Responsive Parallel Computation: Bridging Competitive and Cooperative Threading

Abstract
Very abstract abstract.

Categories and Subject Descriptors D.3.3 [Language Constructs and Features]: Input/output; F.1.2 [Modes of Computation]: Parallelism and concurrency

Keywords Parallelism, Interaction, Responsiveness, Cost Models

1. Introduction
Threads sharing an address space (typically that of an operating system process) is one of the oldest, widely applicable abstractions in computer science. Over the many years of research and practice, two forms of threading have emerged: competitive threading, and cooperative threading. While they both rely on essentially the same abstraction of threads, these two forms of threading differ and complement each other in their domain of applications, the form of scheduling that they use, and their performance goal as summarized by the Table 1.

Broadly used in interactive systems [27], the work on competitive threads goes back to early user interfaces such as Xerox’s STAR [42] and programming environments such as Cedar [45]. Such systems rely on threads to implement responsive interaction between the different components of the system, such as the I/O subsystem, the network etc, and between the system and the users [27]. For example, as the user types on their keyboard, a thread may read the input commands and start other threads to fulfill the requested tasks, which may involve creating threads to receive or send messages over the network, and synchronizing with other threads. Since in interactive systems responsiveness is key to user experience, maximizing responsiveness is the main performance goal in competitive threading. To guarantee responsiveness, threads are scheduled preemptively, usually based on priorities. In competitive threading, the number of processors has been traditionally small, with much of the work focusing only on sequential systems or small numbers of processors [6, 20, 22, 27].

Broadly used in parallel computing, the first uses of cooperative threading goes back to early parallel programming languages such as Id [4] and Multilisp [24] but has re-gained fresh popularity with the increasing mainstream availability of multicore computers. Parallel computing applications, drawn usually from areas such scientific computing, physical simulations, machine learning & AI, and discrete optimization are usually compute intensive and use threads to reduce run-time. To this end, they break up the computation into smaller threads that can be performed in parallel. For example, a parallel sorting algorithm may create threads to sort small inputs, and to merge sorted inputs into the final result. While it is technically possible to use competitive threading to implement parallel programs, the resulting programs are usually inefficient compared to performance with lighter-weight cooperative threads. This has led to the development of specialized programming languages for parallel systems, including OpenMP, Cilk [21], Fork/Join Java [33], Habanero Java [?], NESL [9], TPL [34], TBB [29], X10 [16], parallel ML [?], and parallel Haskell [31?]. To minimize the completion time of a parallel application, cooperative threading maximizes throughput by scheduling threads cooperatively, where each thread is run to completion without preemption. Theoretically and practically efficient non-preemptive schedulers have been studied extensively, e.g., work stealing [2, 3, 12], depth-first schedulers [10], and priority schedulers [28]. Since parallel computing applications can benefit tremendously from compute power, cooperative threading has been used on both small, moderate (10-100), and large numbers (many thousands) of processors.

As multicore computers become the common computing platform for essentially all applications, ranging from compute-intensive and interactive including those that mix both, many applications would benefit from a threading model that bridges competitive and cooperative threading. In such a model, an application can create both competitive and collaborative threads and expect them to be scheduled optimally, that is, to maximize throughput and responsiveness simultaneously. For example, an application that interacts with a user as it also performs compute-intensive tasks (e.g.
some machine learning task, graphics computations, geometric or scientific calculations) in parallel, mixes throughput-oriented parallel computation with responsiveness-oriented interaction.

In this paper, we propose a model that aims to combine the competitive and cooperative threading models in one unified model. Our starting point is graph-based cost models for parallel computing, which represent the execution of a parallel program with a Directed Acyclic Graph (DAG or simply dag) of instructions and the dependencies between them (e.g.,[12, 30]) and bound run time based on two measures: work (the total number of instructions) and span (length of the longest chain of dependent instructions) We extend this model to allow designation of two kinds of tasks foreground and background, which can intuitively be thought as subgraphs of the dag. Intuitively foreground tasks represent high-priority computations that should be executed promptly, while background task represent low-priority tasks that can be executed based on the availability of processors. We present a scheduling principle, called prompt scheduling, which generalizes the traditional greedy scheduling principle [3, 14]. We show that prompt schedules simultaneously bound both the completion time and responsiveness of a computation. To establish our bound, we rely on a new kind of cost measure, called foreground width, which is an upper bound on the number of foreground computations that can be executed in parallel. We show that our bounds are tight by establishing lower bounds that show that no online scheduling algorithm can perform better (within a factor of 2) without having advance knowledge of the structure of the computation dag. This appears to be a reasonable restriction on the scheduler, because practical schedulers do not usually have this kind of information. To establish the bound, we make an important well-formedness assumption that requires absence of priority inversions from the computation.

Our dag-based cost model allows reasoning about the run-time and responsiveness and run-time but like all other dag-based cost models, it has several important weaknesses: 1) it only applies to a specific execution dag of the program, and 2) it includes many lower-level execution-specific details, making it difficult to use as a high-level cost model. To close this gap, prior work developed language-based cost models, a.k.a., cost semantics [7, 8, 23], which allows reasoning about the cost at the level of the program rather than that of the execution. In our model, the gap between a program and its execution is even bigger because in addition to the aforementioned concerns, we have to consider priority inversions.

We therefore consider a lambda-calculus based language, called $\lambda^{ip}$, for writing parallel, interactive programs. The calculus supports cooperative threading by using the popular fork-join paradigm and competitive threading via two constructs for creating and synchronizing foreground (high priority) and background (low priority) computations. We equip $\lambda^{ip}$ with a type system, which is based on linear temporal logic (e.g. [17]), that guarantees that a foreground computation cannot depend on a background computation. Well-typed programs in $\lambda^{ip}$ thus have no priority inversions. We present a cost-semantics for $\lambda^{ip}$ that constructs cost-graphs by using a natural reduction semantics, which employs “big-step” evaluation judgments. We then show that this cost semantics is theoretically realizable by giving a small-step operational semantics that specifies the implementation of a language run-time, and proving that the semantics matches the cost semantics and thus the bounds.

To give some evidence that our proposed model can be implemented in practice, we present a prototype implementation as an SML library that extends the SML language with the constructs of $\lambda^{ip}$. Using this library, we have implemented a number of benchmark programs. Our results show that the theoretical bounds that we present predict the run-time of responsiveness of a number of interactive parallel programs.

### Table 1. A comparison of competitive and cooperative threading.

<table>
<thead>
<tr>
<th>Application</th>
<th>Scheduling</th>
<th>Goal</th>
<th>#Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Competitive</td>
<td>Interactive</td>
<td>Preemptive</td>
<td>Responsiveness</td>
</tr>
<tr>
<td>Cooperative</td>
<td>Parallel</td>
<td>Non-preemptive</td>
<td>Throughput</td>
</tr>
</tbody>
</table>

2. The DAG Model and Prompt Scheduling

**The traditional dag model.** Traditionally, we represent parallel computations using directed acyclic graphs or dags. Vertices of the dag represent atomic instructions of the computation, which are each assumed to require one step of processor time. Edges represent dependencies between these instructions: an edge from $u_1$ to $u_2$ indicates that the instruction represented by $u_1$ must execute before $u_2$. For a dag $g$, we write $u \succeq u'$ to indicate that $u$ is an ancestor of $u'$ in $g$. When it is clear from the context, we drop $g$ and simply write $u \succeq u'$.

For example, consider the classic function $\text{fib}(n)$ which computes the $n$th Fibonacci number in parallel by parallelizing the two recursive subcalls to $\text{fib}(n-1)$ and $\text{fib}(n-2)$. We can write such a function as

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

We can represent an execution of $\text{fib}(3)$ as shown in Figure 1. For brevity, each vertex represents a call to $\text{fib}$ instead of an
Our model. To model parallel programs that mix competitive and collaborative threading, we need to extend the traditional model to account for the latency incurred by certain instructions and to allow computations to be prioritized.

To allow prioritization of computations, we allow certain portions of a dag, called foreground blocks to be specified as foreground or high-priority computations. A foreground block is specified by its source and sink vertex. A foreground block with source \( s \) and sink \( t \) is written \( \langle s, t \rangle \) and is the vertex-induced subdag of \( g \) consisting of all \( u \in V \) such that \( s \not\lesseqgtr u \) and \( u \not\lesseqgtr t \). As a key requirement, we require that no vertex of a foreground block except for the source to have an incoming edge from outside the block. We say that a dag is well-formed if this requirement holds for all its foreground blocks. Intuitively speaking, well-formedness requires the absence of priority inversion, which causes a high-priority computation to depend on (and thus wait for) a low-priority one.

To account for latency, we allow edges to be weighted by a natural number \( \delta \) that represents the number of steps that the target of the edge waits to become ready after the source starts its execution. More specifically, for a weighted edge \((u,v,\delta)\), if \( \delta = 1 \), then \( u \) incurs no latency and \( v \) may execute on the next step. If \( \delta > 1 \), then \( u \) incurs a latency of \( \delta \) and \( v \) may execute anytime \( \delta \) steps after \( u \) starts executing.

Mathematically speaking, we write a dag as a tuple \((s, t, V, E, F)\) consisting of a source vertex \( s \), a sink vertex \( t \), a set \( V \) of vertices, (where \( s,t \in V \) and \( s \not\lesseqgtr t \)), a set \( E \) of weighted, directed edges, and a set \( F \) of foreground blocks. Such a dag is well-formed if no vertex of a foreground block except the source has an edge from outside the block.

As an example, consider a simple program \( \text{fib}_\text{hello} \) which mixes computation and interaction by computing \( \text{fib}(3) \) and, in parallel, asking the user two questions and responding to the user’s answers. By using a keyword \( \text{fg} \) to indicate foreground computations, we can write such a program as follows.

```plaintext
function hello () =
  if i <= 0 then ()
  else
    let
      _ = output(‘What is your first name?’)
      x = input ()
      _ = output(‘What is your last name?’)
      y = input ()
      in _ = output(‘Hello, ’ .. x .. ‘ ‘ .. y) end
  end

function fib_hello () =
  par(fib 3, fg (hello ()�)
```

Figure 2 illustrates the dag for this program. The foreground computation is drawn within a box. The edge weights \( \delta_1 \) and \( \delta_2 \) stand for the latency incurred by the two input instructions.

Cost metrics: work, span, width. Traditionally in parallel computing with cooperative threads, the work \( W \) of a dag 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. Intuitively, 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.

For our model, we extend the traditional model to account for the edge weights and prioritized computations. As we show in the rest of this section, these extensions suffice to prove a bound on both the completion time and the responsiveness of interactive parallel computations. As with the traditional model, we define work as the total number
of vertices by change the definition of span to include edge weights: span is the longest weighted path in the dag. This captures the notion that the time spent blocking on inputs should not be counted as computational work because a scheduler need not dedicate a processor to the blocked thread. The span, on the other hand, takes these delays into account since the computation cannot complete until all of the inputs are available. 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.

In addition, we distinguish between foreground and background work and span. For a foreground block \( f \) which is part of \( g \), we write \( W_b(f) \) and \( S_b(f) \) for the work and span, respectively of the task. For a graph \( g = (s, t, V, E, F) \), the foreground work \( W^o(g) \) and foreground span \( S^o(g) \) as the sum over all foreground blocks:

\[
W^o(g) = \sum_{f \in F} W_b(f) \\
S^o(g) = \sum_{f \in F} S_b(f)
\]

To bound the response time, we define a new notion, which we call foreground width. We say that 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 \( F' \subseteq F \) are independent if for all \( f_1, f_2 \in F' \), \( f_1 \) and \( f_2 \) are not serial. We define the foreground width \( D \) of a graph as

\[
D = \max \{|F'| \mid F' \subseteq F \land F' \text{ are independent}\}
\]

That is, \( D \) is the maximum number of independent foreground blocks.

**Prompt Schedules.** A schedule is an assignment of vertices to processor-step pairs such that if a vertex \( u \) is executed at step \( i \), it is ready at step \( i \). A vertex \( u \) is ready if all of the ancestors of \( u \) have executed and \( u \)'s latency requirements (if any) have expired. As in prior work, a schedule is greedy if as many ready vertices as possible are executed at each step (up to the number of ready vertices or the number of processors). We define a schedule to be prompt if it is greedy and if as many foreground vertices as possible are executed at each step, up to the number of ready foreground vertices or the number of processors. In other words, a prompt schedule does as much work as possible and always prioritizes foreground vertices.

We write \( T_p \) for the time to execute a given parallel computation on \( P \) processors using a given schedule. The \( P \)-processor response time \( R_p(f) \) of a foreground block \( f = \langle s, t \rangle \) for a given execution is the number of steps between when \( s \) becomes ready and when \( t \) is executed (inclusive). The total \( P \)-processor response time of an execution, \( R_p \), is equal to the sum of \( R_p(f) \) for all foreground blocks in the dag.

**Theorem 1.** Consider a parallel computation represented by a well-formed graph \( g = (s, t, V, E, F) \). If this computation is scheduled with a prompt schedule, then \( T_p \leq \frac{W(g)}{P} + S(g) \) and \( R_p \leq D \frac{W(g)}{P} + S^o(g) \).

**Proof.** Since a prompt scheduler is, by definition, greedy, the bound on \( T_p \) follows from the variant of Brent’s theorem for weighted dags [2?]. We now show the bound on the response time.

Consider running the computation. At each step, a token is placed into one of two buckets, \( R_1 \) and \( R_2 \). At a step \( i \), suppose there are \( n_i \) ready foreground vertices which come from \( N_i \) foreground blocks. If \( n_i > P \), then place \( N_i \) tokens in bucket \( R_1 \) and \( P \) tokens in a work bucket. If \( n_i \leq P \), then place \( N_i \) tokens in bucket \( R_2 \).

Since \( N_i < D \), for every \( P \) tokens placed in the work bucket, at most \( D \) tokens are placed in bucket \( R_1 \). So, at any time, if \( W_b \) is the number of tokens in the work bucket, we have

\[
\frac{R_1}{D} \leq \frac{W_b}{P}
\]

At the end of the computation, the total number of tokens in the work bucket is at most \( W^o(g) \) since at steps when \( n_i > P \), a prompt scheduler will execute only foreground vertices. Thus, at the end of the computation,

\[
R_1 \leq D \frac{W^o(g)}{P}
\]

Now consider a token placed in \( R_2 \) at step \( i \). This token corresponds to a foreground block \( f \) for which at least one vertex is ready at step \( i \). Let \( g_i \) be the sub-dag consisting of vertices of \( f \) that have not been executed after step \( i \). Extend this in the following way to form a dag \( g_i^* \). All vertices and edges in \( g_i \) are also in \( g_i^* \). In addition, for all edges \((u, v, \delta)\) where \( u \) is in \( f \setminus g_i \) and \( v \) is in \( g_i \) (that is, \( u \) has been executed by the end of step \( i \) and \( v \) has not), if \( u \) was executed in step \( i - j \), add to \( g_i^* \) vertices \( u_1, \ldots, u_{\delta-j-1} \) and edges \((u_1, u_2), \ldots, (u_{\delta-j-1}, u, 1)\) (that is, add a chain of length \( \delta - j - 1 \) before \( u \)). Note that because \( g_i \) is well-formed, no vertex of \( g_i \) may have edges from outside \( f \) in \( g \) (except the source of \( f \), but the source must be ready or executed at step \( i \) or no vertex of \( f \) would be ready), and so the vertices of \( f \) that are ready at the start of step \( i + 1 \) are exactly those vertices that are contained in \( g_i \) and have in-degree zero in \( g_i^* \). By the definition of a prompt scheduler, it must be the case that all ready vertices of \( f \) at step \( i \) are executed at step \( i \), and so do not appear in \( g_i^* \). In addition, for any vertex \( v \) that is suspended at the start of step \( i \) and so has a chain before it in \( g_i^* \), the chain is decreased by one vertex in \( g_i^* \). Together, these facts mean that every vertex in \( g_i^* \) with in-degree zero is not present in \( g_{i+1}^* \) and so the longest path in \( g_{i+1}^* \) is one shorter than the longest path in \( g_i^* \). Since the longest path in \( g_0^* \), by definition, has length \( S_f(g) \), at most \( S_f(g) \) tokens can be placed in \( R_2 \) corresponding to \( f \). In total,

\[
R_2 \leq S^o(g)
\]

Taking the total response time to be \( R_1 + R_2 \) gives the desired result. \( \Box \)
Lower Bounds for Online Scheduling. Given a parallel computation represented by a well-formed graph $g = (s, t, V, E, F)$, Theorem 4 gives an upper bound on the run time and the responsiveness of a prompt schedule. We now show that this bound is tight by exhibiting a class of computations that require at least half as many steps on any online scheduling algorithm that has no prior knowledge of the computation dag.

Showing that the run-time bound is tight is straightforward. Since, $W(g)/P$ and $S(g)$ are both lower bounds on the computation time, it follows that the bound $W(graph)/P + S(g)$ is within a factor of two of optimal.

For the responsiveness bound, we now establish similar lower bound. Recall that the response time is the sum over all foreground blocks $f$ of the time to execute $f$. Since $S^*(f)$ is a lower bound on the time to execute $f$, $S^*(g)$, which is the sum of the spans over all blocks, is a lower bound on response time. Thus, to establish a 2-factor approximation(?) show that $DW^*/P$ is also a lower bound on response time. To this end, consider a computation with total work $W^* + D$ which consists only of $D < W^*$ foreground blocks, each of which is sequential. The computation simply forks off, doing $D$ work, foreground blocks, and proceeds to complete them. 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 the $P$ 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 least $\frac{W^* - 2D}{P}$ steps (the number of steps it will take to run out of bricks), which will cause the response time to be at least $D\frac{W^* - 2D}{P} \in O(D\frac{W^*}{P})$.

3. Language

In this section, we introduce a core calculus called $\lambda^p$, which extends a simply-typed lambda calculus with constructs for I/O, parallelism and priority. The type system of $\lambda^p$ separates subcomputations by priority (foreground or background). In Section 3.3, we equip $\lambda^p$ with a cost semantics that generates a dag of the form described in Section 2 for a $\lambda^p$ program. This allows one to read off the cost metrics and apply the computation time and response time bounds to a parallel interactive program.

3.1 The core calculus $\lambda^p$

The expression syntax of $\lambda^p$ syntax is presented in Figure 3. Most features are fairly standard for a simply-typed lambda calculus. The expressions $e$ include the standard introduction and elimination forms for base types, functions, pairs and sums: natural numbers $n$, $\lambda$-abstractions, application, pairs, projection, injection, and case analysis. Recursion is possible through the fixed point operator $\text{fix } x. \tau e = e$.

We will introduce the non-standard features of the language through a series of examples. In the examples, for convenience, we will use an ML-style language with constructs like let binding. These constructs can be desugared into the core calculus above in a straightforward way. We also assume additional base types like strings and booleans. Extending $\lambda^p$ to handle these types would be straightforward.

Fork-Join Parallelism. Our starting point for developing a language with responsive parallelism is fork-join parallelism. A parallel tuple $e_1 \parallel e_2$ (written $\text{par}(e_1, e_2)$ in concrete syntax) evaluates $e_1$ and $e_2$ in parallel. For example, we can write a function that computes the $n^{th}$ Fibonacci number by using the standard recursive algorithm as follows.

$$\text{function fib n = }$$
$$\text{if } n <= 1 \text{ then } n$$
$$\text{else let (a, b) = par (fib (n - 1), fib (n - 2)) in a + b}$$

Since the two recursive calls are independent, they can be performed in parallel, leading to an “embarrassingly parallel” algorithm. This function is not an efficient way to compute Fibonacci numbers but is used to illustrate a compute-intensive parallel computation.

Challenge: Responsive Interaction. To see the challenge of responsive interaction, we extend our parallel language with input and output: input $\text{inp}(d)(x. e)$ binds user input to the variable $x$ in evaluating $e$ and output $\text{out}(e)$ outputs the value of $e$ to the user. The annotation $d$ relates to the cost semantics; we will ignore it for now. Since our techniques do not depend on exactly how these operations perform interaction (e.g. via a console, through GUI operations, over a network), we leave this aspect unspecified. Since natural numbers are the only interesting base type of $\lambda^p$, we only provide input and output over natural numbers in the formal language. Extending the I/O constructs to other base types such as strings would not pose any technical problems, and we do so in our examples. We can now write a simple interactive program that asks the user two questions, responds, and repeats, $i$ times.

$$\text{function quest i = }$$
$$\text{if } i <= 0 \text{ then } ()$$
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.

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

This code seems to have the desired functionality. However, there would be no guarantee that it would be responsive since we have given the language no way to distinguish between the interactive thread, which must be run with high priority to ensure responsiveness, and the many lower-priority computation threads which will be spawned by `fib`. A scheduler might get lucky, but in general, responses to the user could get arbitrarily delayed as the computation threads starve the interaction.

**Priority annotations.** The `fib_quest` function illustrates a fundamental challenge in parallel computing: interaction requires certain blocks of computation, such as those that perform interaction, to be given high priority, but existing parallelism constructs and cost models do not support such prioritization. To enable priority, we introduce two language constructs: `fg(e)` annotates the expression `e`, which we refer to as a foreground block, as running in the foreground (with high priority). The annotation `bg(e)` embeds a background computation `e`, which runs with low priority, inside a foreground block.

In `λip`, we can write the `fib_quest` example so that it runs `quest` in the foreground. As shown below, since it returns to the background, `quest` returns a background value.

```verbatim
function quest i =
  if i <= 0 then bg ()
  else
    let _ = output(‘‘What is your name?’’) in
    let nm = input () in
    let _ = output(‘‘What is your quest?’’) in
    let qu = input () in
    let _ = output(‘‘You may pass, ’’ ˆ nm) in
    quest (i - 1)
```

```verbatim
function fib_server () =
  let n = input () in
  if n < 0 then ()
  else
    output (fib n);
    bg (quest (i - 1))
```

```verbatim
function main () =
  fg (fib_server ())
```

**Figure 4.** Parallel Fibonacci without (left) and with priorities (right).

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 4. The expression `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.

### 3.2 Priority inversion and type system

The language as described so far allows programs which cannot be implemented responsively. Consider the following variant of the Fibonacci server:

```verbatim
function fib_server () =
  let n = input () in
  if n < 0 then bg ()
  else
    let fibn = bg (fib n)
    in
    output (fg (fibn));
    fib_server ()
```

```verbatim
function main () =
  fg (fib_server ())
```

This program starts a background thread to compute Fibonacci, but then immediately demands and outputs the result before prompting for the next input. Intuitively, this program will not be responsive because a foreground thread is waiting on a potentially long-running background thread. This relates to the classic idea of priority inversion. It also relates to the notion of well-formed dags defined in Section 2. We will show that programs which have no priority inversions (as determined by the type system we present below) produce well-formed dags in the cost semantics. To prevent the responsiveness problems that result from priority inversions, we introduce a type system for `λip` which makes explicit the distinction between foreground and background computations and prevents such priority inversions. Our type system is based on ideas drawn from linear temporal logic and other
Figure 5. Static semantics of $\lambda^p$ type systems based on LTL, especially those for staged computation. The relationship with staged languages is discussed further in Section 7.

The types $\tau$ include two base types: unit and natural numbers, as well as functions, binary tuples and binary sums and the circle type $\bigcirc \tau$, which represents handles to background threads. The main typing judgment for $\lambda^p$ is

$$\Gamma \vdash e : \tau @ w$$

The types $\Sigma$ have entries of the form $\tau @ w$, 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 $bg(e)$ or starting a foreground block with $fg(e)$. If $e$ is background code of type $\tau$, the expression $bg(e)$ starts a background thread of type $\tau$ and immediately returns a handle of type $\bigcirc \tau$ in the foreground. If $e$ types in the foreground with type $\bigcirc \tau$, i.e. it will evaluate to (a thread running) background code of type $\tau$, the expression $fg(e)$ has type $\tau$ at $\mathcal{B}$. Typing $fg(e)$ at $\mathcal{B}$ 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 $\text{nat}$ to 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 $\bigcirc$, and could easily be expanded to allow any “mobile” type, including unit, sums and products (but not functions or $\bigcirc \tau$).

3.3 Cost Semantics

In this section, we define a cost semantics which constructs a cost dag of the form described in Section 2 for a $\lambda^p$ 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, and are labelled with the delay that an input or output operation incurs, if any.

The cost semantics in Figure 6 generates thread graphs for expressions. The judgment $e; u, g \vdash^t v$ indicates that the expression $e$ evaluates to $v$ and has cost graph $g$ in the presence of $u$. Values $v$ consist of irreducible expressions, plus a new form of thread handle 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:

\begin{align*}
\text{Stefan's remark:} \quad & \text{Update the syntax to stage} \\
& \text{this correctly} \\
S.1
\end{align*}

Values $v := \langle \rangle | n | \lambda x : \tau. e | \langle v, v \rangle | \text{inl}(v) | \text{inr}(v) | \text{tid}(b) | \text{thread}[u](v)$

Many of the rules for the sequential components of the language and parallel tuples are based on the cost semantics of Spoonhower et al. [44], with nontrivial modifications to allow the representation of in-progress computations. The rules for generating and joining with background threads ($bg(e)$ and $fg(e)$, respectively), are based on Spoonhower’s treatment of futures [43], which share the property that an asynchronous
Expression cost semantics $e \triangleright^\alpha_v g$

<table>
<thead>
<tr>
<th>$e$</th>
<th>$e \triangleright^\alpha_v g$</th>
<th>$u$</th>
<th>$v$</th>
<th>$g$</th>
<th>$u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_{\text{val}}$</td>
<td>$e_1 \triangleright^\alpha_v x : \tau. e_2$</td>
<td>$e_1 e_2 \triangleright^\alpha_v v : g_1 @ g_2 @ [u] @ g_3$</td>
<td>$[v/x] e_1 \triangleright^\alpha_v v' : g_1$</td>
<td>$u$</td>
<td>$e_1 \triangleright^\alpha_v (v_1, v_2) : g$</td>
</tr>
<tr>
<td>$e_{\text{snd}}$</td>
<td>$e_1 \triangleright^\alpha_v v : g$</td>
<td>$u$</td>
<td>$e_1 \triangleright^\alpha_v (v_1, v_2) : g_1$</td>
<td>$e_1 \triangleright^\alpha_v v_1 : g_1$</td>
<td>$e_2 \triangleright^\alpha_v v_2 : g_2$</td>
</tr>
<tr>
<td>$e_{\text{inr}}$</td>
<td>$\text{inr}(v)$</td>
<td>$g$</td>
<td>$e_1 \triangleright^\alpha_v (v_1, v_2) : g_1$</td>
<td>$e_1 \triangleright^\alpha_v v_1 : g_1$</td>
<td>$e_2 \triangleright^\alpha_v v_2 : g_2$</td>
</tr>
<tr>
<td>$\text{case}(e)([x:e_1; y:e_2])$</td>
<td>$e_1 \triangleright^\alpha_v v : g_1 @ [u] @ g_2$</td>
<td>$e_1 \triangleright^\alpha_v v_1 : g_1$</td>
<td>$e_2 \triangleright^\alpha_v v_2 : g_2$</td>
<td>$e_1 \triangleright^\alpha_v v_1 : g_1$</td>
<td>$e_2 \triangleright^\alpha_v v_2 : g_2$</td>
</tr>
</tbody>
</table>

Figure 6. Cost Semantics

\[
\begin{align*}
[u] & = (u, u, [u], \emptyset, \emptyset) \\
([u], u_2, \delta) & = (u_1, u_2, [u_1, u_2], ([u_1, u_2, \delta]), \emptyset) \\
(s_1, t_1, V_1, E_1, F_1) \oplus (s_2, t_2, V_2, E_2, F_2) & = (s_1, t_2, (V_1 \cup V_2, E_1 \cup E_2 \cup \{(t_1, s_2, \delta), F_1 \cup F_2)) \\
g_1 @ g_2 & = g_1 \oplus g_2 \\
(s_1, t_1, V_1, E_1, F_1) \otimes (s_2, t_2, V_2, E_2, F_2) & = (s, t, (V_1 \cup V_2 \cup \{s, t\}, E_1 \cup E_2 \cup \{(s, s_1, 1), (s, s_2, 1), (t_1, t_1), (t_2, t_1), F_1 \cup F_2) \\
g \downarrow_0 & = \emptyset \\
(s, t, V, E, F) \downarrow_\delta & = [a] \oplus \emptyset \\
(s, t, V, E, F) \times & = (s, t, V, E, F \cup \{(t\rangle\}} \\
(s, t, V, E, F) \triangleright^\alpha & = (s, t, V, E, F \cup \{\langle t\rangle\}) \\
\text{fix } x : \tau \text{ is } e/x & e \triangleright^\alpha v : g \\
\text{fix } x : \tau \text{ is } e \triangleright^\alpha v : g & u \triangleright^\alpha v : g \\
\text{fix } x : \tau \text{ is } e \triangleright^\alpha v : g & u \triangleright^\alpha v : g \\
\end{align*}
\]

Figure 7. Graph building and composition operations

\[
\begin{align*}
\end{align*}
\]

Figure 8. From left: $g_1 \oplus_\delta g_2, g_1 \otimes g_2,$ and $g\otimes^\alpha u.$ 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 7. 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 $g_1 \oplus g_2$ which joins the sink of $g_1$ to the source of $g_2$ by an edge of weight 1 (a more general form, $\oplus_\delta$, uses an edge of weight $\delta$, as shown in Figure 8). The empty graph $\emptyset$ acts as a unit for the $\oplus$ 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 8). If one of the graphs is empty, the other is simply composed with $s$ and $t$. The rule for $bg(e)$ uses the left parallel composition operator $[43].$ If $g$ is the cost graph for $e,$ the graph $g\otimes^\alpha u$ “hangs $g$ off of” vertex $u$ (Figure 8). 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 $fg(e)$ 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 is $b$ 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 \( \circ \). This operation produces a foreground block \( (s, t) \) if \( s \) and \( t \) are the source and sink of \( g \) and \( g \) has no ancestors (i.e. is not a join point), or a foreground block \( (i, t) \) if \( t \) is the sink of \( g \) and \( g \) depends on other threads. Finally, the rule adds an edge of weight \( \delta \), where \( \delta \) is chosen nondeterministically from \( \Delta(d) \).

**Lemma 1.** If \( \cdot \vdash e : \tau \circ \circ \) and \( e \circ A \vdash \circ \circ \) \((s, t, V, E, F)\), then \( F = \emptyset \) and for all \((v, u, \delta) \in E\), we have \( v \in V \).

**Proof.** By induction on the derivation of \( \cdot \vdash e : \tau \circ \circ \). \( \square \)

**Lemma 2.** If \( \cdot \vdash e : \tau \circ w \) and \( e \circ A \vdash \circ \circ \) \( g \), then \( g \) is well-formed.

**Proof.** Let \( g = (s, t, V, E, F) \). By induction on the derivation of \( e \circ A \vdash \circ \circ \) \( g \). The interesting case is the rule for \( \text{fg}(e) \), which adds a foreground block. By inversion, \( e \circ A \vdash \circ \circ \) \( (s', t', V', E', F') \) and by inversion on the typing rules, \( \tau = \bigcirc \cap w \) and \( \cdot \vdash e' : \tau' \circ \circ \). By Lemma 1, \( F = \emptyset \) and for all \((v, u, \delta) \in E'\), we have \( v \in V' \). By the cost semantics, we have \( F = \{ (s', t') \} \) and \( E = E' \cup \{(u_2, t), (u_1, u_2, t_1)\} \). Since no edge is added, \( e \circ A \vdash \circ \circ \) \( g' \) for all \((u, v, \delta) \in E \). No other rule adds an edge to a vertex of a subdag except to its source, so well-formedness is preserved. \( \square \)

### Stefan’s remark 2:

figure out what to do with the rest of this section

S. 2

The cost semantics allows us to assign costs (work, span, etc.) to programs, as represented by thread pools. The work and span of a thread pool that is in the middle of execution can be thought of as the remaining work and span of the program. The work of a thread pool \( \mu \) is written \( W(\mu) \) and is defined as the maximum work over all dags that can be generated from \( \mu \):

\[
W(\mu, \Delta) = \max \{ W(G) \mid \mu \vdash \circ \circ G \}
\]

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

In the remainder of this section, we show important properties of the cost semantics and its correspondence with the operational semantics. Lemma 3 relates the invariants of the type system to cost graphs: cost graphs generated by \( \circ \) expressions have no nested foreground blocks and edges from other threads occur only at joins.

**Lemma 3.** If \( \cdot \vdash e : \tau \circ \circ \) and \( \cdot \vdash a : \delta \circ \circ \) \( \Sigma, a \circ \circ \) \( \tau \circ \circ \) and \( a \circ \circ \) \( (\delta, e) \circ \circ \mu \), then \( G = a \circ \circ \) \( g \circ \circ a_1 \circ \circ ... \circ \circ a_n \circ \circ (s_n, t_n, V_n, E_n, F_n) \) and \( g = (s, t, V, E, F) \), then

1. \( F = \emptyset \)
2. if \( e \circ w_j \), then there does not exist \((b, u, \delta) \in E \) where \( b \) is a thread identifier
3. if \( e \circ w_j \), then there does not exist \((b, u, \delta) \in E \) for any \( u \neq s \).
4. for all \( a_i \), we have \( s_i \ni t \) if and only if \( a_i \in i_{\mu}(e) \)

**Proof.** Parts 1-3 are by induction on the derivation of \( \cdot \vdash e : \tau \circ \circ \). In part 3, inversion on \( e \circ w_j \) is used to show that any graphs that may be serially composed before a join are empty. Part 4 is by lexicographic induction on the derivations of \( \cdot \vdash e : \tau \circ \circ \) and \( \cdot \vdash a : (0, e) \circ \circ \mu : \Sigma, a \circ \circ \) \( \tau \circ \circ \). See the appendix in the supplementary materials for details. \( \square \)

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 \( RFB(\cdot) \) over graphs. The ready foreground blocks of a graph are the foreground blocks that have ready vertices. Formally,

\[
RFB((s, t, V, E, F)) = \{ (s', t') \in F \mid (s', t') \in F \cap s \not\ni G \}
\]

**Lemma 4.** Fix \( \Delta \). Suppose \( \mu = a_1 \circ \circ (\delta_1, e_1) \circ \circ ... \circ \circ a_n \circ \circ (\delta_n, e_n) \) and \( \cdot \vdash \circ \circ \mu : \Sigma \) and \( e_i \) \( \forall i \) for all \( e_i \) and \( \mu, \mu \vdash \circ \circ G \), where

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

Then

1. \( \text{iready} \mu(a_i) \) if and only if \( s_i \not\ni G \).
2. \( RFB(\mu) = \{ a_i \mid s_i \ni G \cap (s', t') \in \text{RFB}(G) \} \cup \{ a_i \mid s_i \ni G \cap (s', t') \in \text{RFB}(G) \}

**Proof.** See the appendix in the supplementary materials. \( \square \)

Next, we show that the operational semantics and cost semantics agree on the values produced by an expression. One complication in showing such a result is accounting for the value \( \text{thread}[v](u) \) which is produced by the cost
semantics but not the operational semantics\(^1\). We therefore show that the cost semantics and the operational semantics are equivalent up to a relation \(\preceq\), which relates the two forms of thread handle. We define \(\preceq\) inductively. The important rules are the ones for thread handles:

\[
\begin{align*}
\text{thread} & (\text{a}(v)) \preceq \text{tid}[\text{b}] & \text{tid}[\text{b}] & \preceq \text{tid}[\text{b}] \\
v \preceq \nu' \quad & \quad \text{thread} & (\text{a}(v)) \preceq \text{thread} & (\text{a}(v'))
\end{align*}
\]

All other rules simply preserve \(\preceq\). It can be shown that \(\preceq\) 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 \(G_1 \parallel G_2\), 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.

If \(G_2 = a_1 \leftarrow (s_1, t_1, V_1, E_1, \mu_1) \ominus \ldots \ominus a_n \leftarrow (s_n, t_n, V_n, E_n, \mu_n)\), and \(t_\text{Gz}\) is the sink vertex of \(G_2\), then

\[
\begin{align*}
(s, t, V, E, F) \parallel G_2 &= (s, t_\text{Gz}, V \cup V_1 \cup \ldots \cup V_n) \\
&\quad \cup \{(t', s') \mid (t', s', 1) \in E \} \\
&\quad \cup \{(t', s') \mid (t', s', 1) \in E_n \} \\
&\quad \cup \{(t', s', 1) \mid (t', s', 1) \in E_n \} \\
&\quad \cup \{(t', s', 1) \mid (t', s', 1) \in E_n \}
\end{align*}
\]

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

Lemma 5 examines the effect of a transition \(r, \mu \Rightarrow \nu; \mu'\) on the cost graph of a specified thread \(a \leftarrow (s, t, e)\) of \(\mu\). 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 where \(a\) is not one of the threads that transitions. In this case, if \(e\) evaluates to \(v'\) with cost graph \(g'\) 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 \(\nu'\) and \(a \leftarrow (s', t', e') \ominus \mu'\) 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 sequence.

Lemma 5. Fix \(\Delta\) and suppose that \(\cdot \vdash_{\Sigma} e : \tau @ \omega\) and \(\cdot \vdash_{\Sigma} \mu_0 : \Sigma\). Let \(\mu = a \leftarrow (s, t, e) \ominus \mu_0\).

\[1. \text{Suppose } r, \mu \Rightarrow g; r'; \mu'\text{ and } e; \mu' \Downarrow_{\omega} v'; g'. \text{ There exist } v\text{ and } g\text{ such that } e; \mu \Downarrow_{\omega} v; g\text{ and } g \equiv g'\text{ and } v \preceq v'. \]

\[2. \text{Suppose } \mu' = a \leftarrow (s', t', e') \ominus \mu'\text{ and } r, \mu \Rightarrow g; r'\mu'\text{ where } e \mid \mu \Downarrow_{\omega} (s', t') \mid \mu \ominus \mu'\text{ and } e'; \mu' \Downarrow_{\omega} v'; g'\text{ and } g' = g'\text{ and } v \preceq v'. \]

Proof. By induction on the derivations of \(e; \mu' \Downarrow_{\omega} v'; g'\) and \(e \mid \mu \Downarrow_{\omega} (s', t') \mid \mu \ominus \mu'\). See the appendix in the supplementary materials for details.

We can now show the final result of this section: that if a well-typed \(\lambda^p\) 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 \(\cdot \vdash_{\tau @ w}\) \(a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\) and \(r; a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\) \(r; a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\) \(a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\) \(r; a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\) \(r; a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\)

and \(a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\) \(a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\) \(a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\) \(a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\) \(a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\) \(a \leftarrow (s, t, e) \mid \Sigma, a \sim \tau @ w\)

Proof. Since \(e' val\), we have \(e'; \mu' \Downarrow_{\omega} e'; 0\). Proceed by an inductive application of Lemma 5.

4. \textbf{Realization}

The operational semantics of \(\lambda^p\) consists of two components: local and global. This separation, and much of our notation, is drawn from Harper [26]. The local semantics concerns individual threads, and indicates how expressions transition. Selected rules are presented in Figure 9 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 \(\text{tid}[\text{b}]\).

The local transition judgment is

\[e \mid \mu \Downarrow_{\omega} (s', t') \mid \mu \ominus \mu'\]

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 \(\Delta : \text{InputIDs} \rightarrow 2^{\mathbb{N}}\), 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 \(e_1 e_2\), 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

\[1\]

\[1\]The form \(\text{tid}[\omega]\) is not explicitly produced by the cost semantics, but can be carried through since it is an irreducible expression and therefore evaluates to itself under the cost semantics.
Lemma 6 (Local Progress). If \( \cdot \vdash_{\Sigma} e : \tau @ w \) and \( \cdot \vdash \mu : \Sigma, a \sim \tau @ w \), then either e val or \( e | \mu \Rightarrow^\lambda_{a} (\delta, e') \ | \mu \otimes \mu' \) or there exists \( b \leadsto (\delta_b, e_b) \in \mu \) such that \( \delta_b > 0 \) or \( e_b | \mu \Rightarrow^\lambda_{a} (\delta'_b, e'_b) \ | \mu \otimes \mu' \).

Proof. By induction on the derivation of \( e | \mu \Rightarrow^\lambda_{a} (\delta, e') \ | \mu \otimes \mu' \).

The body of the abstraction using standard capture-avoiding substitution. A parallel tuple \( e_1 \parallel e_2 \) 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 \( \text{tid}[b] \). Note that, while threads spawned by parallel tuples and threads spawned by \( \text{bg}(e) \) are treated identically by the semantics (i.e. they are stepped with the same transitions and not distinguished in the thread pool), the threads \( b \) and \( c \) spawned by a parallel tuple are never referred to by thread handles (e.g. \( \text{tid}[b] \)) because these threads are not first class.

The expression \( \text{fg}(e) \) steps \( e \) until it reaches \( \text{fg}([\text{tid}[b]]) \), which then blocks until thread \( b \) has evaluated its expression down to an irreducible value \( e' \), at which point \( \text{fg}([\text{tid}[b]]) \) steps to \( e' \). The input rule is the only one which results in a delay, which is chosen nondeterministically from \( \Delta(d) \). After the delay, the new expression \( \text{in}(x,e) \) nondeterministically chooses a natural number \( n \) to substitute for \( x \) in \( e \), representing the uncertainty in the input from the user or environment.

We can prove type safety at the local level by showing that progress and preservation results hold for the local dynamics. Both lemmas have some unusual features. Local progress states that if an expression is well-typed, it is either fully evaluated or can take a step, or is waiting for some other thread using \( \text{join} \) or \( \text{fg} \) (which can take a step or is delayed).
and its expression be able to take a step (i.e. it cannot be waiting for other threads and cannot be an irreducible value).

A foreground thread is one that is currently executing a foreground block. The auxiliary definitions of Figure 11 determine which threads are currently running foreground blocks. For a thread pool \( \mu \), a thread \( a \) and its associated expression \( e \), \( RFB_a(e,a) \) 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 \( RFB_a(e,a) \) should only contain ready foreground blocks, the rule for \( e_1 \) or \( e_2 \) must split on whether \( e_1 \) or \( e_2 \) is currently evaluating.

The global step relation is \( r; \mu \Rightarrow g ; r'; \mu' \). We can now prove progress and preservation for the global semantics. Most of the work is done by Lemmas 6 and 7.

**Lemma 8** (Progress). If \( \cdot \vdash \mu : \Sigma \), then either \( \mu \) is final or there exist \( r' \) and \( \mu' \) such that \( r; \mu \Rightarrow g ; r'; \mu' \).

**Proof.** Let \( \mu = a_1 \leftarrow (\delta_1, e_1) \otimes \cdots \otimes a_m \leftarrow (\delta_m, e_m) \). 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 the configuration typing derivation, we have \( \Sigma = a_1 \sim \tau_1 @ w, \ldots, a_n \sim \tau_n @ w \) and for all \( i \), there exists \( \Sigma_i \) such that \( \cdot \vdash \Sigma_i e_i : \tau_i @ w \). By Lemma 6, either there exists some \( i \) such that \( e_i \mid \mu \Rightarrow^* (\delta'_i, e'_i) \mid \mu \otimes \mu' \) or for all \( i, e_i \) is final. In the former case, the configuration can take a step, and in the latter case, \( \mu \) is final.

**Lemma 9** (Preservation). If \( \cdot \vdash \mu : \Sigma \) and \( r; \mu \Rightarrow g ; r'; \mu' \), then there exists \( \Sigma' \) such that \( \cdot \vdash r'. \mu' : \Sigma' \).

**Proof.** Apply Lemma 7 to each local step, then use weakening and Lemma ?? 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-joinedness”. It is defined by the judgment \( e \nmid j \) (“\( e \) is well-joined”) in Figure 12. Intuitively, well-joinedness is the property that \( \text{join} \) expressions appear only in the part of an expression which is currently being evaluated.\(^2\) In particular, they may not appear encapsulated in functions, or in expressions which have not yet been evaluated. The auxiliary judgment \( e \nmid j \) (“no joins”) indicates that \( e \) contains no \( \text{join} \) expressions. Its straightforward definition is omitted for space reasons.

We first show that well-joinedness is preserved by local transitions: if all expressions of a thread pool are well-joined and one thread steps, then all resulting expressions are well-joined.

**Lemma 10.**

\[
e_0 \mid a_1 \leftarrow (\delta_1, e_1) \otimes \cdots \otimes a_n \leftarrow (\delta_n, e_n) \Rightarrow^*_a (\delta_0, e'_0) \mid a_1 \leftarrow (\delta_1, e'_1) \otimes \cdots \otimes a_n \leftarrow (\delta_n, e_n) \otimes \cdots \otimes a_m \leftarrow (\delta_m, e_m)
\]

and for all \( 0 \leq i \leq n \), we have \( e_i \nmid j \), then \( e'_0 \nmid j \) and for all \( n < i \leq m \), we have \( e_i \nmid j \).

**Proof.** By induction on the derivation of the transition judgment. See the appendix in the supplementary materials for details.

Finally, we prove a theorem which encompasses type safety and well-joinedness. If an initial thread pool consisting

\(^2\)For those familiar with evaluation contexts or stack machine semantics, \( \text{join} \) can only appear in the “hole” of an evaluation context or at the top of a stack.
**Theorem 3** (Type Safety and Well-Joinedness). If \( \cdot \vdash e : \tau \circ w \) and \( a \leftarrow (0, e) \Rightarrow_\sigma g \rightarrow r, \mu' \), then

1. there exists \( \Sigma' \) such that \( \cdot \vdash e : \mu : \Sigma' \)

2. either \( \mu' \) final or there exist \( r'' \) and \( \mu'' \) such that \( r, \mu'' \Rightarrow g \rightarrow r'' ; \mu'' \)

3. For all \( b \leftarrow (\delta, e_b) \in \mu' \), we have \( e_b \) wj.

**Proof.** Parts 1 and 2 are simply an inductive application of Lemmas 8 and 9. We prove part 3 by induction on the derivation of \( a \leftarrow (0, e) \Rightarrow_\sigma g \rightarrow r, \mu' \). If \( \mu' = a \leftarrow (0, e) \), then we must have \( e \) wj since \( e \) types with an empty signature, and this implies that \( e \) wj (these facts can be shown by a straightforward induction on the typing derivation and the derivation of \( e \) wj, respectively).

Otherwise, suppose \( 0 ; a \leftarrow (0, e) \Rightarrow_\sigma g \rightarrow r'' ; \mu'' \) and \( r'' ; \mu'' \Rightarrow_\sigma g \rightarrow r, \mu' \). By induction, \( e_b'' \) wj for all \( b \leftarrow (\delta'', e_b'') \in \mu'' \). Let \( b \leftarrow (\delta, e_b) \in \mu' \). We have three cases: (1) \( b \Leftarrow (\delta'', e_b'') \in \mu'' \) or (2) \( b \Leftarrow (0, e_b'') \in \mu'' \) and \( e_b'' \in \mu'' \) or (3) there exists \( c \leftarrow (0, e_c) \in \mu'' \) such that \( e_c | \mu'' \Rightarrow_\sigma c | b | e_b'') \in \mu'' \) and \( b \Leftarrow (\delta', e_b) \in \mu'' \). In case (1), the result is clear by induction. In cases (2) and (3), \( e_b'' \) wj by Lemma 10. \( \square \)

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 \( D W' / P + S' \). 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.

---

**Figure 10.** Global Dynamics.

**Figure 11.** Auxiliary definitions for the global dynamics.

**Figure 12.** Rules for well-joinedness of a single source expression (which is well-typed under the empty context and signature) evaluates to \( \mu' \) after some number of steps, then \( \mu' \) is well-typed, not stuck and all of its expressions are well-joined.
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 5, which shows that a local transition on a thread decreases the work and span of the thread’s dag by at least 1.

**Lemma 11.** Fix \( \Delta \) and suppose that \( \cdot \vdash e : \mu : \Sigma \) and that \( e \) \( \mathsf{w} \) \( j \) for all \( a \leftarrow (\delta, e) \in \mu \). If \( r ; \mu \Rightarrow \mathsf{g} \ r' ; \mu' \), 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. \( W^o(\mu', \Delta) \leq W^o(\mu, \Delta) \)
5. \( S^o(\mu', \Delta) \leq S^o(\mu, \Delta) \)
6. \( W(\mu, \Delta) - W^o(\mu', \Delta) \geq P \) or \( S^o(\mu', \Delta) - S^o(\mu, \Delta) \geq r' - r \) or \( r' = r \).

**Proof.** See the additional materials.

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

**Theorem 4.** Fix \( \Delta \) and let \( e \) be such that \( \cdot \vdash e : \tau @ \mathbb{B} \). Suppose \( e ; 0 \mathrel{\upharpoonright}_g \mathrel{\upharpoonright}_3 v ; g \) and let \( W = W(g) \) and \( S = S(g) \) and \( W^o = W^o(g) \) and \( S^o = S^o(g) \) and \( P = D(g) \). If \( 0 ; a \leftarrow (0, e) \Rightarrow^g \mathsf{g} r ; \mu \) and \( \mu \) final, then \( T \leq \frac{W}{P} + S \) and \( r \leq \frac{W^o}{P} + S^o \).

**Proof.** Let \( \mu_0 = a \leftarrow (0, e) \) and \( \mu_T = \mu \) and \( r_0 = 0 \) and \( r_T = r \).

We have a sequence \( 0 ; \mu_0 \Rightarrow^g r_1 ; \mu_1 \Rightarrow^g \ldots \Rightarrow^g r_T ; \mu_T \).

For each \( i \), let \( W_i = W(\mu_i, \Delta) \) (and similar for \( S_i, W^o_i, S^o_i \)). Note that \( W_0 = W \) (and similar for \( S, W^o, S^o \)) and that

\[
W_T = S_T = W^o_T = S^o_T = 0
\]

By Theorem 3, \( e_b \mathrel{\mathsf{w}} j \) for all \( b \leftarrow (\delta, e_b) \in \mu_i \). By Lemma 11,

\[
\frac{W_0}{P} + S_0 \geq 1 + \frac{W_1}{P} + S_1 \geq \cdots \geq 1 + \frac{W_T}{P} + S_T = 1
\]

This immediately gives \( \frac{W_0}{P} + S_0 \geq T \).

For each \( i \), consider the quantity \( D_i = \frac{W_i}{P} + S_i + r_i \). Note that for \( i = 0 \), \( D_0 = \frac{W_0}{P} + S_0 + r_1 = \frac{W}{P} + S^o \) and for \( i = T \), \( D_T = \frac{W_T}{P} + S_T^o + r_T = r \). When \( r_1; \mu_1 \Rightarrow^g r_{i+1}; \mu_{i+1} \), by Lemma 11, either

1. \( r_{i+1} = r_i \) and the other terms do not increase or
2. \( W^o_i - W^o_{i+1} \geq P \) and \( r_{i+1} - r_i = |DFB(\mu_i)| \leq D \) (the last inequality is by definition of \( D \)) or
3. \( S^o_i - S^o_{i+1} \geq |DFB(\mu_i)| \) and \( r_{i+1} - r_i = |DFB(\mu_i)| \)

In all three cases, the quantity above decreases or remains the same, so \( r \leq \frac{W^o}{P} + S^o \). \( \Box \)

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 \( f \) of the time taken to execute \( f \). Since \( S_f(f) \) is a lower bound on the time to execute \( f \), it is clear that \( S^o \), 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 4 is within a factor of two of optimal, it remains to show that \( D W^o/P \) 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 \( W^o \) which consists only of \( D \ll W^o \) foreground blocks, each of which is sequential. Think of the work of the computation as \( W^o \) “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 least \( \frac{W^o}{P} - 2P \) steps (the number of steps it will take to run out of bricks), which will cause the response time to be at least \( D \frac{W^o - 2D}{P} \in O(D \frac{W^o}{P}) \).

6. Implementation and Examples

6.1 Algorithm and Implementation

So far, we have bounded the length and response time of a prompt schedule (Section 2), designed a language which can take advantage of these bounds for responsive parallel programs (Section 3) and shown that the bounds can be approximated by a lower-level (closer to the machine) operational semantics (Section 4). Like Brent’s original result, none of this specifies a concrete scheduling algorithm or accounts for the overhead of running a scheduler that must make schedul-
ing decisions in an online fashion. Many algorithms, such as work stealing, have been developed which asymptotically approximate the greedy scheduling bound, and we will now discuss extensions to such algorithms which meet the prompt scheduling bound.

The main difference between algorithms for prompt scheduling and prior work is preemption. In algorithms that approximate greedy schedules, it is not necessary to preempt a processor once it is working on a task. In the prompt setting, however, preemption seems necessary for guaranteeing responsiveness because otherwise lower priority threads can starve higher-priority ones for processors.

Probably the simplest algorithm that could meet the prompt scheduling bound would be one based on a global priority queue. At periodic intervals, all processors are interrupted and a central scheduler will assign them ready threads from a global queue, prioritizing foreground tasks. Like a central-queue algorithm for non-prioritized scheduling, this would be simple to implement and analyze but would incur high overhead because of the global queue.

Instead, we propose an algorithm based on a low-contention variant of stealing [2] which uses message passing to cooperatively send tasks between processors. In our algorithm, each processor has two private double-ended queues (deques), one for foreground work and one for background. Each processor also has a public communication cell to which other processors can send threads. Each time the scheduler is invoked, each processor attempts to send ("deal") a thread to another processor, dealing foreground threads whenever one is available, by atomically writing a pointer to the thread into the target processor’s communication cell. Each processor then checks its own queue and communication cell and begins working on the highest-priority task available. All processors are interrupted at a fixed granularity (between 10 and 100ms) to return to the scheduler in case a higher-priority task has become available through a deal.

Formally showing that this algorithm approximates the prompt scheduling bound is beyond the scope of this paper. Instead, we demonstrate that such an algorithm is plausible in practice by demonstrating a prototype implementation. Our implementation of the algorithm is a parallel extension of Standard ML [43, 44] and uses fork-join parallelism and futures, both built into the parallel ML extension, to create threads. The implementation also consists of a simple prioritized scheduler. We did not extend SML’s type system to implement λp’s type system.

6.2 Benchmarks

In order to evaluate our implementation, we have developed a number of parallel interactive benchmarks. To bring the expertise of prior work into this new area of parallel interaction, we have drawn from established benchmarks for parallel computation and added a number of different interactive elements. The benchmarks serve two purposes: first, to show that the overhead and scalability of our scheduler is empirically acceptable, and second, to show that the techniques described in this paper are expressive enough, and our implementation scalable enough, to develop real applications. To this end, we divide our benchmark suite into two groups:

- Four synthetic “orthogonal benchmarks” in which the parallel and interactive components are more or less disjoint, allowing us to isolate these two elements and tune them separately while observing the effect that each element has on the performance of the other.
- Two “integrated benchmarks” in which the parallel computation and interaction are integrated together. These benchmarks are slightly more representative of real applications, but still synthetic enough to perform detailed quantitative analysis.
- Three “application benchmarks”, which are more realistic applications.

6.2.1 Orthogonal benchmarks

The purpose of the orthogonal benchmarks is to combine a known parallel computation benchmark with a relatively well-behaved interactive element in such a way that the two computations do not impact each other’s execution except through the scheduler. This allows us to design precise experiments in which we vary one parameter of either the interaction or the computation (frequency of interaction, size of computation data set, etc.) to determine how this impacts the scheduler without affecting the other type of task.

In order to develop a set of benchmarks that was representative of the space of parallel interactive programs, we divided parallel algorithms and interactive elements into two classes: “regular” and “irregular.” Regular parallel algorithms have a uniform and balanced structure or may even be entirely static with the division of the job into parallel tasks done ahead of time via static partitioning techniques. Irregular parallel algorithms do not display such structure; tasks are generated dynamically in large numbers, and in an unbalanced and unpredictable fashion. The division of interactive elements is similar: regular interaction involves interaction with a constant number of users or agents at predictable intervals while in irregular interaction, users or agents may come and go at runtime or widely vary their frequency of interaction.

We selected two established parallel computation benchmarks and two interactive elements, one each from the “regular” and “irregular” classes. The parallel computation benchmarks are Fibonacci (regular) and Unbalanced Tree Search (UTS, irregular) [37]:

- **Fibonacci** The benchmark computes the 43rd Fibonacci number by a naïve recursive (exponential-time) algorithm. This computation, while deliberately work-inefficient, is embarrassingly and uniformly parallel.
- **UTS** The benchmark counts the number of nodes in a tree of depth 10, where the number of children of each node is determined randomly according to a geometric distribution.

```plaintext
The benchmark counts the number of nodes in a tree of depth 10, where the number of children of each node is determined randomly according to a geometric distribution.
```
with an expected value of 5.0. This benchmark is adversarial to load balancing because, at any node, the size of each subtree is unknown and may vary widely.

The interactive elements are Console Echo and Network Echo:

**Console Echo** The benchmark reads a name from standard input, immediately responds with “Hi, name!” and repeats. This is the only input task at any time.

**Network Echo** The benchmark opens a socket and listens for incoming connections. When a connection is received, it spawns a new thread (implemented as a future) for this connection. The interaction between the program and each connected client proceeds as in Console Echo above, until the program terminates or the client disconnects. In the same way that UTS is adversarial to computation load balancing, this is adversarial to handling of interaction since the number of clients that will connect is unknown and many clients may be active at a time.

We combine each pair of computation benchmark and interactive element to form four benchmarks, fib_term, fib_net, uts_term and uts_net, which cover each of the four combinations of regular and irregular parallelism and interaction. In each case, the program forks at the top level to spawn two parallel tasks, a background one which runs the computation benchmark (which itself forks off parallel tasks) and a foreground one for the interaction.

### 6.2.2 Integrated benchmarks

**Fibonacci “server”** In this benchmark, an interactive loop waits for the user to enter a number on the console. When a number \( n \) is entered, the interactive prompt returns immediately (in the foreground) while the \( n^{th} \) Fibonacci number is computed asynchronously in the background. The result is printed when it is completed.

**Interactive convex hull** In a somewhat more realistic variant of the Fibonacci server, we replace the Fibonacci computation with a parallel convex hull computation.\(^4\) A graphical interface allows the user to input new points by clicking on the screen. New points are displayed on the screen immediately (in the foreground). Asynchronously in the background, the convex hull of all points is computed. The lines of the hull are drawn on the screen when the hull is completed. Convex hull itself may not be of great interest as a benchmark. However, since convex hull is related to Delaunay triangulation, one could easily imagine scaling this benchmark up to a computer-aided design application in which users enter points in 3D and the mesh of the points is computed dynamically.

### 6.2.3 Application benchmarks

**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 \( bg(fg(. . .)) \)) 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 the number of visits to each page). The analytics can be viewed by requesting `stats.html` from the server.

**Photo viewer** Our photo viewer benchmarks allows users to navigate through a folder of JPEG images. The user can scroll through images using the arrow keys and can also jump to an image out of order. As usual, the user interaction is performed in the foreground. Images which have already been decoded are cached, so they can be displayed to the screen immediately. In addition, a background computation decodes the next several images (one image per processor can be decoded; decoding itself is sequential), so that images which are viewed in order (the common case) can be displayed more quickly. If the user selects an image which has not yet been decoded, it is decoded in the foreground.

**Music server** A streaming music server listens for clients to connect on a socket. Each time a client connects, a new thread is spawned to handle the connection. The client requests a music file from the server, and the server opens the file and sends its bytes over the connection until the end of the file is reached. Some clients (perhaps those paying for a higher level of service) are designated high-priority, and these are handled by foreground threads. Remaining clients are designated low-priority. If there are few enough threads, these clients will still be handled properly, but if the server becomes overloaded, high-priority clients will be handled first. The loop that accepts connections is itself run in the foreground but is usually idle.

7. Related Work

Abstractions and cost models for parallel programming have been studied extensively and many programming languages and extensions have been created [16, 21, 29, 31, 33, 34]. 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

\(^4\) We use a parallel implementation of Quickhull developed by Daniel Spoonhower in his PhD thesis.
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 $\lambda^p$ is based on that of Davies [17] 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 [32, 35, 46?]. 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 [17], similar to that of [32, 35, 46?], 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 $\lambda^p$, we don’t order evaluation according to the stages—we allow them to occur concurrently. We know that a stage 1 ($\mathbb{P}$) computation cannot possibly inspect a stage 2 ($\mathbb{B}$) 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 $\lambda^p$ 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 [38, 39]. Cost semantics have proved to be particularly important in lazy languages (e.g., [39, 40]) and parallel languages (e.g., [7, 8, 44]). Our approach builds directly on the work of Blelloch and Greiner [8] and Spoonhower et al. [44], who use computation graphs represented as daggs (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 [14]. 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 [48], Brent [14], and Eager et al. [18] 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, 11, 12, 15, 19, 21, 23, 25, 36, 47]. More recent papers showed that priority-based schedulers can improve performance in practice [28, 49, 50], but offer no bounds. All of this work, however, considers non-interactive, compute-intensive applications. [?] 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 [41]. There has been significant recent interest in making operating systems work well on multicore machines [5, 13]. 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 [32, 35, 46?]. 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 responsive-
ness developed in this paper will hopefully prove useful in such further studies of responsive parallelism.

References


