 shows a slightly simplified dag for an instance of the function `fib` with argument $n = 3$. In the figure, vertices are labeled with the "steps" of the computation that they correspond to. For example, the vertex labeled by `fib(2)` calls to a call to `fib` with argument `2`, and the vertex labeled by `1+1` correspond to the step computing the sum `1+1`. The edges between the vertices represent the sequential control dependencies between the corresponding computations. For, example vertices with out-degree two "fork" two parallel computations. Vertices with in-degree two "join" two parallel computations; a join vertex synchronizes its two in-neighbors by waiting for both of them to complete before executing. Such fork and join edges correspond to the `par` construct in the cost semantics of λ-alg.

2.2 Realizability of the λ-alg Semantics

The cost graph assignment given by the dynamics provides the foundation for analyzing the complexity of programs written in λ-alg. The programmer needs nothing more than the cost dynamics to derive asymptotic bounds on the efficiency of his or her program. Importantly, no "hand compilation" is required, nor any implicit understanding of how the language is implemented. On the other hand the association of cost graphs to evaluations is motivated by the intended implementation of the language. To ensure that the costs are

```
fix fib n =
    if n <= 1 then n
    else
      par x = fib (n - 1)
      and y = fib (n - 2)
      in x + y
```

Figure 3: Code for parallel Fibonacci.

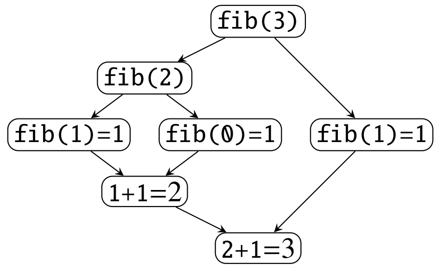

Figure 4: A dag representation of `fib(3)`.

$$\frac{e_1 \longmapsto e_1' \quad e_2 \longmapsto e_2'}{e_1(e_2) \longmapsto e_1'(e_2')} \qquad \frac{}{(\lambda\,(x : \tau_2)\, e)(v_2) \longmapsto [v_2/x]e} \tag{2a}$$

$$\frac{e_1 \longmapsto e_1' \quad e_2 \longmapsto e_2'}{\texttt{par}\, e_1 = e_2\, \texttt{and}\, x_1 = x_2\, \texttt{in}\, e \longmapsto \texttt{par}\, e_1' = e_2'\, \texttt{and}\, x_1 = x_2\, \texttt{in}\, e} \tag{2b}$$

$$\frac{}{\texttt{par}\, v_1 = v_2\, \texttt{and}\, x_1 = x_2\, \texttt{in}\, e \longmapsto [v_1/v_2]x_1 x_2 e} \tag{2c}$$

Figure 5: Parallel Transition Dynamics of λ-alg (Selected Rules)

sensible, they must be validated by showing that they can be *realized* on an (idealized) computer.

To do so we model a computer by a state transition system whose transitions correspond to abstract machine instructions that alter the state. There are many choices for the transition system, according to what aspects of the computation are to be made salient. For the purposes of asymptotic analysis of the work and span, it suffices to consider a transition system whose states are expressions and whose steps correspond to the simultaneous execution of any number of primitive calculation steps. Such a transition system is given in Figure 5. The cost dynamics given above is validated by the following realization theorem relating the span of the cost graph to the number of transitions of the idealized computer implementing the λ-alg instruction set.

Theorem 1 (Span Validation)**.** *$e \Downarrow^c v$ with $S(c) = s$ iff $e \longmapsto^s v$.*

The next step of the realizability is to map the transition system onto an abstract machine such as the asynchronous Parallel-RAM. This requires simulating steps of the transition system on RAM style processors, and accounting for the costs of the scheduler, and the memory manager. This will serve as the bridge between the semantics and the resource bounded implementation.

Numerous extensions of and variations on λ-alg are possible. For example, it may be extended with sum, product, and recursive types to model data structures such as trees or lists, a necessity for practical programming.[2] It may also be extended with exceptions, or even with a command layer to admit computational effects [38]. Furthermore, one may also consider notions of cost capturing other aspects of a computation, such as its cache locality behavior [13], a well-known difficulty for any abstract programming language, sequential or parallel. Such models must be shown to be realizable with respect to a lower-level machine model, following along the lines of Brent's seminal approach.

3 Resource-Bounded Implementation

To achieve our goal of end-to-end analyzability of parallel programs, we shall develop an embodiment of the λ-alg calculus and its extensions as a functional programming language. Rather than building a new parallel language implementation from scratch, which is a substantial effort, we plan to leverage the PI's previous work on parallel implementations of ML, which has so far taken place in the context of the Manticore system [27, 62, 28, 8, 6], and extensions to the high-performance MLton compiler [55, 66, 2, 61]. Both of these systems provide implementations of a parallel dialect of Standard ML, which we call Parallel ML (PML). Parallel ML supports a superset of the core λ-alg features.

While they remained mostly separate projects until the last couple of years, these two systems have been converging. For example, the Manticore project has started using the MLton front-end instead in its compiler. We plan to continue this effort of convergence to produce one polished implementation that can be used at our institutions and elsewhere in research and education.

[2]One important extension, with finite sequences, is discussed in Section 4.

In the rest of this section, we briefly describe aspects of the Manticore implementation that are relevant to the proposed work, and then describe a plan for verifying that the compiler realizes the λ-alg cost semantics. We believe that these techniques are general enough to apply to other compilers with similar architecture, such as the MLton compiler.

The PML compiler is organized as a series of phases, each of which is defined by a distinct intermediate representation (IR). Of particular importance is the BOM IR, which is a monomorphic core-ML extended with first-class continuations, mutable data structures, atomic memory operations, and a processor abstraction. In addition to being a representation suitable for a host of optimizations, the PML compiler also supports importing code written in BOM. We use this mechanism to define the machine-level representations of primitive types and operations (*e.g.*, integer arithmetic) and as the language for programming many aspects of the PML runtime system, such as thread creation, thread scheduling, and work stealing [7]. When a PML program is compiled, the BOM code that implements the Basis Library and runtime system are loaded and combined with the BOM code generated from the source program. In addition to enabling cross-layer optimizations, this process makes the scheduling policy *manifest* in the intermediate program. Thus, a cost semantics for BOM can be used to reason about both the cost of the user's program and the cost of the implementation's runtime.

After optimization, the BOM IR is first CPS converted to our CPS IR,[3] and then closure-converted to a first-order control-flow-graph (CFG) IR. We then generate assembly code from the CFG IR using either the MLRISC framework [33, 32] or, more recently, LLVM [24]. The assembly code is linked with a small runtime system written in C that implements a processor abstraction layer on top of pthreads, the garbage collector, and an interface to operating system services.

To connect the λ-alg cost model to our implementation will require first giving a cost semantics to the BOM IR. Since this IR is basically an extended λ-calculus (similar in some ways to ANF [26]), developing a cost semantics for single-threaded BOM code should be similar to that developed for λ-alg. Beyond this cost semantics for sequential BOM code, we need to account for multiple processors and interprocessor coordination and communication.

Once we have related the cost semantics of λ-alg to a realization in BOM, we need to also verify that the rest of the compiler pipeline is faithful to the BOM cost semantics. This process will involve showing that for each step of the compiler pipeline, the transformations only introduce constant overhead (*e.g.*, converting a nested function into a first-order function with an explicit heap-allocated closure requires allocating the closure, which has cost bounded by the number of live variables for the function). We also must show that runtime system costs, such as garbage collection, can be accounted for by the original cost model (by amortization arguments).

The theoretical work on establishing the faithfulness of the cost of the compilation process with respect to the cost semantics will primarily involve asymptotic analysis, possibly ignoring important constant factors. We shall, however, aim to create a high-performance, scalable, state-of-the art parallel language implementation. Prior work on Manticore has shown that it is possible to build very scalable implementations of parallel functional languages, and work on the MLton compiler has demonstrated that functional language implementations can be competitive with traditional imperative languages for many applications. We specifically identify the following implementation issues that have a direct impact on scalability and performance.

- Garbage collection is a potential sequential bottleneck and a significant source of runtime overhead. We have demonstrated scalable parallel garbage collection techniques [5, 61] for PML that we believe can be further improved. The resulting garbage collection techniques will have to be shown to impose

[3] "CPS" stands for *Continuation-Passing Style* and is a representation that makes continuations explicit. This property is particularly useful for concurrent and parallel language implementations, since continuations provide a natural representation of suspended threads.

$$\frac{e_0 \Downarrow^{c_0} v_0 \quad \ldots \quad e_{n-1} \Downarrow^{c_{n-1}} v_{n-1}}{[e_0,\ldots,e_{n-1}] \Downarrow^{\bigotimes_{i=0}^{n-1} c_i} [v_0,\ldots,v_{n-1}]} \qquad \frac{e \Downarrow^{c} [v_0,\ldots,v_{n-1}]}{|e| \Downarrow^{c \oplus \mathbf{1}} \overline{n}} \tag{3a}$$

$$\frac{e_1 \Downarrow^{c_1} [v_0,\ldots,v_{n-1}] \quad e_2 \Downarrow^{c_2} \overline{i} \quad (0 \leq i < n)}{e_1[e_2] \Downarrow^{c_1 \oplus c_2 \oplus \mathbf{1}} v_i} \tag{3b}$$

$$\frac{e_2 \Downarrow^{c} \overline{n} \quad [\overline{0}/x]e_1 \Downarrow^{c_0} v_0 \quad \ldots \quad [\overline{n-1}/x]e_1 \Downarrow^{c_{n-1}} v_{n-1}}{\texttt{tab}(x.e_1; e_2) \Downarrow^{c \oplus \bigotimes_{i=0}^{n-1} c_i} [v_0,\ldots,v_{n-1}]} \tag{3c}$$

Figure 6: Extension of λ-alg With Sequences

only constant-time overhead.

- Related to garbage collection are techniques used to improve data locality and reduce the overhead from NUMA architectures. This is an area where the semantics of ML allows implementations significant freedom to relocate, replicate, and reorganize data to improve locality.
- Efficient parallel implementations require effective scheduling of work and dynamic load balancing. We have explored various advanced techniques for scheduling and work stealing [27, 73, 8, 3], but we have not yet tried integrating them into a cost model.

Based on the outcome of our empirical analysis, we also plan to augment our cost models to take into account some of these costs. A particularly interesting extension would be to extend the cost semantics with information about allocated locations to account for data locality.

4 Data Parallelism via Aggregates

Aggregates such as sequences, sets, maps and graphs form a core component of programming parallel algorithms. Any aggregate data type can be built on top of the core λ-alg using recursive types in a way that is reasonably suited for parallelism—e.g. using balanced trees (instead of lists). This is perfectly sufficient for many purposes, at least asymptotically—for example map, reduce or filter on sequences can all be implemented with linear work and logarithmic span (assuming the function being passed is constant work). However, for some algorithms using recursive types will involve a $O(\log n)$ loss in efficiency. In particular this is the case if the algorithm requires random access into a sequence. For this purpose we plan to include array-sequences as an extension to the core λ-alg, and extend the behavioral and cost semantics appropriately. We will also add some syntactic sugar for sequence comprehensions (as used in Figure 1).

The first kind of sequence we will add is a purely functional sequence, allowing constant-work random-access reads from the sequence. We refer to these as array sequences. The extension of λ-alg with array sequences is given in Figure 6. The extension just adds five functions (forms), one for creating a sequence, one for taking the length, one for accessing an element, one for grabbing a subsequence (not shown in the rules), and a tabulate function (`tab`).

Although array sequences are sufficient for implementing many other functions efficiently (e.g. map, reduce, zip), and are also sufficient to implement a variety of algorithms (including several graph algorithms) efficiently, they do not supply an efficient way to update the sequence. Updating effectively requires copying the full sequence and therefore takes linear work (although constant span). In many algorithms it is necessary to update a sequence repeatedly. A good example of this is in the implementation of depth-first-search on a graph. This is an algorithm that can be implemented in linear work in the standard RAM model. The problem with achieving this bound in λ-alg extended with just array sequences is that maintaining the set of vertices that have been visited requires an update each time a vertex is visited. Using trees to represent the set requires $O(\log n)$ work per update, whereas array sequences require linear work per update (i.e. in this case trees are better, but still not as good as for mutable arrays). This problem leads to a logarithmic loss in efficiency for a variety of problems including generating a random permutation, finding biconnected components, topological sort, and cycle detection on graphs. We will refer to this as the array update problem.

Threaded Aggregate types

A variety of approaches have been suggested to alleviate the array-update problem. Many of the approaches are based on the observation that if there is only a single reference to an array, it can be safely updated in place. The approaches include using monads [56, 75], linear types [35, 74], or reference counting [40, 39]. The problem with all these approaches is they do not work well with parallelism. In fact monads and linear types require the program's use of the array to be single threaded. They also require new language features that are only there for helping with efficiency and not at all necessary for the behavioral semantics (pure arrays are perfectly sufficient for behavior). Reference counting can in theory be used with parallelism, but it is hard to reason about efficiency since a count could depend on scheduling order.

Another class of approaches is to fully support functional arrays, even with sharing, but using more sophisticated data structures. This is sometimes referred to as version tree arrays [1] or fully persistent arrays [23]. Previous suggestions for such implementations, however, do not support parallelism [18, 58].

To deal with the array update problem we plan to extend λ-alg with threaded aggregate types which are motivated by version trees, but also safe for parallelism. In ongoing work [10] we are studying such an approach. Here we briefly describe how it might be included in λ-alg as an extension, how the realizability theorem can work, and how to make a resource bounded implementation

Extending λ-alg. The extension to the core language will behaviorally act just like pure arrays with absolutely no restrictions on how the arrays are used (i.e. they can be passed to two parallel tasks). They will have the same behavioral semantics as given in Figure 6 but will also support an update function that updates a single location (or possibly multiple locations). The cost semantics, however will be different. The idea is to extend them to keep track of whether an array is a leaf or interior node in the version tree. This requires maintaining a store in the cost-semantics that maps an array to whether it is interior or a leaf. The cost of reading and updating an array depends on this state, and in particular is constant work for reading or updating a leaf array. The innovative part of the approach is how the store is maintained when making parallel calls. Instead of forcing single threadeness, it makes a copy of the store in both branches. These copies are then merged when the parallel branches join.

Extending realizability. To realize the cost bounds on a parallel machine model we have to implement a persistent data structure for the arrays. Then to account for different interleavings of the machine threads, which, as it turns out, makes a difference to the internal structures of the persistent arrays, we need to consider all possible interleavings and take the worst case. This means the realizability bound is a worst-case bound, and in fact the bounds could be better for a particular run.

Extending the resource bounded implementation. As part of the research we plan to implement the approach as part of the runtime system. The approach is unlikely to require many if any changes to the compiler itself. Beyond just arrays we plan to extend the idea to other aggregate types such as maps and sets. In particular we believe, for example, that we can implement sets using hash tables allowing for constant time reads and updates. As with arrays this requires some form of version tree since in a persistent setting we need to be able to access old versions. Finally we plan to run a broad set of experiments that analyze how efficient these approaches are compared to more standard imperative style programming.

5 Interactive Parallelism

Since shared-memory parallelism became mainstream in the mid 2000s, there has been much work on developing programming languages, scheduling algorithms and runtime systems to support parallel computing

```
fix helloFib n i =
let
  fix hello i =
    if i <= 0 then bg ()
    else
    let
      _ = output(‘‘Last name:’’)
      x = input ()
      _ = output(‘‘ First name:’’)
      y = input ()
      _ = output(‘‘Hello ’’ ^ x ^ ‘‘ ‘‘ ^ y)
    in hello (i − 1)
in  par r = fib n
    and x = fg( hello i )
```

Figure 7: A simple parallel interactive program.

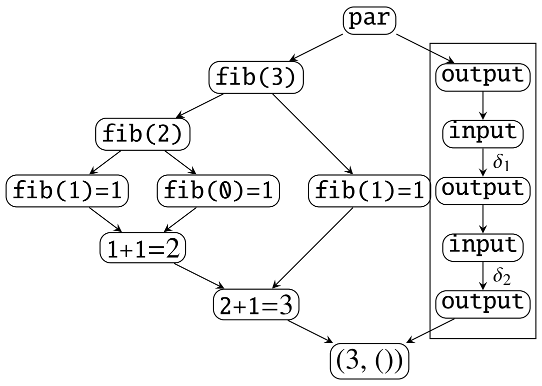

Figure 8: The dag of `helloFib` 3 1.

(e.g., [31, 47, 50, 42, 43, 17, 28, 46]). The focus of nearly all of this work has been on minimizing completion time in compute-intensive applications such as sorting routines, matrix operations, scientific simulations (e.g., the Barnes-Hut algorithm), etc. The common feature of these applications is that they perform a large amount of computation on given data with little interaction with the world, for example with a user or another piece of software. *Throughput*, usually measured as the number of operations in unit time, has been used as the key measure of performance in such applications.

In this project, we will extend λ-alg to support *interactive parallel* programs that communicate with the external world as they perform compute-intensive tasks. Example interactive parallel applications include web servers, games, and similar interactive systems. Such interactive parallel applications mix compute-intensive tasks with user interaction, e.g., a game must interact responsively as it also ensures that compute-intensive calculations proceed as quickly as possible. As a simple illustrative example, consider the program shown in Figure 7. For now, please ignore the underlined keywords *bg* and *fg*. The function `hello` interacts with the user in a very simple way by requesting the user's first and last names and greeting them *i* times. While it interacts with the user, the function also computes the Fibonacci of *n*, which exemplifies a relatively large parallel computation. Consider compiling and executing this program in a traditional parallel programming system such as one that uses work stealing with $n = 42$ and $i = 20$. Since such systems do not give priority to the interactive function `hello`, it starves as the many fine-grained threads created by the call to `fib 42` take over the available processors (we confirmed this by implementing the example in a system based on work stealing). Following the traditional recipe for writing interactive code [64], we could instead dedicate a separate pthread to the interactive `hello` function and trust that the OS will do the right thing, but it might not. It is thus not clear how such an approach would scale to larger programs involving many more interactive and parallel tasks, without controlling the scheduling of interactive threads.

Extending λ-alg: prioritized computations. To enable parallel interaction, we extend λ-alg with basic input-output constructs and with two simple staging constructs: background and foreground, written in code as `bg` and `fg` respectively. These constructs, which correspond to `prev` and `next` from Linear Temporal Logic [22, 60], can then be interpreted by the semantics to assign a priority to computations. For example, we can say that `bg` *e* runs in the *background*, i.e., with low priority. Symmetrically, the expression `fg` *e* runs in the *foreground*, i.e., with high priority. Such an interpretation corresponds to the “two worlds” interpretation of the linear temporal logic [25], but with the background computations running asynchronously with, rather than strictly after, the foreground ones. More generally, we can interpret the same two forms to allow for many priorities. For simplicity, here, we assume just two levels of priority.

We can extend the type system for λ-alg with rules akin to that of Davies [22]. The two key rules are the following:

$$\frac{\Gamma \vdash_\Sigma e : \tau@\mathbb{B}}{\Gamma \vdash_\Sigma \mathtt{bg}(e) : \bigcirc\, \tau@\mathbb{F}} \bigcirc - I \qquad \frac{\Gamma \vdash_\Sigma e : \bigcirc\, \tau@\mathbb{F}}{\Gamma \vdash_\Sigma \mathtt{fg}(e) : \tau@\mathbb{B}} \bigcirc - E$$

The judgment $\Gamma \vdash_\Sigma e : \tau@w$ indicates that expression e has type τ in world w (which may be foreground $\mathbb{F}$ or background $\mathbb{B}$). Most expressions type at either world. The two rules shown transition between the two worlds. A background expression that types at the "background world" (corresponding to stage 2 of LTL) is ascribed the type $\bigcirc\, \tau$ using rule $\bigcirc - I$. When desired, the result of a background computation may be demanded by rule $\bigcirc - E$, which types its argument in the foreground and the resulting expression in the background. Using this type system, we can prevent at compile time *priority inversion*, which can lead to a high-priority computation waiting for a low-priority one. For example, in Figure 7, the call to function `hello` is designated to be a foreground computation while the rest, including the call to `fib 42`, is designated to be background computation.

Having established the type system, we extend the cost semantics of λ-alg to mark the priority of tasks in the cost dag, as well as the delay that each interaction with the user, via `input` and `output`, incurs by including an edge weight for the out-edges of the corresponding vertices in the dag. For example, Figure 8 illustrates the dag for `helloFib` 3 1. The high-priority interaction task, illustrated by a rectangle around the vertices representing the run of the function `fg(hello 1)`, performs an output (to prompt the user with the question) and input for each question, followed by a final output. The edge weights δ_1 and δ_2 stand for the latency incurred by the two input instructions.

Consider now a dag with just two levels of priorities, foreground and background. As usual, we define the work W of a dag as the total number of vertices in the dag and the span S as the longest weighted path in the dag. Note that the latencies of input instructions, represented by edge weights in the dag, do not factor into the work but are included in the span. In addition, define the *foreground dag* as the dag induced by the foreground vertices (these are the subdags marked in rectangles). Define the *foreground work* W° and the *foreground span* S° as the total sum of the work and span of components of the foreground dag. Furthermore, define the *foreground width* of the dag as the maximum number of foreground components.

Extending realizability: prompt scheduling. To minimize *response time*, defined as the total sum of the latency of all foreground vertices, we introduce a *prompt schedule* as a schedule where vertices are assigned greedily to processors in priority order.

Theorem 2 (**Prompt Scheduling**). *Consider a cost dag generated from our extended λ-alg with work W span S, foreground work W° and foreground span S°. Let D be the foreground width of the dag. The following hold for the total computation time and the response time with a prompt schedule:*

1. *the response time is at most $D\frac{W^\circ}{P} + S^\circ$, and*
2. *the total completion time is at most $\frac{W}{P} + S$.*

The theorem bounds the total response time only in terms of foreground computations without penalizing the total completion time, which stays the same as in Brent's original bound. The basic idea behind the proof of this theorem is to generalize the proof of the greedy schedules to account for the latency incurred by all foreground and background vertices and bound this quantity by using an accounting argument. We note that this bound is made possible by the fact that the type system ensures that the program has no priority inversions. To establish realizability, we will formalize a suitable transition system for the interactive parallel language and show that it validates the cost semantics.

Extending Resource-Bounded Implementation and Evaluation. To give a resource-bounded implementation, we will develop a scheduling algorithm for realizing Theorem 2. Our starting point will be prior

```
fix subsetSum($S, k$) =
  let fix SS($i, j$) =
          if ($j = 0$) then true
          else if ($i = 0$) then false
          else if ($S[i-1] > j$) then SS($i-1, j$)
          else par a = SS($i-1, j-S[i-1]$) and b = SS($i-1, j$)
               in a or b
  in SS($|S|, k$)
```

Figure 9: A parallel algorithm for the subset-sum problem—i.e. given a sequence of integers S determine if any subset of them add up to k. Making use of sharing this requires $O(|S|k)$ work and $O(|S|)$ span. Without sharing it requires work that is exponential in $|S|$, although still $O(|S|)$ span.

work on extending decentralized work stealing schedulers with priorities [76, 77, 41]. Based on this and our recent work [3], we developed and implemented a preemptive priority-based work-stealing algorithm that our initial investigations show to be a promising starting point. The main difference between our algorithm and the prior work is that ours is preemptive. Preemption seems necessary for guaranteeing responsiveness because otherwise lower priority threads can starve higher-priority ones for processors. Our implementation of this algorithm interrupts processors every 100ms to participate in a preemptive scheduling step. We tested our algorithm with a handful of parallel benchmarks, including a simple web server that responds to HTTP requests in the foreground while performing a compute-intensive parallel computation (simulated by computing the 42^{nd} Fibonacci number) in the background. A video of the web server's interactivity using our scheduler is available. [4] We have observed that our implementation performs well and stays responsive even in the presence of large low-priority computations. We also observed that standard work stealing without priorities performs poorly; this is not surprising because non-preemptive scheduling allows background threads to starve foreground ones. In this project, we will develop an empirical framework for evaluating interactive parallel software quantitatively by empirically analyzing response time of subcomputations at different levels of priority.

6 Futures and Dynamic Programming

We plan to augment the λ-alg, realizability results, and implementations with futures (speculative evaluation) and features for supporting dynamic programming effectively in parallel. Since futures have been discussed in previous research [36, 29], we focus in this section on proposed research on dynamic programming. We note, however, that even incorporating futures into the framework will require significant work since the original realizability theorems are for simplified machine models, and there was no previous work on actually implementing these ideas in a provably efficient manner.

Dynamic Programming

Dynamic programming is a powerful technique for solving a variety of algorithms. Furthermore the dynamic programming solutions for most problems are inherently parallel. Unfortunately the expression of such dynamic programs in a nested parallel style, is not particularly elegant, and forces synchronizations that are not inherent in the dependence structure. The most natural way to express dynamic programs is using the inherent recursive structure. For example, Figure 9 gives λ-alg code for solving the subset sum problem. In the core language this code requires work that is exponential in the size of the input sequence S. In particular it does not express the sharing of recursive calls. Implementing memoization on top of λ-alg although sufficient for a sequential version, is not efficient for parallelism since it requires single threading the memo table. We plan to extend the λ-alg so that the sharing can be captured by the cost semantics and implemented

[4] http://www.cs.cmu.edu/~smuller/pd-prio-25.ogv

efficiently as part of the realizability theorem and resource bounded implementation. The language will remain behaviorally pure.

Extending λ-alg. To extend the core language we plan to add a form to the language to indicate functions that should be memoized. The form will require an equality function and a hash function on the arguments (e.g. the syntax, in the example might be `fixmem SS`(i, j) `(eqIntPair, hashIntPair)`). The cost semantics then needs to be augmented with a store for mapping inputs to computed instances of the function. The parallel composition of DAGs will then only use a single subgraph for each distinct call, with a dependence to all calls that use it.

Extending the realizability theorem. To extend realizability theorem we have to show that the approach can be mapped onto the machine model efficiently. We believe this can be done by using a concurrent data structure for the memo table. The idea is when invoking a call to the memoized function (e.g. SS) to check if it has been called before on the given argument—which can happen concurrently for many calls. There are three possibilities: it has been called before and the value already calculated, it has been called before by another task and in the process of being calculated, and it has not been called before. The first case is easy to handle. The third is also relatively easy to handle as long as taking ownership is done atomically. The second requires suspending the task to be resumed later when the value is calculated. This is somewhat similar to what is required with futures.

Extending the bounded implementation. This will require efficiently implementing the ideas in the realizability theorem and will need to consider specific concurrency mechanisms, such as a compare-and-swap, as well as specific methods for suspending and reactivating tasks.

7 Verification

The λ-alg framework described in Section 2 provides a mathematically precise description of the behavior and complexity of parallel programs using a rigorously defined cost semantics. Besides providing a precise definition of λ-alg, the cost semantics is sufficiently precise to be directly amenable to formalization in a proof checkers such as Twelf [72], Agda [4], or Coq [21]. Doing so opens the way to using a proof assistant to verify the correctness and complexity of particular algorithms, of properties of λ-alg itself such as the realizability theorem stated in Section 2, and of the correctness and complexity of compilers and run-time systems for λ-alg.

The Twelf proof checker is a well-established tool for verification of the safety of extensions to λ-alg. In previous work one of the PI's (Harper) has, with collaborators, mechanized the proof of safety of the Standard ML language using Twelf [48]. That effort has taught us how to define languages in such a way that mechanized proof of their safety (internal coherence) is relatively routine using Twelf. We expect to replicate those accomplishments as a baseline standard for the proposed work outlined in this proposal. That work, however, does not address the question of validating the cost semantics for λ-alg. We propose to carry out a mechanized proof of the realizability theorem stated in Section 2, which validates the cost semantics relative to a high-level abstract machine. Stronger forms of the realizability theorem are needed to establish Brent-type bounds on the implementation of λ-alg. To do so would require formalization of PRAM-like models of computation and of the translation from λ-alg into such machine models, coupled with a proof of a Brent-type theorem relating the source to target code.

There is a significant gap between the high-level description of a scheduling strategy and the actual implementation of the scheduler as part of the run-time. For instance, in Blumofe and Leiserson's analysis of the work-stealing scheduler used in Cilk [14], the algorithm is described as atomically manipulating nodes of a computation graph. The nodes represent discrete steps of the source program, and the edges between them are dependencies. Hardware threads randomly "steal" pending nodes from other threads when they finish all of their own work. Even at this level of abstraction, the complexity analysis is rather difficult. And in a real implementation [31], these scheduling operations actually involve pushing and popping frame pointers from a lock-free double-ended queue. Subsequent work has developed even more efficient double-ended queues [57, 54] in which expensive memory barrier instructions are avoided. The correctness of these optimizations involves subtle arguments about the possible effects of hardware re-ordering of memory operations. Given the complexity of these implementations, it is not obvious that they correctly implement the high-level description used in the theoretical analysis.

We propose bridging this gap by giving a machine-checked proof of correctness and running time analysis for such a work-stealing scheduler. We believe that such a proof is feasible in light of recent work, including our own, on developing and using program logics to verify fine-grained concurrent data structures [45, 70, 68], while taking into account the effects of weak memory [71, 67] . Because the performance analysis of a work-stealing scheduler involves probabilistic analysis of the random choices threads make about where to steal from, we would need to extend these program logics to incorporate probabilistic reasoning.

More generally, many parallel algorithms use randomization to ensure good bounds on the work in expectation and on the span with high probability. It would be interesting to explore the mechanization of proofs of even simple randomized algorithms, because to do so requires the development of a body of discrete probability theory and numeric reasoning that we expect would challenge even full-scale proof development systems such as Coq. Doing so would increase our confidence in proofs of efficiency that have proved notoriously difficult to get right.

8 Experimental Validation

Although the theoretical aspects of our realization bounds will be validated through verification, the practical aspects of the performance of our compiler will be validated through experimentation. We feel this is important since the theoretical bounds will most likely be asymptotic, and if any constants are included they will likely be sloppy. Also the target machine models we use will not exactly model real machines—which are far to complicated to model precisely. We plan to develop a set of benchmark code in λ-alg. To do this we will leverage the significant collection of benchmarks we already have as part of our work on the course mentioned below ("2-10"), and as part of our development of Manticore [28], and parallel MLton [66]. We will convert the benchmarks to λ-alg and measure their performance on reasonably large multicore machines (we already have one with 64 cores, and plan to get larger ones as they become available). We note that one of the PIs also has a significant ongoing effort developing a benchmark suite of algorithmic problems [63] (currently all coded in C++ with Cilk). We will use the results from the experiments to improve our compiler and help identify any potential problems with our methodology (e.g. highly inaccurate assumptions about the machine). Assuming the timings are good, the results will also be used as a proof of concept of our approach.

References

[1] Annika Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988. 11

[2] Umut A. Acar, Guy Blelloch, Matthew Fluet, Stefan K. Muller, and Ram Raghunathan. Coupling memory and computation for locality management. In *Summit on Advances in Programming Languages (SNAPL)*, 2015. 4, 8

[3] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2013. 5, 10, 14

[4] The Agda programming language. 6, 15

[5] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John H. Reppy. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness (MSPC)*, pages 51–57, 2011. 9

[6] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. Data-only flattening for nested data parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '13)*, pages 90–106, New York, NY, February 2013. ACM. 4, 8

[7] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Lazy tree splitting. In *ICFP 2010*, pages 93–104. ACM Press, September 2010. 9

[8] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Lazy tree splitting. *Journal of Functional Programming*, 22(4-5):382–438, September 2012. 8, 10

[9] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 226–237, 1995. 2, 3, 5

[10] Guy Blelloch, Robert Harper, and Ananya Kumar. Parallel functional arrays. In *Proceedingd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017. 11

[11] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, 2011. 5

[12] 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. 3

[13] Guy E. Blelloch and Robert Harper. Cache and i/o efficent functional algorithms. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 39–50, 2013. 3, 4, 5, 8

[14] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. 15

[15] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974. 5

[16] F. Cardone and J. R. Hindley. λ-calculus and combinators in the 20th century. In D. M. Gabbay and J. Woods, editors, *Logic from Russell to Church*, Handbook of the History of Logic. North-Holland, 2009. 1

[17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005. 12

[18] Tyng-Ruey Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In *Symposium Proceedings on 4th European Symposium on Programming*, ESOP'92, pages 110–129, London, UK, UK, 1992. Springer-Verlag. 11

[19] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, April 1932. 1

[20] Alonzo Church. An unsolveable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936. (Presented April 19, 1935). 1

[21] The Coq proof assistant. 6, 15

[22] Rowan Davies. A temporal-logic approach to binding-time analysis. In *LICS*, pages 184–195, 1996. 12, 13

[23] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1), 1989. 11

[24] Kavon Farvardin and John Reppy. Compiling with continuations and LLVM. In *ACM SIGPLAN Workshop on ML*, September 2016. 9

[25] Nicolas Feltman, Carlo Angiuli, Umut A. Acar, and Kayvon Fatahalian. Automatically splitting a two-stage lambda calculus. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 255–281, 2016. 12

[26] Cormac Flanagan, Amr Sabry, Bruce Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 237–247, 1993. 9

[27] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2008. 4, 8, 10

[28] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011. 3, 4, 5, 8, 12, 16

[29] Matthew Fluet, Mike Rainey, John H. Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ICFP '08, pages 119–130, 2008. 14

[30] L. Fortnow and S. Homer. A short history of computational complexity. *Bulletin of the European Association for Theoretical Computer Science*, 80, June 2003. 1

[31] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 212–223, 1998. 12, 16

[32] Lal George and Andrew Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996. 9

[33] Lal George, Florent Guillame, and John Reppy. A portable and optimizing back end for the SML/NJ compiler. In *Fifth International Conference on Compiler Construction*, number 786 in Lecture Notes in Computer Science, pages 83–97, New York, NY, April 1994. Springer-Verlag. 9

[34] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write asynchronous PRAM model. *Theor. Comput. Sci.*, 196(1-2):3–29, 1998. 5

[35] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. 11

[36] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Trans. Program. Lang. Syst.*, 21(2):240–285, March 1999. 3, 14

[37] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. 3

[38] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, 2016. 5, 6, 8

[39] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 351–363, New York, NY, USA, 1986. ACM. 11

[40] Paul Hudak and Adrienne G. Bloss. The aggregate update problem in functional programming systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 300–314, 1985. 11

[41] Shams Imam and Vivek Sarkar. Load balancing prioritized tasks via work-stealing. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, pages 222–234, 2015. 14

[42] Shams Mahmood Imam and Vivek Sarkay. Habanero-java library: a java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, pages 75–86, 2014. 12

[43] Intel. *Intel Threading Building Blocks*, 2011. 12

[44] Joseph Jaja. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Company, 1992. 2

[45] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015. 16

[46] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, 2010. 12

[47] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000. 12

[48] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ML. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 173–184. ACM, 2007. 15

[49] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992. 5

[50] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, 2009. 12

[51] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.02*. INRIA, 2013. 5

[52] Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 107–118, 2007. 4

[53] Simon Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 339–401, 2011. 4

[54] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 45–54, 2009. 16

[55] MLton web site. http://www.mlton.org. 8

[56] Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. Symposium on Logic in Computer Science (LICS)*, pages 14–23, 1989. 11

[57] Adam Morrison and Yehuda Afek. Fence-free work stealing on bounded TSO processors. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 413–426, 2014. 16

[58] Melissa Elizabeth O'Neill. *Version Stamps for Functional Arrays and Determinacy Checking: Two Applications of Ordered Lists for Advanced Programming Languages*. PhD thesis, Simon Fraser University, 2000. 11

[59] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *FSTTCS*, pages 383–414, 2008. 4

[60] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. 12

[61] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 392–406, New York, NY, USA, 2016. ACM. 4, 8, 9

[62] John Reppy, Claudio Russo, and Yingqi Xiao. Parallel Concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming ICFP '09*, pages 257–268, New York, NY, August–September 2009. ACM. 4, 8

[63] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 68–70, 2012. 16

[64] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts (7. ed.)*. Wiley, 2005. 12

[65] Standard ml family github project. 5

[66] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008. 3, 4, 8, 16

[67] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 110–120, 2015. 16

[68] Joseph Tassarotti, Ralf Jung, and Robert Harper. A higher-order logic for concurrent termination-preserving refinement, 2016. Draft. 16

[69] Alan M. Turing. On computable numbers with an application to the *Entscheidungsproblem*. *Proc. of the London Mathematical Society*, s2-42(1):230–265, 1937. (Presented Nov. 12, 1936). 1

[70] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390, 2013. 16

[71] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. GPS: navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, pages 691–707, 2014. 16

[72] The Twelf project. 15

[73] Mike Rainey Umut A. Acar, Arthur Charguéraud. Efficient synchronization-free work stealing. Submitted for publication., August 2011. 10

[74] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990. 11

[75] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag. 11

[76] Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. Work-stealing with configurable scheduling strategies. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 315–316, 2013. 14

[77] Martin Wimmer, Francesco Versaci, Jesper Larsson Träff, Daniel Cederman, and Philippas Tsigas. Data structures for task-based priority scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 379–380, 2014. 14