Responsive Parallel Computation

Stefan Muller  
Carnegie Mellon University  
smuller@cs.cmu.edu

Umut A. Acar  
Carnegie Mellon University  
umut@cs.cmu.edu

Robert Harper  
Carnegie Mellon University  
rwh@cs.cmu.edu

Abstract

As parallel (multicore) hardware proliferates, there is growing interest in developing languages and techniques for writing and reasoning about parallel programs. One important direction is to abstract away from the details of how parallelism is implemented by using implicitly parallel programming languages and reasoning about performance using abstract cost models based on the metrics of work and span, which can then be mapped to actual performance by taking advantage of classic results in the field, such as Brent’s Theorem. While very effective for compute-bound applications, the integration of these methods with effects such as input-output has not been well understood, even though many applications (e.g., games and servers) increasingly involve such interaction.

In this paper, we propose responsive parallel computing to bring together performant implicit parallelism with responsive interaction. We first present a parallel language that provides separate mechanisms for interaction and parallelism, and for prioritization of computations. The language separates foreground (high priority) and background (low priority) computations using a type system based on linear temporal logic and comes with both an operational semantics and a cost model. The cost model is based on a refinement of the work-span model that introduces the notions of foreground work and span. As our key result, we prove a “Brent-type” theorem that establishes a bound on the responsiveness as well as the completion time of a computation. We present a small implementation and several examples that give some evidence of the practical implications of our results.

1. Introduction

Shared-memory parallelism has become mainstream, with nearly every computer today including multicore chips. Many programming languages have been developed to support parallel computing on such shared-memory multiprocessors, including OpenMP, Cilk [19], Fork/Join Java [30], Habanero Java [25], TPL [31], TBB [26], X10 [12], parallel ML [18], and parallel Haskell [11, 28].

These languages enable the programmer to express parallelism at an abstract level by means of primitives such as fork-join, async-finish, and futures. This approach relieves the programmer of the complexities of specifying the mapping of tasks onto processors, instead relying on the runtime system to schedule tasks onto processors in an online fashion to minimize completion time. Determining an optimal schedule of tasks is NP-hard [44], but a classic result by Brent [9] shows that a 2-factor approximation can be computed by using a greedy scheduling principle, which requires keeping processors as busy as possible by greedily assigning ready tasks to them. Brent’s scheduling principle leads naturally to cost models for parallelism based on the notions of work, defined as the total computation to be performed, and span, defined as the longest chain of dependent computations [5, 27, 41]. Brent’s scheduling principle implies that a parallel computation with work $W$ and span $S$ can be executed in $W/P + S$ time. As a result, well-designed parallel computations should show good speedup, which measures relative improvement in completion time with respect to a single processor.

However, nearly all of these advances in programming languages, cost models and scheduling algorithms focus on minimizing completion time in compute-intensive applications such as matrix operations, Fast Fourier Transformation, Barnes-Hut, etc. Such algorithms perform a large amount of computation over given data to produce an output, with little or no interaction with the outside world.

As the potential applications of parallelism expand beyond these traditional, computational workloads, we are interested in the question of how to write and reason about parallel programs that interact with the external world. For example, an application such as a game or server that interacts with users or clients must respond quickly to ensure effective interaction while also performing a compute-intensive task (e.g. analytics on a database, AI strategy in a game) in parallel, mixing parallel computation and interaction. Such applications must maximize not only speedup, but also responsiveness (or, equivalently, minimize response time, the time by which interactive sub-computations are delayed). Very little research, theoretical or experimental, has focused on parallel interaction.

In this paper, we seek to address this problem of responsive parallel computation by developing a language and a cost model for interactive parallel programs which highlight not just overall completion time but also the responsiveness of portions of the program which are marked as high-priority. Contributions of this paper include the following:

- A calculus $\lambda^p$ with features for parallelism, interaction and priority annotations. The type system of $\lambda^p$, based on ideas from linear temporal logic (e.g. [13]), cleanly separates computation at different priorities. Its operational semantics utilizes a prompt scheduling principle which generalizes the traditional notion of greedy scheduling to prioritize foreground computations.
- A cost model which constructs cost graphs indicating both the parallel structure and the priorities of subcomputations.
- Upper bounds on both the parallel completion time and the total response time of executions of $\lambda^p$ programs, generalizing Brent’s Theorem to show that a prompt schedule is within a constant factor of optimal in terms of both completion time and responsiveness.
- A preliminary implementation of a responsive parallel library for Standard ML, and implemented examples to give some evidence of the practicality of the proposed techniques.
2. Overview

We present an overview of the results in the paper, giving relevant background as necessary. Focusing on intuition, the presentation here is informal but the rest of the paper makes the ideas precise. For the purposes of illustration, we use an ML-like (strict, purely functional) language.

Fork-Join Parallelism. Our starting point is a purely functional ML-like language with fork-join parallelism. A parallel tuple \( \text{par}(e_1, e_2) \) evaluates \( e_1 \) and \( e_2 \) in parallel, returning the resulting values as an ordinary tuple. For example, we can write a function that computes the \( n \)th Fibonacci number by using the standard recursive algorithm as follows.

```ml
function fib n =  
  if n <= 1 then n  
  else  
    let (a, b) = par (fib (n - 1), fib (n - 2))  
    in a + b
```

Since the two recursive calls are independent, they can be performed in parallel, leading to an “embarrassingly parallel” algorithm.

The average response time for each user is on the order of 1 second with 4 or more processors. With fewer processors, we registered no response. Such response times are unacceptable—e.g., a thousand threads generated by the Fibonacci function, with how work stealing operates: it treats the threads performing I/O in the same way as the thousands of threads generated by the standard recursive algorithm as follows.

```
function fib n =  
  if n <= 1 then n  
  else  
    let (a, b) = par (fib (n - 1), fib (n - 2))  
    in a + b
```

The basic idea behind the type system is to differentiate between foreground and background computations to be typed in the foreground by a distinguished type annotation \( \text{fg} \) and with prioritized computations. To enable priority, we introduce two language constructs: \( \text{fg} \) annotates a piece of code, which we refer to as a foreground block, as running in the foreground (with high priority). The annotation \( \text{bg} \) embeds a background computation, which runs with low priority, inside a foreground block.

To keep interaction responsive, it is key that foreground computations never wait on background computations by demanding their results. However, a foreground block should be able to start background computations and “pass them around” until they can be used (in the background). In Section 3, we present a language called \( L^p \) and a type system based on linear temporal logic that enforces this key invariant. The language extends a simply-typed lambda calculus with constructs for prompt scheduling (which guarantees that foreground computations never wait on background computations by demanding their results). However, a foreground block should be able to start background computations and “pass them around” until they can be used (in the background).

Composing our two examples, Fibonacci and quest, we can now write a very simple parallel interactive function that performs a large Fibonacci computation as it interacts with the user.

```
function fib_quest () =  
  par(fib 43, quest 15)
```

To see how our fib_quest example runs on an actual parallel machine, we implemented a variant of the example using a parallel extension to the ML language [40, 41] that uses a well-engineered work-stealing scheduler\(^1\). The average response time for each user input, shown by the top curve (labeled “Standard work stealing”) in Figure 1 is on the order of 1 second with 4 or more processors. With fewer processors, we registered no response. Such response times are unacceptable—effective interaction requires response times to be on the order of milliseconds. These measurements are consistent

---

\(^1\) The implementation allows I/O operations to proceed without blocking the underlying OS process/thread.

---

![Figure 1. The average response time of a simple terminal application (in microseconds, log scale).](image-url)
function fib_server () =
    let n = input () in
    if n < 0 then bg ()
    else
        output (fib n);
        fib_server ();

function main () =
    fib_server ()

Figure 2. Parallel Fibonacci without (left) and with priorities (right).

In the fib_quest example, foreground computations and background computations do not interact in interesting ways. As a more complex example, consider a "Fibonacci server". The function asks the user for an input n (a natural number) and computes the n\textsuperscript{th} Fibonacci number using fib. Since a Fibonacci computation can take a long time, the input loop could become sluggish. To solve this problem, the programmer can run fib_server in the foreground while pushing the call to fib to the background, as shown on the right in Figure 2. The expression \(\text{bg (output (fib n))}\) spawns a new background thread to asynchronously perform the Fibonacci computation and output the result. The foreground computation can spawn many background computations, each of which computes the requested Fibonacci number in parallel with other other background computations as well as the foreground interactive server loop.

Note that the foreground computation never demands the results of the background computations (though it may pass around handles to them as first class values). This requirement that foreground computations do not depend on background computations is the key principle that responsive parallel computations must enforce.

**Cost Semantics.** We establish that the evaluation strategy of \(\lambda^\text{ip}\) is efficient and responsive by extending the traditional work-span model for non-interactive parallel computing. The classic cost models represent parallel computations using a dag (directed acyclic graph) in which each vertex represents an instruction and each edge represents a dependency between the instructions. In its most basic form, each vertex/instruction represents a machine-level operation, but more abstractly, any sequence of operations can be considered as a vertex/instruction. As an example, Figure 3 illustrates the dag for our fib function with input value \(n = 3\), using a vertex to represent each recursive call to fib. 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.

Given a dag, work \(W\) is defined as the number of vertices in the dag and span \(S\) is defined as the length of the longest path in the dag. Work can be thought of as the (asymptotic) time needed to complete the computation with one processor. Span can be thought of as the (asymptotic) time needed to complete the computation with infinitely many processors. The structure of the fib function shows that it performs exponential work in linear span, i.e., \(W(n) = \Theta(d^n)\) and \(S(n) = \Theta(n)\). Brent’s Theorem establishes the key result that a computation with \(W\) work and \(S\) span can be executed on \(P\) processors in \(T_P \leq W/P + S\) time. This bound is within a factor of two of optimal, since \(W/P\) and \(S\) are each, individually, lower bounds on the total computation time.

In interactive parallel computation, in addition to parallel run time, we are interested in the responsiveness of the foreground blocks. For this, we use total response time, which we define as the total time required to execute foreground blocks, including the time blocks wait to be scheduled. We extend the classical model with the notions of foreground work \(W_f\), foreground span \(S_f\), and foreground width \(D_f\). Foreground work and span refer to the total work and span of the foreground blocks, which we mark off in the dag. Foreground width is the maximum number of foreground blocks which can be running at a time. It is called "width" since it corresponds to the maximum number of foreground blocks that can be crossed by a cut separating the already-executed and the not-yet-executed vertices in the dag at any given time in an execution. Figure 4 shows the dag for fib_quest (using \(n = 3\) and \(i = 1\)). The foreground block for quest is indicated by the rectangle enclosing part of the dag. The edge weights \(\delta_1\) and \(\delta_2\) stand for the latency incurred by the two input instructions, which are included in the span but not the work. The foreground work and span are thus \(S_f\) and \(S_f + \delta_1 + \delta_2\), respectively. The foreground width is 1.

The cost semantics for \(\lambda^\text{ip}\) (Section 4) specifies these notions precisely by generating a dag that makes it possible to read off the (1) work and span, (2) foreground work and span, and foreground width. We then show (Section 5) that the operational semantics of \(\lambda^\text{ip}\) evaluates a program with work \(W\), span \(S\), foreground work \(W_f\), foreground span \(S_f\), and foreground width \(D\) in time \(\frac{W}{P} + S + D\) with total response time \(\frac{W}{P} + S_f\). Note that the total response time depends only on the properties of the foreground blocks, effectively excluding whatever other work might be performed in the background, and that the completion time for the whole program is the same as given by Brent’s classic result. We are thus able to show that responsiveness can be guaranteed without penalizing non-interactive computations. For example, for both fib_quest and fib_server, the response time is bounded by the relatively small \(\frac{W}{P} + S_f\) and is independent of the (much larger) work and span of the Fibonacci computations.

An important property of Brent’s principle is that it guarantees that the P-processor run-time is within a factor 2 of optimal. For our prompt scheduling principle, we establish a similar but slightly weaker optimality bound for total response time which assumes that the scheduler has no information about the shape of the dag and thus cannot make decisions based on its future shape. Since computation dags unfold dynamically, this is a reasonable assumption.
3. Language

In this section, we introduce a core calculus called $\lambda^P$, which extends a simply-typed lambda calculus with the features introduced in Section 2 for I/O, parallelism and priority. The type system of $\lambda^P$ separates subcomputations by priority (foreground or background). The operational semantics for the language makes explicit the available pool of threads and simulates a run of a $P$-processor parallel scheduler on the program. The threads scheduled at each step of the semantics are chosen nondeterministically, allowing for a variety of scheduling policies and mechanisms (e.g. work stealing).

The semantics require only that the scheduler be prompt, keeping all processors busy when possible and prioritizing foreground blocks.

3.1 Syntax

The syntax of $\lambda^P$ syntax is presented in Figure 5. Most features are fairly standard for a simply-typed lambda calculus. The types $\tau$ include two base types: unit and natural numbers, as well as functions, binary tuples, binary sums and the circle type $\bigcirc \tau$, which represents handles to background threads. The expressions $e$ include the standard introduction and elimination forms for base types, functions, pairs and sums: natural numbers $\mathbb{N}$, $\lambda$-abstractions, application, pairs, projection, injection, and case analysis. Recursion is possible through the fixed point operator $\text{fix}\ x : \tau \to e$.

Parallel pairs $e_1 \parallel e_2$ evaluate $e_1$ and $e_2$ simultaneously and return a pair of their values when both computations complete. The form $\text{join}(a, b)$ where $a$ and $b$ are thread identifiers, will, in the operational semantics, indicate the point at which two parallel computations join, but this form is not included in source programs.

The command $\text{out}(e)$ evaluates $e$ to a (natural number) value and then outputs it, returning $. The command $\text{inp}(d) \langle x, e \rangle$ takes a natural number as input from the user, which blocks for some amount of time, and binds this value to $x$ in $e$. The input happens in two stages: $\text{inp}(d) \langle x, e \rangle$ performs the blocking and steps to $\text{in}(x,e)$ (which does not appear in source programs), which actually substitutes the input into $e$. Blocking is controlled by the symbol $d$, which identifies this input. The operational semantics and the cost semantics of Section 4 will be parametrized over an assignment $\Delta$ which maps these symbols to sets of non-negative integer delays, thus specifying how long each input command may block. Generally, the set will be an interval specifying the minimum and maximum delays. The use of identifiers and mappings in this way allows different inputs to incur different delays. For example, input might be used to represent a “sleep” operation that waits exactly $n$ time steps and then returns $. If this input was marked with identifier $d$, then we would run the program with a $\Delta$ such that $\Delta(d) = n$. On the other hand, if an input waits for a response from the user, we might use an interval containing a reasonable estimate of how long it would take the user to respond. There is no difficulty in extending the I/O constructs to handle other base types, but we restrict ourselves to natural numbers for simplicity.

The circle type $\bigcirc \tau$ is the type of a handle to a background thread running an expression of type $\tau$. The introduction form for $\bigcirc \tau$ is $bg(e)$, which returns a first-class thread handle $\text{tid}(b)$. Thread handles are eliminated by $fg(e)$, which runs $e$ in the foreground until it evaluates to a background thread handle $fg(e)$.

3.2 Static Semantics

The type system of $\lambda^P$ makes explicit the distinction between foreground and background computations, which prevents responsiveness problems that could result if foreground blocks were accidentally allowed to wait for background code, or if time-sensitive
foreground code were accidentally run in the background. We enforce this separation using a type system based on ideas drawn from linear temporal logic and other type systems based on LTL, especially those for staged computation. The relationship with staged languages is discussed further in Section 7.

The main typing judgment for \( \lambda^p \) is \( \Gamma \vdash e : \tau @ w \). This judgment indicates that \( e \) has type \( \tau \) at world \( w \) (where \( w \in \mathbb{P} \) or \( \emptyset \)). The judgment makes use of two contexts. Variable contexts \( \Gamma \) have entries of the form \( x : \tau @ w \), indicating that variable \( x \) is in the context with type \( \tau \) at world \( w \). Thread signatures \( \Sigma \) have entries of the form \( a \sim \tau @ w \), indicating that thread \( a \) is running an expression of type \( \tau \) at world \( w \). Most of the rules allow expressions to type at any world, but require all subexpressions to be at the same world as the whole expression, enforcing the restriction that code can only be moved between worlds by spawning an asynchronous background thread with \( \text{bg}(e) \) or starting a foreground block with \( \text{fg}(e) \). If \( e \) is background code of type \( \tau \), the expression \( \text{bg}(e) \) starts a background thread of type \( \tau \) and immediately returns a handle of type \( \circ \tau \) in the foreground. If \( e \) types in the foreground with type \( \circ \tau \), i.e. it will evaluate to (a thread running) background code of type \( \tau \), the expression \( \text{fg}(e) \) has type \( \tau @ \emptyset \). Typing \( \text{fg}(e) \) at \( \emptyset \) prevents the nesting of foreground blocks, as desired.

There are two rules for typing variables. If \( x : \tau @ w \) is in the context, the variable \( x \) has type \( \tau \) at world \( w \). We also allow variables of type \( \text{nat} \) at type at either world, allowing foreground code to make use of variables (of type \( \text{nat} \)) bound in the background and vice versa. The restriction to type \( \text{nat} \) ensures that code can’t "escape" to the wrong world encapsulated in a function or thread. This is related to the mobility restriction of Murphy et al. [33], and could easily be expanded to allow any "mobile" type, including \( \text{unit} \), sums and products (but not functions of \( \circ \)). The rules for \( \text{join}(a,b) \) and \( \text{tid}(b) \) look up the thread identifiers in the signature and produce the appropriate types.

The final judgment, \( \Gamma \vdash e : \Sigma \), indicates that the thread pool \( \mu \) has the signature \( \Sigma \). The rules require that \( a \sim \tau @ w \in \Sigma \) if and only if \( a \Rightarrow (\delta, e) \in \mu \) and \( e \) has type \( \tau \) at world \( w \). For the purposes of typing, thread pools are ordered and threads may only refer to threads that come later in \( \mu \). This ensures that the references between threads are acyclic. However, we will occasionally treat the thread pool as the unordered set of its threads when this property is not important. Expressions are also allowed to refer to threads in \( \Sigma' \), which must be disjoint from \( \Sigma \), allowing us to type just a part of a thread pool, whose expressions may refer to threads outside this part. Whenever \( \mu \) is the entire thread pool, \( \Sigma' \) will be empty.

Lemma 1 states a property of thread pool typing which will be useful later: the concatenation of two thread pools is well-typed with the concatenation of the two signatures.

**Lemma 1.** If \( \Gamma \vdash e : \Sigma_1 \) and \( \Gamma \vdash e : \Sigma_2 \) then \( \Gamma \vdash e : \Sigma_1 \Sigma_2 \).

**Proof.** By induction on the derivation of \( \Gamma \vdash e : \Sigma_1 \). If \( \mu_1 = \emptyset \), then the result is trivial. Otherwise, \( (\delta, e) \in \mu_1 \) and \( \Sigma_1 = \Sigma', \sigma \sim \tau @ w \in \Gamma \vdash e : \sigma @ w \) and \( \Gamma \vdash e : \Sigma_1 \Sigma_2 \). By induction, \( \Gamma \vdash e : \mu_1 \Sigma_2 \). By weakening, \( \Gamma \vdash e : \Sigma' \Sigma_2 \). The result follows from the thread pool typing rules.

\( \Box \)

### 3.3 Dynamic Semantics

The operational semantics of \( \lambda^p \) consists of two components: local and global. This separation, and much of our notation, is drawn from Harper [23]. The local semantics concerns individual threads, and indicates how expressions transition. Selected rules are presented in Figure 7 as small-step transition rules. The rules in this figure define two judgments. The judgment \( e \text{ val} \) indicates that \( e \) is an irreducible value. Values are the unit value, numerals, functions, pairs and injections of values, and thread handles \( \text{tid}(b) \).

The local transition judgment is

\[
e \mid \mu \Rightarrow_a (\delta', e') \mid \mu @ \mu'
\]

which states that thread \( a \) running \( e \) transitions to \( e' \), possibly spawning new threads, which are collected in \( \mu' \). The original thread pool \( \mu \) is unchanged; threads are never altered or removed by local transitions. The thread identifier \( a \) is not important for the local transition, but will be used in some of the global definitions and results. The new expression \( e' \) will be able to run after a delay of \( \delta' \) steps (if \( \delta' = 0 \), it can run immediately). The judgment is also parametrized by \( \Delta : \text{InputIDs} \Rightarrow \mathcal{V} \), a mapping which assigns a set of possible delays to each input identifier \( d \).

Most of the transition rules are straightforward and are omitted. The complete rules for function application are given as an example: in \( e \circ e_2 \), the subexpression \( e_1 \) is stepped until it is a lambda abstraction, then \( e_2 \) is stepped until it is a value, which is then substituted for the variable in the body of the abstraction using standard capture-avoiding substitution. A parallel tuple \( e_1 | e_2 \) spawns two new threads \( b \) and \( c \) to execute \( e_1 \) and \( e_2 \), respectively. The local thread \( a \) steps to \( \text{join}(b,c) \), indicating that this thread is now waiting for \( b \) and \( c \) to complete. When both threads have stepped to irreducible values, \( \text{join}(b,c) \) steps to a pair of the two values. In the same vein, \( \text{bg}(e) \) spawns a new thread \( b \) to evaluate \( e \) and returns the thread handle \( \text{tid}(b) \).

The case for foreground. By inversion, \( e \text{ val} \) or \( e \text{ is a \lambda \ abstraction} \), then either \( e \text{ val} \) or \( \mu \Rightarrow_a (\delta, e') \mid \mu @ \mu' \) or there exists \( b \Rightarrow (\delta, e_0) \in \mu \) such that \( \delta > 0 \) or \( e_0 \Rightarrow (\delta, e') \mid \mu @ \mu' \).

**Lemma 2** (Local Progress). If \( \cdot \vdash e : \tau @ w \) and \( \cdot \vdash \mu : \Sigma_1 \rightarrow \tau @ w \), then either \( e \text{ is \lambda \ abstraction} \) or \( e \Rightarrow_a (\delta, e') \mid \mu @ \mu' \). Consider the case for foreground. By inversion, \( b \sim \tau @ w \in \Sigma \). The case for background is similar. The interesting cases are \( e \Rightarrow (\delta, e_0) \) and \( e = \text{join}(b,c) \).

The statement of preservation is more standard but requires finding a signature \( \Sigma' \) which accounts for the new threads that are created when \( e \) takes a step.

**Lemma 3** (Local Preservation). If \( \cdot \vdash e : \tau @ w \) and \( e \Rightarrow_a (\delta, e') \mid \mu @ \mu' \), then there exists \( \Sigma' \) such that \( \cdot \vdash e : \tau @ w \) and \( \cdot \vdash e : \Sigma' \).

2016/7/7
The global rules in Figure 8, together with the auxiliary definitions in Figure 9, define the transitions of entire thread pools, i.e., the entire state of the computation. The judgment \( \mu \text{ final} \) states that \( \mu \) has completed evaluating and its rules simply require that all threads in \( \mu \) be irreducible. If a thread pool is not final, we wish to step threads according to the prompt scheduling principle. We indicate that thread \( i \) is ready and foreground. Threads \( i \) are currently running foreground blocks. For a thread pool \( \mu \), a thread \( a \) and its associated expression \( e \), \( \text{RFB}(e,a) \) is a set \( A_1,\ldots,A_n \) where each \( A_i \) represents a separate, currently ready, foreground block in \( e \) and is a set of the threads currently working on the foreground block. A foreground block is ready if any of the threads working on it is ready. Expressions not containing foreground blocks result in the empty set. Because \( \text{RFB}(e,a) \) should only contain ready foreground blocks, the rule for \( e_1, e_2 \) must split on whether \( e_1 \) or \( e_2 \) is currently evaluating. The subexpression which is currently evaluating is recursively explored for foreground blocks. Finally, \( \text{fg}(e) \) contains one foreground block involving \( a \) and any other threads which are involved in evaluating \( e \) (since \( e \) may contain parallel pairs). This is determined by the function \( J_\delta(e) \), which gives the set of threads \( a \) on which \( e \) is (transitively) waiting with join expressions. For example, if \( e = \text{join}(b,c) \) and thread \( b \) is running \( \text{join}(d,e) \) and thread \( e \) is fully evaluated, then \( J_\delta(e) = (b,c,d,e) \). The definition is inductive on the expression and is straightforward. At a \( \text{join} \), the two parent threads are added to the set and their expressions are explored recursively. Finally, \( \text{RFB}(\mu) \) collects all of the ready foreground blocks for each thread in \( \mu \). The judgment \( \text{isfg}(a) \) indicates that \( a \) is a foreground thread in \( \mu \) and simply checks whether \( a \) is involved in any block of \( \text{RFB}(\mu) \).

The global step relation is \( \tau ; \mu \Rightarrow \tau' ; \mu' \) and has only one rule, which allows some number of threads whose delay is 0 to step using the local dynamics. The relation also includes a counter for the total response time \( \tau \), which at each step is incremented by the number of ready foreground blocks. The rule first separates the threads \( a_1,\ldots,a_m \) into categories (here we treat the thread pool as an unordered set of threads). Threads \( 1 \) through \( j \) are ready and foreground. Threads \( j+1 \) through \( k \) are ready and background. Threads \( k+1 \) through \( n \) are neither ready nor delayed, and threads \( n+1 \) through \( m \) are delayed. The rule will run the first \( N \) threads, where \( N \) is the smaller of \( k \) and \( P \). In this way, the scheduler runs as many threads as possible, prioritizing foreground threads, as required by the prompt scheduling principle. Since the threads may be reordered within the categories arbitrarily, the rule does not, for example, specify which foreground thread to schedule if more than \( P \) are available. The new thread pool consists of the updated threads \( 1 \) through \( N \), the unaltered threads \( N+1 \) through \( n \) and the delayed threads \( n+1 \) through \( m \) with their delays decremented. Finally, the total response time \( \tau \) is incremented by the number of ready foreground blocks. (Commuting the summations, counting the number of blocks at each step is equivalent to counting the number of steps taken to execute each block, which is the response time.)

We can now prove progress and preservation for the global semantics. Most of the work is done by Lemmas 2 and 3.

**Lemma 4 (Progress).** If \( \cdot \vdash \cdot : \Sigma \) and then either \( \mu \text{ final} \) or there exist \( \tau' \) and \( \mu' \) such that \( \tau ; \mu \Rightarrow \tau' ; \mu' \).

**Proof.** Let \( \mu = a_1 \Rightarrow (\delta_1, e_1) \circ \ldots \circ a_m \Rightarrow (\delta_m, e_m) \). If any \( \delta_i > 0 \), then the configuration can take a step to reduce \( \delta_i \), so consider the case where \( \delta_1 = \cdots = \delta_m = 0 \). By inversion on configuration typing derivation, we have \( \Sigma = a_1 \sim \tau_1 @ w, \ldots, a_m \sim \tau_m @ w \) and for all \( i \), there exists \( \Sigma_i \) such that \( \cdot \vdash \cdot : \tau_i @ w \). By Lemma 2, either there exists some \( i \) such that \( e_i | \mu \Rightarrow (\delta_i, e'_i) | \mu' \) for all \( i, \mu \text{ final} \). In the former case, the configuration can take a step, and in the latter case, \( \mu \text{ final} \).

**Lemma 5 (Preservation).** If \( \cdot \vdash \cdot : \Sigma 
and \tau ; \mu \Rightarrow \tau' ; \mu' \), then there exists \( \Sigma' \) such that \( \cdot \vdash \cdot : \Sigma' \).

**Proof.** Apply Lemma 3 to each local step, then use weakening and Lemma 1 to combine the results. See the appendix in the supplementary materials for details.

We could now show a fairly standard type safety theorem, showing that a well-typed thread pool will not become “stuck”. However, there is one additional property, in addition to well-typedness, which we wish to ensure is preserved during execution. We call this property “well-jointedness”. It is defined by the judgment \( \epsilon \in \mathcal{J} (= \text{is well-jointed}) \) in Figure 10. Intuitively, well-jointedness

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Rule} & \text{Pre-condition} & \text{Post-condition} \\
\hline
\text{fg(e)} & \mu \Rightarrow (\delta, \text{fg}(e)) | \mu' & \mu \Rightarrow (\delta, \text{fg}(e)) | \mu' \\
\text{join(b,c)} & (0, e_1) | (0, e_2) & (0, (e_1, e_2)) | \mu \\
\text{inl(e)} & \mu \Rightarrow (0, e) | \mu & \mu \Rightarrow (0, e) | \mu \\
\text{inr(e)} & \mu \Rightarrow (\delta, e) | \mu & \mu \Rightarrow (\delta, e) | \mu \\
\text{tid(a)} & \mu \Rightarrow (\delta, \text{tid}(a)) | \mu & \mu \Rightarrow (\delta, \text{tid}(a)) | \mu \\
\text{out(e)} & \mu \Rightarrow (\delta, e) | \mu & \text{in}(\delta-1, \text{in}(x,e)) | \mu \\
\text{fix(x : \tau) \Rightarrow e(x : \tau)} & \mu \Rightarrow (\delta, e) | \mu & \mu \Rightarrow (\delta, e) | \mu \\
\text{Join(b,c)} & (0, e_1) | (0, e_2) & (0, (e_1, e_2)) | \mu \\
\hline
\end{array}
\]
If thread steps, then all resulting expressions are well-joined.

We first show that well-joinedness is preserved by local transitions: if all expressions of a thread pool are well-joined and one expression which is currently being evaluated is the property that

$$
\mu = a_1 \rightsquigarrow (0, e_1) \otimes \cdots \otimes a_n \rightsquigarrow (0, e_n) \rightsquigarrow a_{n+1} \rightsquigarrow (\delta_n, e_n) \otimes \cdots \otimes a_m \rightsquigarrow (\delta_m, e_m)
$$

$$
\forall 1 \leq i \leq j \text{ isready}_\mu(a_i) \land \text{isfg}_\mu(a_i) \quad \forall j < i \text{ isready}_\mu(a_i) \land \lnot(\text{isfg}_\mu(a_i)) \\
N = \min(k, P) \quad \forall 1 \leq i \leq N, e_i \models \delta_i \otimes \mu \Rightarrow \delta_i \otimes \mu' \\
\mu' = a_1 \rightsquigarrow (\delta_1, e_1) \otimes \cdots \otimes a_N \rightsquigarrow (\delta_N, e_N) \otimes a_{n+1} \rightsquigarrow (0, e_{n+1}) \otimes \cdots \otimes a_m \rightsquigarrow (\delta_m, e_m) \otimes \mu' \otimes \cdots \otimes \mu'_n
$$

$$
\forall 0 \leq i \leq n, \text{we have } e_i \text{ wj, then } e_i' \text{ wj and for all } n < i \leq m, \text{ we have } e_i \text{ wj.}
$$

$$
\text{Proof. By induction on the derivation of the transition judgment. See the appendix in the supplementary materials for details.}
\Box
$$

Finally, we prove a theorem which encompasses type safety and well-joinedness. If an initial thread pool consisting of a single source expression (which is well-typed under the empty context and signature) evaluates to μ after some number of steps, then μ′ is well-typed, not stuck and all of its expressions are well-joined.

**Theorem 1 (Type Safety and Well-Joinedness).**

1. there exists Σ′ such that \( \cdot \vdash e : \tau \land \delta \Rightarrow r; \mu' \), then
2. either \( \mu' \) final or there exist \( r'' \) and \( \mu'' \) such that \( r; \mu' \Rightarrow g \) \( r''; \mu'' \)
3. For all \( b \rightsquigarrow (\delta, e_b) \in \mu' \), we have \( e_b \text{ wj} \).

$$
\text{Proof. By induction on the derivation of the transition judgment. Recall that, in such a dag, vertices represent instructions of a program and edges represent control dependencies between the instructions. Vertices with no ancestor relationship between them may be executed in parallel. We also use a recent extension to the dag model [32] which labels each edge with a positive integer weight \( \delta \) to represent the delays incurred by input operations. An edge from } u_1 \text{ to } u_2 \text{ with weight } \delta \text{ is written } (u_1, u_2, \delta). \text{ If } \delta = 1, \text{ } u_1 \text{ incurred no latency and } u_2 \text{ may execute on the next step. If } \delta > 1, \text{ } u_1 \text{ incurred a latency of } \delta \text{ and } u_2 \text{ may not execute fewer than } \delta \text{ steps after } u_1. \text{ In the weighted dag model, the notion of work is unchanged; it is still the total number of vertices. The span, on the other hand, is now defined as the longest weighted path. This captures the notion that the time spent blocking}
$$

### 4. Cost Semantics

In this section, we define a cost semantics which constructs a cost dag of the form described in Section 2 for a \( \mathcal{A}^\delta \) program. The parallel structure of the program, as well as the cost metrics such as work and span, can be read off from the resulting dag. Recall that, in such a dag, vertices represent instructions of a program and edges represent control dependencies between the instructions. Vertices with no ancestor relationship between them may be executed in parallel. We also use a recent extension to the dag model [32] which labels each edge with a positive integer weight \( \delta \) to represent the delays incurred by input operations. An edge from \( u_1 \) to \( u_2 \) with weight \( \delta \) is written \( (u_1, u_2, \delta). \) If \( \delta = 1, \) \( u_1 \) incurred no latency and \( u_2 \) may execute on the next step. If \( \delta > 1, \) \( u_1 \) incurred a latency of \( \delta \) and \( u_2 \) may not execute fewer than \( \delta \) steps after \( u_1. \) In the weighted dag model, the notion of work is unchanged; it is still the total number of vertices. The span, on the other hand, is now defined as the longest weighted path. This captures the notion that the time spent blocking.
The former represents a foreground block whose source vertex has not yet been executed. It may still be ready if the source vertex is delayed and can run in \( u \) auxiliary vertex \( \alpha \) of an edge can, in addition to a vertex, be a thread identifier or thread dag as a tuple \( (s, t) \). For each dag, we will generate a traditional dag, which we now call a thread graph or thread dag with a single source and single sink. These are then composed to form a configuration graph or configuration dag by adding edges that correspond to the inter-thread dependencies created by \( \text{join} \) and \( \text{fg} \).

We will use metavariables \( g \) (and similar) for thread dags. The notation \( u \| v \) indicates that \( u \) is an ancestor of \( v \) in \( g \). If \( u \) has no ancestor in \( g \), we write \( u \uparrow g \). We write \( g_1 \equiv g_2 \) to mean that \( g_1 \) and \( g_2 \) are isomorphic. We write a non-empty thread dag as a tuple \( (s, t, V, E, F) \) (an empty dag is written \( \emptyset \)). The first two components, \( s \) and \( t \), are the source and sink vertices respectively, and \( V \) is the set of vertices. We have \( s, t \in V \) and \( s \preceq t \). The fourth component is a set of weighted, directed edges \( E \subseteq \mathcal{P}(V \cup \text{Threads} \cup \text{AuxVertices}) \times V \times \mathbb{N} \). Note that the source of an edge can, in addition to a vertex, be a thread identifier or an auxiliary vertex. Edges starting at thread identifiers indicate dependencies between thread dags in a configuration dag. Auxiliary vertices \( \alpha \) are used as placeholders for delays. A vertex \( u \) which is delayed and can run in \( \delta \) timesteps will have an in-edge (\( \alpha, u, \delta \)). The auxiliary vertex \( \alpha \) does not count toward the work but is counted as an ancestor of \( u \), e.g. for the purposes of determining whether \( u \) is ready. The final component of the tuple is a set \( F \) of foreground blocks, where a foreground block \( f \) takes one of two forms:

1. a pair of the source and sink vertices of the block, written \( s \mapsto t \), where \( s, t \in V \) and \( s \preceq t \). The foreground block is the induced subdag of \( g \) consisting of all \( u \in V \) such that \( s \preceq u \) and \( u \preceq t \).
2. a sink vertex, written \( \mapsto \), where \( t \in V \). The foreground block is the induced subdag of \( g \) consisting of all \( u \in V \) such that \( u \preceq t \).

The former represents a foreground block whose source vertex has not yet been executed. It may still be ready if the source vertex is ready. The latter represents a ready foreground block whose source vertex has been executed. Multiple threads may be running code that is part of this block. We write \( u \in f \) to indicate that \( u \) is part of the subdag induced by \( f \), and we say that \( u \) is a foreground vertex.

The work \( W(g) \) of a dag is the total number of vertices in the dag. The span \( S(g) \) is the longest weighted path in the dag. For a foreground block \( f \) which is part of \( g \), we will write \( W_f(f) \) and \( S_f(f) \) for the work and span respectively, of the subdag of \( g \) induced by \( f \).

For a graph \( g = (s, t, V, E, F) \), the foreground work and span \( W'(g) \) and \( S'(g) \) are defined as:

\[
W'(g) = \sum_{f \in F} W_f(f) \quad S'(g) = \sum_{f \in F} S_f(f)
\]

Two foreground blocks \( f_1 \) and \( f_2 \) are serial if there exists a directed path in the graph from a vertex of \( f_1 \) to a vertex of \( f_2 \) or vice versa. A set of foreground blocks \( F' \subseteq F \) may happen in parallel if for all \( f_1, f_2 \in F' \), \( f_1 \) and \( f_2 \) are not serial. The foreground width \( D \) of a graph is defined as:

\[
\max \{ |F'| \mid F' \subseteq F \land F' \text{ may happen in parallel} \}
\]

That is, \( D \) is the maximum number of foreground blocks that may be ready at the same time.

A configuration graph \( G \) mirrors the structure of the thread pool \( \mu \); it is a mapping from thread names to thread graphs:

\[
G = \mathcal{a} \mapsto g_1, \mathcal{a} \mapsto g_2, \ldots, \mathcal{a} \mapsto g_n
\]

The vertices, edges and foreground blocks of a configuration graph are the union of the vertices, edges and foreground blocks of the component thread graphs. If \( G = \mathcal{a} \mapsto g \), \( b \mapsto g_2 \), an edge \( (a, u, \delta) \) may be viewed as an edge from the sink of \( g_1 \) to \( u \). If \( g_1 = \emptyset \), this edge is ignored. The metrics such as work, span and foreground width extend in the natural way to configuration graphs.

The cost semantics in Figure 11 generates thread graphs for expressions. The judgment \( e, \mu \vdash G \triangleright v \) indicates that the expression \( e \) evaluates to \( v \) and has cost graph \( g \) in the presence of \( \mu \). The vertices \( v \) consist of irreducible expressions, plus a new vertex \( \text{thread} \) which abstractly represents a thread as the value to which it will evaluate and a handle to the sink of its expression’s cost graph:

\[
\text{Values} \quad v \ ::= \{ \} \mid n \mid \lambda x. \tau \cdot v \mid \text{inl}(v) \mid \text{inr}(v) \mid \text{tid}[b] \mid \text{thread}[u](v)
\]

The expression being evaluated may refer to threads in \( \mu \). These threads are included so that the value can be generated, but their cost is not included in \( g \). Many of the rules for the sequential components of the language and parallel tuples are based on the cost semantics of Spoonhower et al. [41], with nontrivial modifications to allow the representation of in-progress computations. The rules for generating and joining with background threads \( \text{bg}(e) \) and \( \text{fg}(e) \), respectively, are based on Spoonhower’s treatment of futures [40], which share the property that an asynchronous expression is spawned in one part of a computation and demanded in another. The generation of cost graphs is defined inductively on expressions. Subexpressions are evaluated, and their cost graphs are combined using the operations defined in Figure 12. The figure also defines notation for simple graphs consisting of a single vertex \( [u] \) or a single edge \([u_1, u_2, \delta]\).

In most operations, the subexpressions are evaluated sequentially, represented in the cost graph by combining the cost graphs of the subexpressions using serial composition \( g_1 \otimes g_2 \) which joins the sink of \( g_1 \) to the source of \( g_2 \) by an edge of weight \( \delta \) (a more general form, \( \otimes \), uses an edge of weight \( \delta \), as shown in Figure 13). The empty graph \( \emptyset \) acts as a unit for the \( \otimes \) operator. In the rule for \( e_1 \parallel e_2 \), however, the cost graphs for \( e_1 \) and \( e_2 \) are combined using parallel composition \( g_1 \oplus g_2 \), which joins the graphs in parallel with new vertices \( s \) and \( t \) as the source and sink (Figure 14). If one of the vertices is empty, the other is simply composed with \( s \) and \( t \). The rule for \( \text{bg}(e) \) uses the left parallel composition operator [40]. If \( g \) is the cost graph for \( e \), the graph \( g \uplus \text{"g"} \) “hangs g off of” vertex \( u \) (Figure 15). For the purposes of sequentially composing this graph with other graphs, \( u \) is both the source and the sink, reflecting the fact that the new thread is executed concurrently with the continuation of the current thread. The rule for \( \text{fg}(e) \) evaluates \( e \) to a background thread and also gets a handle to the sink of the cost graph for the thread’s expression. The sink is \( u \) if the thread is of the form \( \text{thread}[u](v) \) (a thread that hasn’t yet been spawned and is represented abstractly in the cost graph) or \( b \) if the thread is of the form \( \text{tid}[b] \) (an active thread with identifier \( b \)). The rule adds an edge between the sink and the vertex representing the \( \text{fg} \) instruction. In the rule for \( \text{fg}(e) \), the cost graph for \( e \) is marked as foreground with the operation \( g \).

This operation produces a foreground block \( s \mapsto t \) if \( s \) and \( t \) are the source and sink of \( g \) and \( s \) has no ancestors (i.e. is not a join point), or a foreground block \( \mapsto \) if \( s \) is the sink of \( g \) and \( g \) depends on other threads. Finally, the input rule adds an edge of weight \( \delta \), where \( \delta \) is chosen nondeterministically from \( \Delta(d) \).

The judgment \( \mu; \mu_\mu \vdash G \) generates a portion of a configuration graph from the threads in a partial thread pool \( \mu \), by generating
Expression cost semantics \( e; \mu \downarrow^v_c v; g \)

\[
\begin{array}{c|c|c|c|c}
\text{val} & e; \mu & \mu^v_c & v; g & e; \mu \downarrow^v_c (v_1, v_2); g & \text{u fresh} \\
\hline
e_1; \mu & \mu^v_c & \mu^v_c & v; g & e_1; e_2; \mu^v_c & \mu^v_c & v; (g_1 \oplus g_2 \oplus [u]) \oplus g_3 & \text{u fresh} \\
\hline
\text{snd}(e); \mu & \mu^v_c & \mu^v_c & v; g & e_1; e_2; \mu^v_c & \mu^v_c & v; (g_1 \oplus g_2 \oplus [u]) & \text{u fresh} \\
\hline
\mu = \mu' \oplus b \mapsto (\delta, e_0) \circ b \mapsto (\delta, e_1) \quad e_0; \mu \downarrow^v_c v; g & e_1; e_2; \mu \downarrow^v_c v; g & \text{u fresh} \\
\hline
\text{inl}(v); g & [v/x]e_1; \mu \downarrow^v_c v'; g & \text{u fresh} \\
\hline
\text{case}(e)(x; e_1; x; e_2); \mu \downarrow^v_c v'; g & [v/y]e_2; \mu \downarrow^v_c v'; g & \text{u fresh} \\
\hline
\text{inr}(v); g & \text{u fresh} \\
\hline
\text{bg}(e); \mu \downarrow^v_c v; g & \text{u fresh} \\
\hline
\text{thread}[1](v); g & \text{u fresh} \\
\hline
\text{out}(e); \mu \downarrow^v_c v; g & \text{u fresh} \\
\hline
\end{array}
\]

Thread pool cost semantics \( \mu; \mu_2 \downarrow^v_c v; g \)

\[
\begin{array}{c|c|c|c|c}
0; \mu_2 & \downarrow^v_c () & a \mapsto (\delta, e) \oplus \mu; \mu_2 & a \mapsto g & \text{u fresh} \\
\end{array}
\]

\[\mu; \mu_2 \downarrow^v_c (G) \]

\[e; \mu \downarrow^v_c v; g \neq \emptyset \]

\[\emptyset; \mu \downarrow^v_c G \]

---

**Figure 11. Cost Semantics**

**Figure 12. Graph building and composition operations**

**Figure 13.** \( g_1 \oplus g_2 \)

**Figure 14.** \( g_1 \oplus g_2 \)

**Figure 15.** \( g \uparrow^a \)

---

The remaining work and span of the program. The work of a thread pool \( \mu \) under \( \Delta \) is written \( W(\mu, \Delta) \) and is defined as the maximum work over all dags that can be generated from \( \mu \):

\[
W(\mu, \Delta) = \max\{W(G) \mid \mu; \mu^v_c G\}
\]

We take the maximum since the cost semantics is nondeterministic. The definitions of \( S(\mu, \Delta), W(\mu, \Delta) \) and \( S' (\mu, \Delta) \) are similar.

In the remainder of this section, we show important properties of the cost semantics and its correspondence with the operational semantics. Lemma 7 relates the invariants of the type system to cost graphs: cost graphs generated by \( P \) expressions have no nested foreground blocks and edges from other threads occur only at joins.

**Lemma 7.** If \( x \mapsto (\delta, e) \oplus \mu : \Sigma, a \sim \tau @ P \) and \( a \mapsto (\delta, e) \oplus \mu; \mu^v_c G \) then

1. \( F = \emptyset \)

\[G = a \mapsto g \oplus a_1 \mapsto (s_1, t_1, V_1, E_1) \oplus \ldots \oplus a_n \mapsto (s_n, t_n, V_n, E_n, F_n)\]

and \( g = (s, t, V, E, F) \), then
The important rules are the ones for thread handles:

- Parts 1-3 are by induction on the derivation of \( \cdot \vdash e : \tau \otimes \pi \).
- In part 3, inversion on \( e \vdash t \) is used to show that any graphs that may be serially composed before a join are empty. Part 4 is by

\[ \text{lexicographic induction on the derivations of } \cdot \vdash e : \tau \otimes \pi \text{ and } \vdash a \iff (0, e) \otimes \mu : \Sigma, a \vdash \tau \otimes \pi \text{.} \]

See the appendix in the supplementary materials for details.

Many of the functions and properties we have defined over thread pools have corresponding definitions over cost graphs. For example, we have already defined foreground threads of a thread pool and foreground vertices of a cost graph. We can also define the function \( \text{RFB}(v) \) over graphs. The ready foreground blocks of a graph are the foreground blocks that have ready vertices. Formally,

\[ \text{RFB}(s, t, V, E, F) = \{ t \vdash e \vdash F \} \cup \{ s \vdash e \vdash F | s \vdash e \vdash G \} \]

\[ \text{RFB}(a_0 \vdash g_0 \otimes \cdots \otimes a_n \vdash g_n) = \text{RFB}(g_0) \otimes \cdots \otimes \text{RFB}(g_n) \]

For these definitions to make sense, it should be the case that the thread pool and cost graph versions of the definitions agree. First, a thread is ready in \( \mu \) if only if its source has no ancestors in \( G \). Second, an element of \( \text{RFB}(\mu) \) consists of threads whose sources are elements of a ready foreground block in \( \text{RFB}(G) \).

**Lemma 8.** Fix \( \Delta \). Suppose \( \mu = a_1 \vdash (\delta_1, e_1) \otimes \cdots \otimes a_n \vdash (\delta_n, e_n) \) and \( \vdash a : \Sigma \) and \( e \vdash w \) for all \( e \) and \( \mu, \mu' \vdash (G) \), where

\[ G = a_1 \vdash (s_1, t_1, V_1, E_1, F_1) \otimes \cdots \otimes a_n \vdash (s_n, t_n, V_n, E_n, F_n) \]

Then

1. \( \text{isready}_\mu(a) \) if and only if \( s_i \not\vDash \).
2. \( \text{RFB}(\mu) = \{ [a_i | s_i \not\vDash t] | t \vdash e \in \text{RFB}(G) \} \cup \{ [a_i | s_i \not\vDash t] | s \vdash t \in \text{RFB}(G) \} \)

**Proof.** See the appendix in the supplementary materials.

Next, we show that the operational semantics and cost semantics agree on the values produced by an expression. One complication in showing this result is accounting for the value \( \text{thread}[a](v) \) which is produced by the cost semantics but not the operational semantics\(^3\). We therefore show that the cost semantics and the operational semantics are equivalent up to a relation \( \preceq_\mu \) which relates the two forms of thread handle. We define \( \preceq_\mu \) inductively. The important rules are the ones for thread handles:

\[ \mu = b \vdash (\delta, e, \mu') \quad e, \mu' \vdash v : g \quad \text{thread}[a](v) \preceq_\mu \text{tid}[b] \]

\[ \text{tid}[b] \preceq_\mu \text{tid}[b] \]

\[ v \preceq_\mu v' \]

\[ \text{thread}[a](v') \preceq_\mu \text{thread}[a](v) \]

All other rules simply preserve \( \preceq_\mu \). It can be shown that \( \preceq_\mu \) is reflexive, transitive and respects substitution.

In order to show how individual steps of the operational semantics change the cost graph (which we will in turn use to show the correspondence between the two versions of the semantics), we generalize serial composition to allow thread graphs to be composed with configuration graphs. In \( G \otimes G_2 \), the sink vertex of \( G_1 \) is joined to all source vertices of \( G_2 \) with edges of weight 1. Source vertices of \( G_2 \) which are auxiliary vertices are eliminated in the process.

\[ G_2 = a_1 \vdash (s_1, t_1, V_1, E_1, F_1) \otimes \cdots \otimes a_n \vdash (s_n, t_n, V_n, E_n, F_n) \]

and \( t_2 \vdash s_{G_2} \) is the sink vertex of \( G_2 \), then

\[ (s, t, V, E) \mathrel{\boxdot G_2} = (s, t_2, V \cup V_1 \cup \cdots \cup V_n, E \cup E_1 \cup \cdots \cup E_n \cup \{ (t, s', 1) | s' \not\vDash \}) \cup \{(t', s, \delta + 1) | (t, s', \delta) \in E_1 \cup \cdots \cup E_n \}, F \cup F_1 \cup \cdots \cup F_n \]

Note that the operation \( G_1 \boxdot G_2 \) is not necessary; ordinary serial composition works in this case since \( G_1 \) has a unique sink.

Lemma 9 examines the effect of a transition \( \tau \vdash G ; \mu \Rightarrow G' ; \mu' \) on the cost graph of a specified thread \( a \vdash (\delta, e) \vdash \mu' \). In other words, the lemma shows how the cost semantics behaves under “converse evaluation” or “head expansion”, a standard step in relating small-step and big-step semantics. Part 1 considers the case in which \( a \) is not one of the threads that transitions. In this case, if \( e \) evaluates to \( v' \) with cost graph \( \mu' \) under \( \mu' \), it evaluates to a related value and isomorphic cost graph under \( \mu \). Part 2 considers the more complex case in which \( a \) steps from \( e \) to \( e' \), adding the threads in \( \mu' \). In this case, if \( e \) evaluates to \( v' \) under \( \mu' \) and \( a \vdash (\delta', e') \vdash \mu'' \) produces the cost graph \( G' \), then \( e \) evaluates to a value related to \( v' \) with a cost graph that adds at least one vertex as an ancestor of \( G' \). This lemma will then be used to show that the cost semantics and the operational semantics correspond on the final values, and will later be used to show the Brent-type theorem that the cost graph is an accurate representation of the length of a prompt schedule.

**Lemma 9.** Fix \( \Delta \) and suppose that \( \cdot \vdash e : \tau \otimes w \) and \( \vdash a : \Sigma \). Let \( \mu = a \vdash (\delta, e) \vdash \mu \).

1. Suppose \( \mu \vdash G ; \mu' \Rightarrow G' ; \mu'' \) and \( \exists v \) such that \( e, \mu' \vdash v : g \) and \( v \preceq_\mu v' \).

2. Suppose \( \mu' = a \vdash (\delta', e') \vdash \mu_1 \vdash \mu_2 \) and \( \exists v \) such that \( e \vdash \mu_1 \vdash v : g \) and \( v \preceq_\mu v' \).

**Proof.** By induction on the derivations of \( e, \mu_1 \vdash v : g \) and \( e \vdash \mu_2 \vdash v' \).

We can now show the final result of this section: that if a well-typed \( \Pi \) program evaluates to a value using the operational semantics, the cost semantics will produce a cost graph for that program, along with the same final value.

**Theorem 2.** If \( \vdash a : (\delta, e) \vdash \Sigma \) and \( a \vdash \tau \otimes w \)

and \( a \vdash (\delta, e) \vdash \mu \Rightarrow r \vdash a \vdash (0, e') \vdash \mu' \)

and \( e \vdash (\delta', e') \vdash \mu'' \) final, then there exist \( v \) and \( g \) such that \( e, \mu_1 \vdash v : g \) and \( v \preceq_\mu v' \).

**Proof.** Since \( e' \vdash \mu_1 \vdash v' \), proceed by an inductive application of Lemma 9.

### 5. Cost Bounds for Prompt Scheduling Principle

The main result of this paper is showing a generalization of Brent’s Theorem (and similar results) to our language and our cost model, which takes into account responsiveness as well as total computation time. We show that a \( P \)-processor prompt schedule of a responsive parallel computation with work \( W \), span \( S \), foreground work \( W' \), foreground span \( S' \) and foreground width \( D \) completes in total time at most \( W/P + S \) with total response time at most \( DW/P + S \).

The bound on the computation time is known to be within a factor 2 of optimal. We will show that, in the worst case and for an online scheduler (one that does not know the computation ahead of time), the bound on the response time is also within a factor 2 of optimal.
The key step in showing the bound on the computation time is showing that a global transition step decreases the total work by $P$ or the total span by $1$. The intuition behind this proof, as in most proofs of Brent-type theorems, is that, by definition, a greedy scheduler (all prompt schedulers are greedy) will either execute $P$ instructions or execute all ready instructions (an entire "level" of the dag), decreasing the critical path by 1. To show the bound on the response time, we show that, if any foreground blocks are ready, a global step decreases the foreground work by $P$ or the foreground span by the number of ready foreground blocks. The intuition is similar to the above: a prompt scheduler will either execute $P$ foreground instructions or a level of every ready block. The proof of this lemma makes heavy use of part 2 of Lemma 9, which shows that a local transition on a thread decreases the work and span of the thread’s dag by at least 1.

**Lemma 10.** Fix $\Delta$ and suppose that $\cdot \vdash \mu : \Sigma$ and that $e \wedge j$ for all $a \leftrightarrow (\delta, e) \in \mu$. If $r; \mu \Rightarrow g r' ; \mu'$, then

1. $W(\mu', \Delta) \leq W(\mu, \Delta)$
2. $S(\mu', \Delta) \leq S(\mu, \Delta)$
3. $W(\mu, \Delta) - W(\mu', \Delta) \geq P$ or $S(\mu, \Delta) - S(\mu', \Delta) \geq 1$
4. $W(\mu', \Delta) \leq W'(\mu, \Delta)$
5. $S(\mu', \Delta) \leq S'(\mu, \Delta)$
6. $W(\mu, \Delta) - W'(\mu, \Delta) \geq P$ or $S'(\mu, \Delta) - S'(\mu', \Delta) \geq r' - r$ or $r' = r$.

**Proof.** See the appendix in the supplementary materials. □

The proof of the response time and computation time bounds is then straightforward.

**Theorem 3.** Fix $\Delta$ and let $e$ be such that $\cdot \vdash e : \tau \Rightarrow \emptyset$. Suppose $\cdot \vdash W : \emptyset$ and let $W = W(g) \text{ and } S = S(g)$ and $W' = W'(g)$ and $S' = S'(g)$ and $D = D(g)$. If $a \leftrightarrow (0, e) \Rightarrow g r; \mu$ and $\mu \Sigma$, then $T \leq \frac{W}{P} + S$ and $r \leq \frac{W'}{P} + S'$.

**Proof.** Let $\mu_0 = a \leftrightarrow (0, e)$ and $\mu_T = \mu$ and $t_0 = 0$ and $t_T = r$. We have a sequence $0; \mu_0 \Rightarrow g r_1; \mu_1 \Rightarrow \ldots \Rightarrow g r_T; \mu_T$.

For each $i$, let $W_i = W(\mu_i, \Delta)$ (and similar for $S_i, W'_i$ and $S'_i$). Note that $W_0 = W$ (and similar for $S, W', S'$) and that

$$W_T = S_T = W'_T = S'_T = 0$$

By Theorem 1, $e \wedge j$ for all $b \leftrightarrow (\delta, e) \in \mu$. By Lemma 10,

$$\frac{W_0}{P} + S_0 \geq 1 + \frac{W_1}{P} + S_1 \geq \ldots \geq 1 + \frac{W_T}{P} + S_T = 1$$

This immediately gives $\frac{W_T}{P} + S_T \geq 1$.

For each $i$, consider the quantity $D_i = D_i^w + S_i^w + r_i$. Note that for

1. $r_{i+1} = r_i$ and the other terms do not increase or decrease.
2. $W'_i - W_{i+1} \geq P$ and $r_{i+1} - r_i = |\text{FB}(\mu_i)| \leq D$ (the last inequality is by definition of $D$)
3. $S_i - S_{i+1} \geq |\text{FB}(\mu_i)|$ and $r_{i+1} - r_i = |\text{FB}(\mu_i)|$

In all three cases, the quantity above decreases or remains the same, for $r \leq \frac{W'}{P} + S'$. □

Clearly, $W/P$ and $S$ are both lower bounds on the computation time, so the bound of $W/P + S$ is within a factor of two of optimal. Recall that the response time is the sum over all foreground blocks of the time taken to execute $f$. Since $S(f)$ is a lower bound on the time to execute $f$, it is clear that $S'$, the sum of the spans over all blocks, is a lower bound on response time.

In order to argue that the bound on response time given by Theorem 3 is within a factor 2 of optimal, it remains to show that $DW'/P$ is also a lower bound on response time. This is not the case in general, but we will argue that it is a lower bound in the worst case assuming an online scheduler by presenting a class of computations on which the bound is tight. Consider a computation with total work $W'$ which consists only of $D = W'$ foreground blocks, each of which is sequential. Think of the work of the computation as $W'$ “bricks” which are distributed arbitrarily into $D$ stacks. At each step, a prompt scheduler will remove one brick from each of $P$ stacks (blocks). When a stack is empty, that block is complete and no longer counts toward the response time. Since an online scheduler only knows which blocks are ready (which stacks have a brick on top) and cannot base its decisions on how large each stack is (this would require knowing how long a block will take to execute, which is impossible in general), we may play a game against the scheduler. Start by placing two bricks on each stack. Keep the rest of the bricks hidden. At each step, when the scheduler removes a brick from a stack, place another brick at the bottom of that stack until you run out of bricks. In this way, all $D$ blocks will be ready for at most $\frac{W'}{P} - 2D$ steps (the number of steps it will take to run out of bricks), which will cause the response time to be at least $D \frac{W'}{P} - 2D \in O(D \frac{W'}{P})$.

### 6. Implementation and Examples

We developed a preliminary implementation of $\lambda \beta$ by building on a parallel extension of Standard ML [40, 41]. The implementation uses fork-join parallelism and futures, both built in to the parallel ML extension, to create threads, and supplies constructs to allow threads to be annotated as foreground or background. The implementation also consists of a simple prioritized scheduler. We did not extend SML’s type system to implement $\lambda \beta$’s type system.

We implemented several parallel interactive examples to show that the language features are easy to use and that prioritization can ensure responsiveness. All of the examples are very responsive when written with correct priorities but, as expected, quickly become unresponsive when all code is run in the background and the amount of computation is increased.

### Fibonacci Server

Our implementation of the Fibonacci server from Section 2 accepts integer inputs from the user (over standard input) in the foreground, and computes their Fibonacci numbers in the background, outputting the results asynchronously.

### Interactive Convex Hull

This program displays a window on which the user can click to add points. The points are displayed immediately (in the foreground). When a new point is added, a background computation is started to compute the convex hull of the current collection of points using the parallel Quickhull algorithm. The hull is displayed asynchronously when computed.

### Web Server

An interaction loop (in the foreground) listens for connections. When a connection is opened, the loop spawns a new thread which is immediately promoted to the foreground (using bgi([g(. . .)]) to listen for requests over that connection, allowing the main loop to immediately listen for more connections. The new connection thread waits for an HTTP request, which it serves in the foreground. The request is also added to a log (stored as a global mutable reference). Meanwhile, a background thread periodically checks the log and performs analytics on it (currently just tallying

---

4 Such a computation is not technically expressible in our language, since it will take some background work to start up the blocks, but if $W'$ is large enough, this starting background work and span can be neglected.
Abstractions and cost models for parallel programming have been studied extensively and many programming languages and extensions have been created [12, 18, 19, 25, 26, 28, 30, 31]. The focus of nearly all of this work on parallel computing has been maximizing throughput in compute-intensive applications. Our work builds on this prior work by proposing language abstractions, a cost semantics and a scheduling principle for responsiveness in interactive parallel applications. Our results build on prior work on type systems for staged computation, semantics and cost semantics for parallel computing, and also more remotely on the broader area of scheduling.

### Type Systems for Staged Computation

The type system of \( \lambda^0 \) is based on that of Davies [13] for binding time analysis, which he derived from linear temporal logic via the Curry-Howard correspondence. This work influenced much followup work on metaprogramming and staged computation [14, 29, 34, 42]. The idea behind these systems is to allow computation at a stage to create and manipulate, but not eliminate, a computation in a later stage. For example, a stage 1 computation can create a Stage 2 computation as a “black box” but cannot inspect that computation by, for example, pattern matching on its result. We specifically use a two-stage variant of the \( \Box \) modality of Davies [13], similar to that of Feltman et al. [17], which inspires some of our notation.

While our type system is essentially a staged type system, our operational interpretation is different from that of staged computation. In staged computation, evaluation proceeds in order of increasing stages. For example, in a two-staged system, all computations of the first stage are evaluated, followed by the second stage. In \( \lambda^0 \), we don’t order evaluation according to the stages—we allow them to occur concurrently. We know that a stage 1 (\( \Box \)) computation cannot possibly inspect a stage 2 (\( \Box \)) computation, but there is no need to wait for all stage 1 computations to complete before we can start a stage 2 computation. This is key to responsive and efficient parallel computation.

### Cost Semantics

The cost semantics for \( \lambda^0 \) can be viewed as instrumenting the evaluation to help the programmer to reason about cost. This idea of instrumenting evaluations goes back to the early 1990s [36, 37]. Cost semantics have proved to be particularly important in lazy languages (e.g., [37, 38]) and parallel languages (e.g., [4, 5, 41]). Our approach builds directly on the work of Blelloch and Greiner [5] and Spoonhower et al. [41], who use computation graphs represented as dags (directed acyclic graphs) to reason about time and space in functional parallel programs. These cost models, however, consider compute-intensive applications and do not consider interactive applications and responsiveness.

### Scheduling

In this paper, the scheduling principle presented, prompt scheduling, can guarantee the completion/run time and response time bounds for parallel interactive computations. Prompt scheduling, however, is a principle rather than an algorithm in the sense that our bounds do not take into account the cost of determining the schedule itself. We only bound the length of the schedule (which implies time) and the total response time. Our bounds are thus similar to Brent’s result for scheduling parallel (non-interactive) computations [9]. The design, analysis, and implementation of scheduling algorithms is a vast research topic, spanning multiple areas such as parallel computing, high-performance computing, operating systems, and queueing theory. Here, we briefly discuss a sample of the more closely related work.

The work on scheduling for parallel programs goes back to the 1970’s. Ullman [44], Brent [9], and Eager et al. [15] established the hardness of optimal scheduling and the greedy (or Brent) scheduling principle. Based on these early results, many scheduling algorithms have been developed and bounds have been proven [1, 2, 6, 7, 10, 16, 19–21, 35, 43]. More recent papers showed that priority-based schedulers can improve performance in practice [24, 45, 46], but offer no bounds. All of this work, however, considers non-interactive, compute-intensive applications. Muller and Acar [32] developed an algorithm for scheduling blocking parallel programs to hide latency, but do not consider responsiveness.

Scheduling is a key problem in the operating systems community [39]. There has been significant recent interest in making operating systems work well on multicore machines [3, 8]. The focus, however, has been on reducing contention within the OS and, as in the high-performance computing community, distributing resources to jobs so that they can run effectively. Scheduling within a job is less central to OS research.

There has been a great deal of work on scheduling for responsiveness in queueing theory [22]. This line of work assumes a continuous stream of independent jobs arriving for processing according to some stochastic process. Each job is processed or “served” by a single processor (server) that decides at every point in time which of the current jobs to run. The work on queueing-theoretic scheduling, however, has given almost no consideration to parallel jobs, typically assuming jobs to be sequential. Nor has there been any consideration of jobs which, as part of their execution, interact with the external world and thus might need to guarantee responsiveness bounds for specific blocks or tasks.

### 8. Conclusion

The problem of responsive parallel computing consists of writing parallel programs which perform both computational tasks and interaction, and running these programs so that they show good parallel speedup and remain responsive to input. We predict that this problem will become more important as parallel programming becomes the norm. The language features presented in this paper allow easy expression of responsive parallel programs. A promising area of future work would be to juxtapose these features with a new, well-engineered scheduler based on the prompt scheduling principle. The cost metrics and the cost model for reasoning about responsiveness developed in this paper will hopefully prove useful in such further studies of responsive parallelism.

### References


