Correctness of Dynamic Dependence Analysis for Implicitly Parallel Tasking Systems

Wonchan Lee
Stanford University
wonchan@cs.stanford.edu

George Stelle
Los Alamos National Laboratory
stelleg@lanl.gov

Patrick McCormick
Los Alamos National Laboratory
pat@lanl.gov

Alex Aiken
Stanford University
aiken@cs.stanford.edu

***Abstract*—In this paper, we rigorously verify the correctness of dynamic dependence analysis, a key algorithm for parallelizing programs in implicitly parallel tasking systems. A dynamic dependence analysis of a program results in a *task graph*, a DAG of tasks constraining the order of task execution. Because a program is automatically parallelized based on its task graph, the analysis algorithm must generate a graph with all the dependencies that are necessary to preserve the program's original semantics for any non-deterministic parallel execution of tasks. However, this correctness is not straightforward to verify as implicitly parallel tasking systems often use an optimized dependence analysis algorithm. To study the correctness of dynamic dependence analysis in a realistic setting, we design a model algorithm that captures the essence of realistic analysis algorithms. We prove that this algorithm constructs task graphs that soundly and completely express correct parallel executions of programs. We also show that the generated task graph is the most succinct one for a program when the program satisfies certain conditions.**

I. Introduction

A class of implicitly parallel tasking systems [1]–[5] have been proposed for ease of parallel programming. In these programming models, programs are implicitly parallel: a program is decomposed into tasks that are automatically parallelized at runtime. By construction, these task-based programs are free of common parallel programming errors, such as data races and deadlocks, because the runtime system inserts necessary synchronization and data movement on behalf of programmers. Compared to explicitly parallel programming models that require programmers to manage parallelism, synchronization, and communication manually, implicitly parallel tasking systems simplify parallel programming.

Dynamic dependence analysis is one of the key algorithms in these implicit systems. A dynamic dependence analysis algorithm takes a stream of tasks as input, analyzes dependencies between tasks, and constructs a *task graph*, a DAG of tasks where edges denote task dependencies constraining the execution order of tasks. The runtime system then finds the tasks that are mutually unreachable in this graph and (potentially) runs them in parallel. The analysis is performed dynamically because programs often have task dependencies that can be determined only at runtime. A dynamic dependence analysis must find all the task dependencies that are necessary for preserving the *sequential semantics* of the program; any parallel execution of a program must yield the same result as the sequential execution.

In this paper, we report our initial efforts on formally verifying the correctness of dynamic dependence analysis. Because in task-based programming models programmers depend on the runtime system's dynamic dependence analysis to parallelize programs, any bugs in the algorithm can be extremely difficult to understand, let alone fix. Therefore, the correctness of dynamic dependence analysis is essential for the usability of implicitly parallel tasking systems. Although straightforward implementations are trivially correct, implicitly parallel tasking systems often use an optimized algorithm for their dynamic dependence analysis, whose correctness is far from obvious. To verify the correctness of realistic dynamic dependence analysis algorithms, we design a model dependence analysis algorithm $\mathcal{D}_{\mathsf{epoch}}$ that captures the essence of Legion's dynamic dependence analysis [6], and prove that it generates *sound* task graphs, i.e., task graphs that describe only the correct parallel executions of tasks.

The $\mathcal{D}_{\mathsf{epoch}}$ algorithm tries to minimize *transitive* dependencies, i.e., dependencies that are transitively expressed by other dependencies, because finding transitive dependencies is redundant and unnecessary. The $\mathcal{D}_{\mathsf{epoch}}$ algorithm achieves this goal by reducing the number of tasks that must be analyzed by only considering *epochs* of tasks at the *frontier* of a dynamically constructed task graph. This optimization is based on the observation that any dependencies between a task and those not at the frontier of the task graph are transitive. This optimization avoids a quadratic enumeration of all possible task dependencies, most of which are likely transitive. On the other hand, this optimization complicates the soundness proof of the $\mathcal{D}_{\mathsf{epoch}}$ algorithm, as the proof needs to show that in every step the algorithm maintains the invariant that epochs contain all the tasks at the frontier.

In addition to soundness, we discuss two optimality criteria for dynamic dependence analysis: *completeness* and *parsimony*. The soundness of a task graph means that it describes only the correct executions of a program, but it does not assert how much parallelism it exposes; a linear task graph that forces tasks to run sequentially is sound but rarely useful, unless the program has no parallelism at all. Therefore, it is desirable that a task graph be *complete*; i.e., it should capture all the correct *parallel* executions of tasks. We prove that the $\mathcal{D}_{\mathsf{epoch}}$ algorithm generates task graphs that are not only sound but also complete.

Among task graphs that express the same set of parallel

```
1 task F(x) reads(x),writes(x)
2 task G(x) reads(x)
3 while t < T do
4   for i = 0,2 do F(A[i]) end
5   if t % k == 0 then
6     for i = 0,2 do G(A[g(i)]) end
7   end
8 end
```

Fig. 1: Example Program

executions, one with fewest edges is most succinct, or *parsimonious*. Although a parsimonious task graph is most economical in terms of space, directly constructing parsimonious task graphs is often avoided as it requires an expensive reduction of transitive edges. We show that when a program satisfies a certain condition, the $\mathcal{D}_{\mathsf{epoch}}$ algorithm generates a parsimonious task graph without a costly transitive reduction.

This paper makes two contributions:

- We present a formal framework for proving the correctness of dynamic dependence analysis algorithms. To the best of our knowledge, our paper is the first to present a formal treatment of dynamic dependence analysis.
- We design a realistic, epoch-based dependence analysis algorithm $\mathcal{D}_{\mathsf{epoch}}$ and prove its three key properties: soundness, completeness, and parsimony.

The rest of this paper describes the task-based programming model that is used throughout the paper in Section II, presents the design of the model dependence analysis algorithm ($\mathcal{D}_{\mathsf{epoch}}$) and proves its properties in Section III, summarizes related work in Section IV, and concludes in Section V.

II. Task-based Programming Model

A. Programs

Consider the example program in Figure 1. This program launches two tasks $\mathtt{F}(\mathtt{A}[\mathtt{0}])$ and $\mathtt{F}(\mathtt{A}[\mathtt{1}])$ in each iteration, and two other tasks $\mathtt{G}(\mathtt{A}[\mathtt{g}(\mathtt{0})])$, and $\mathtt{G}(\mathtt{A}[\mathtt{g}(\mathtt{1})])$ every $\mathtt{k}$ iterations.

Any attempt to statically identify parallel tasks in this program is hampered in two ways. First, tasks $\mathtt{G}(\mathtt{A}[\mathtt{g}(\mathtt{0})])$ and $\mathtt{G}(\mathtt{A}[\mathtt{g}(\mathtt{1})])$ use an opaque function $\mathtt{g}$ whose values are unknown statically. Second, whether the `if` branch (lines 5-7) is taken in each iteration cannot be precisely tracked, because the branch condition is resolved only at runtime.

Implicitly parallel tasking systems do not have this limitation, because they dynamically analyze a stream of tasks that a program generates, rather than analyzing the program itself. Assuming $\mathtt{g}(\mathtt{0}) = \mathtt{g}(\mathtt{1}) = \mathtt{0}$, the example program generates a task stream of the following pattern:

$$\Big(\big(\mathtt{F}(\mathtt{A}[\mathtt{0}]);\mathtt{F}(\mathtt{A}[\mathtt{1}]);\big)^{\mathtt{k}}\mathtt{G}(\mathtt{A}[\mathtt{0}]);\mathtt{G}(\mathtt{A}[\mathtt{0}]);\Big)^{*}.$$

Because all the function values and the control divergence are resolved in this stream, dependencies between the tasks can be precisely analyzed.

To model this aspect of implicitly parallel tasking systems, we abstract a program as the sequence of tasks it generates:

$$\text{Programs}\quad p \in \mathit{Program} = \mathit{Task}^{*} \qquad p ::= \epsilon \mid t; p$$

B. Tasks and Task Dependencies

Dependencies between tasks are determined by how tasks access data. Tasks can access data stored in *regions* according to the *privileges* requested by the task; tasks can read or write data only when they have the corresponding read (rd) or write (wr) privilege. In the program in Figure 1, every task $\mathtt{F}(\mathtt{A}[\mathtt{i}])$ has read and write privileges on the input region $\mathtt{A}[\mathtt{i}]$, and every task $\mathtt{G}(\mathtt{A}[\mathtt{i}])$ has only read privilege on $\mathtt{A}[\mathtt{i}]$.

We define a task as a pair of a unique identifier and a set of pairs of regions and privileges. (In the following, we use $\wp(X)$ for the powerset of X.)

$$\begin{array}{lll} \text{Tasks} & t \in \mathit{Task} & = \mathit{Id} \times \wp(\mathit{Region} \times \mathit{Privilege}) \\ \text{Regions} & r \in \mathit{Region} & \\ \text{Privileges} & pv \in \mathit{Privilege} & = \{\mathsf{rd}, \mathsf{wr}\} \end{array}$$

For simplicity, we assume that each region is unique. We write $t(pv)$ to denote a set of regions in task t whose privilege is pv, i.e., $t(pv) \triangleq \{r \mid (r, pv) \in t\}$, and $\mathsf{rgn}(t) = t(\mathsf{rd}) \cup t(\mathsf{wr})$.

Tasks are *dependent* when they access the same region and at least one can write to it.

Definition 1. *We write $t_1 \Leftrightarrow t_2$ when tasks t_1 and t_2 are dependent on each other, i.e.,*

$$\exists r, pv_1, pv_2.(r, pv_1) \in t_1 \wedge (r, pv_2) \in t_2 \wedge (pv_1 = \mathsf{wr} \vee pv_2 = \mathsf{wr}).$$

We write $t_1 \star t_2$ when tasks t_1 and t_2 are independent of each other, i.e., $t_1 \star t_2 = \neg(t_1 \Leftrightarrow t_2)$.

In the example program in Figure 1, each $\mathtt{F}(\mathtt{A}[\mathtt{0}])$ task at iteration $\mathtt{t}$ is dependent on $\mathtt{F}(\mathtt{A}[\mathtt{0}])$ at iteration $\mathtt{t} - \mathtt{1}$ and is also dependent on two $\mathtt{G}(\mathtt{A}[\mathtt{0}])$ tasks if $\mathtt{t}$ is a multiple of $\mathtt{k}$.

The dependence in Definition 1 is *syntactic*; the definition only requires interference between tasks' privileges on the same region. Implicitly parallel tasking systems check this syntactic dependence between tasks to avoid performing analysis of task bodies [7]. For this syntactic check to be sound, implicitly parallel tasking systems require that each task's region privileges must match the behavior of that task. This condition can be checked by a suitable type system [7].

Assumption 1. *We assume that every task t and its semantics $[\![t]\!]$ satisfy the following properties:*

- $\forall r.[\![t]\!](h)(r) \neq h(r) \Longrightarrow r \in t(\mathsf{wr})$ *and*
- $\forall h, h'.\ (\forall r \in t(\mathsf{rd}).h(r) = h'(r)) \Longrightarrow \forall r \in t(\mathsf{wr}).[\![t]\!](h)(r) = [\![t]\!](h')(r),$

where the semantics $[\![t]\!] : \mathit{Heap} \to \mathit{Heap}$ of task t is an opaque function on heap states and a heap state h is a map from regions to data:

$$h \in \mathit{Heap} = \mathit{Region} \to \mathit{Data}.$$

Note that the semantics $[\![t]\!]$ is not well-defined when task t is non-terminating; we consider only terminating tasks to simplify reasoning.

An important consequence of Assumption 1 is that independent tasks can execute in parallel.

Lemma 1. *Let t_1 and t_2 be independent tasks (i.e., $t_1 \star t_2$). Then, their semantics satisfy the following property:*

$$(\llbracket t_1 \rrbracket \circ \llbracket t_2 \rrbracket)(h) = h \lhd \llbracket t_1 \rrbracket(h)|_{t_1(\mathsf{wr})} \lhd \llbracket t_2 \rrbracket(h)|_{t_2(\mathsf{wr})},$$

where $f|_X$ is the restriction of f to X and

$$(h_1 \lhd h_2)(r) \triangleq \begin{cases} h_2(r) & \text{when } r \in dom(h_2) \\ h_1(r) & \text{otherwise.} \end{cases}$$

Proof. Follows directly from Assumption 1 of task semantics. □

Note that semantics functions in this lemma do not use the result of one another, allowing them to be evaluated simultaneously.

C. Schedules

The primary goal of implicitly parallel tasking systems is to guarantee that a program has a *sequential semantics*; any parallel execution of the tasks in a program must yield the same result as the sequential execution. Assuming each task is running *atomically*, i.e., running to completion once scheduled on a processor, a parallel execution of tasks can be represented by a set of linearized executions. For example, a parallel execution of three tasks t_1, t_2, and t_3 can be represented by the following six linearized executions:

$$\begin{array}{lll} t_1;t_2;t_3, & t_1;t_3;t_2, & t_2;t_1;t_3, \\ t_2;t_3;t_1, & t_3;t_1;t_2, \text{ and} & t_3;t_2;t_1. \end{array}$$

Therefore, proving that an implicitly parallel tasking system guarantees the sequential semantics for a program amounts to showing that the system admits only the linearized task executions that produce the same result as the sequential execution.

To describe a linearized execution of tasks, we use a *schedule*, which is a sequence of all the tasks in the program.

$$\text{Schedules} \quad sc \in Schedule = Task^* \qquad sc ::= \epsilon \mid t; sc$$

The sequential execution of a program is the program itself.

Among all possible schedules of a program, implicitly parallel tasking systems allow only the *valid* schedules; a schedule is valid iff it respects program order of all the dependent tasks in a program.

Definition 2. *A schedule sc of program p is valid iff*

$$p \vdash t_1 \succ^+ t_2 \wedge t_1 \Leftrightarrow t_2 \implies sc \vdash t_1 \succ^+ t_2,$$

where $s \vdash a \succ^+ b$ denotes that b occurs somewhere after a in s. We write $p \vdash sc$ when sc is a valid schedule of p.

Valid schedules are important because they produce the same result as the sequential schedule.

Theorem 2. *A valid schedule sc of program p satisfies*

$$\llbracket sc \rrbracket = \llbracket p \rrbracket,$$

where $\llbracket sc \rrbracket$ denotes the semantics of a schedule, which is a composition of individual tasks' semantics, i.e.,

$$\llbracket t_1; t_2; \ldots; t_n \rrbracket \triangleq \llbracket t_n \rrbracket \circ \cdots \circ \llbracket t_2 \rrbracket \circ \llbracket t_1 \rrbracket.$$

Proof. Follows from Lemma 3 and the induction on the program size. □

Key to the proof of this semantics preservation is the fact that order of running independent tasks makes no change to the semantics, formally stated as the following *commutativity* lemma.

Lemma 3. $t_1 \star t_2 \implies \llbracket t_1 \rrbracket \circ \llbracket t_2 \rrbracket = \llbracket t_2 \rrbracket \circ \llbracket t_1 \rrbracket$

Proof. Follows directly from Lemma 1 and commutativity of the $(\lhd)$ operator when domains of operands are disjoint. □

Note that invalid schedules sometimes preserve sequential semantics as well when there are commutative tasks, i.e., tasks t_1 and t_2 such that $t_1 \Leftrightarrow t_2 \wedge \llbracket t_1 \rrbracket \circ \llbracket t_2 \rrbracket = \llbracket t_2 \rrbracket \circ \llbracket t_1 \rrbracket$. Unlike independent tasks, commutative tasks cannot run in parallel, but they bring flexibility to scheduling as they can execute in any order as long as they run atomically. Though some implicitly parallel tasking systems exploit commutativity of tasks in scheduling [1], [4], we do not consider commutative tasks in our formalism, as they do not change the essence of dynamic dependence analysis.

III. Dynamic Dependence Analysis

A. Task Graphs

For the purpose of discovering valid schedules, dynamic dependence analysis algorithms construct a *task graph*, which concisely represents a set of schedules. A correct dependence analysis of a program must yield a task graph that expresses only the valid schedules of that program.

We define a task graph as a directed acyclic graph of the tasks in a program, where each edge between nodes constrains the order in which they can be scheduled. Any topological ordering of nodes in this graph represents a schedule.

$$\begin{array}{ll} \text{Task Graphs } G = \langle T, D \rangle \in & TaskGraph \\ & = \wp(Task) \times \wp(Task \times Task) \end{array}$$

Definition 3. *A schedule sc is a topological ordering of the task graph $\langle T, D \rangle$ if it satisfies the following property:*

$$\forall t_1, t_2 \in T. \langle t_1, t_2 \rangle \in D \implies sc \vdash t_1 \succ^+ t_2.$$

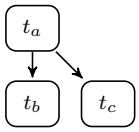
We write $\llbracket G \rrbracket$ for a set of all the schedules expressed by G.

Note that only the nodes that are mutually unreachable in a task graph represent tasks that can execute in parallel; if there is a path between tasks in this graph, their execution must be ordered.

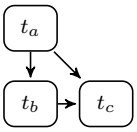
A *sound* task graph for a program p is a task graph whose schedules are all valid for p and a *complete* task graph for a program p is the one that expresses all the valid schedules of p.

Definition 4 (Soundness)**.** *We say a task graph G is sound for a program p if $\forall sc \in \llbracket G \rrbracket. p \vdash sc$.*

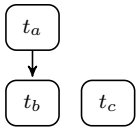
Definition 5 (Completeness)**.** *We say a task graph G is complete for a program p if $\forall sc. p \vdash sc \implies sc \in \llbracket G \rrbracket$.*



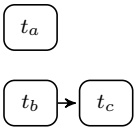
(a) Sound and complete G_1



(b) Sound but incomplete G_2



(c) Complete but unsound G_3



(d) Unsound and incomplete G_4

Fig. 2: Task graphs for program p_1 in Example 1

The soundness of a task graph is essential for correctness because an unsound task graph can express the schedules that break sequential semantics. Completeness, on the other hand, is an optimality criterion; if a task graph is incomplete, it might serialize some tasks that could otherwise run in parallel.

Example 1. *Suppose we have a program* $p_1 = t_a; t_b; t_c;$*, where*

$$t_a = \langle 1, \{\langle A, \mathsf{wr}\rangle\}\rangle, t_b = \langle 2, \{\langle A, \mathsf{rd}\rangle\}\rangle, \text{ and}$$
$$t_c = \langle 3, \{\langle A, \mathsf{rd}\rangle\}\rangle$$

The set S_{valid} *of valid schedules for this program has two schedules* $t_a; t_b; t_c$ *and* $t_a; t_c; t_b$*.*

The following table summarizes the soundness and the completeness of each task graph in Figure 2.

Graph	*Sound?*	*Complete?*	*Reason*
G_1	*Yes*	*Yes*	$[\![G_1]\!] = S_{\mathsf{valid}}$
G_2	*Yes*	*No*	$[\![G_2]\!] = \{t_a; t_b; t_c\} \subset S_{\mathsf{valid}}$
G_3	*No*	*Yes*	$[\![G_3]\!] = S_{\mathsf{valid}} \cup \{t_c; t_a; t_b\} \supset S_{\mathsf{valid}}$
G_4	*No*	*No*	$t_a; t_c; t_b \in S_{\mathsf{valid}} - [\![G_4]\!]$
			$t_b; t_c; t_a \in [\![G_4]\!] - S_{\mathsf{valid}}$

Another optimality criterion that we consider for task graphs is *parsimony*; since multiple task graphs can express the same set of schedules, we consider the one with the least number of edges as parsimonious.

Definition 6 (Parsimony)**.** *We say a task graph* $G = \langle T, D\rangle$ *is parsimonious if there exists no other task graph* $G' = \langle T', D'\rangle$ *such that*

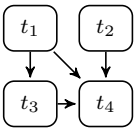
$$[\![G]\!] = [\![G']\!] \wedge \exists t_1, t_2. \langle t_1, t_2\rangle \in D \wedge \langle t_1, t_2\rangle \notin D'.$$

Parsimonious task graphs have no *transitive* edges between nodes connected by some other path. Although the parsimonious task graph is most succinct, making task graphs parsimonious is often avoided as it requires a costly transitive reduction for the whole graph.

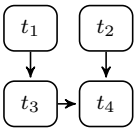
Example 2. *Figure 3 shows task graphs* G_1 *and* G_2 *that are sound and complete for program* $p_2 = t_1; t_2; t_3; t_4$*, where*

$$t_1 = \langle 1, \{\langle A, \mathsf{wr}\rangle\}\rangle, \qquad t_2 = \langle 2, \{\langle B, \mathsf{wr}\rangle\}\rangle,$$
$$t_3 = \langle 3, \{\langle A, \mathsf{wr}\rangle\}\rangle, \text{ and } t_4 = \langle 4, \{\langle A, \mathsf{rd}\rangle, \langle B, \mathsf{rd}\rangle\}\rangle.$$

Though G_1 *and* G_2 *express the same set of schedules, only* G_2 *is parsimonious, because* G_1 *has a transitive edge between* t_1 *and* t_4 *and removing any edge from* G_2 *yields an unsound task graph expressing a different set of schedules.*



(a) Non-parsimonious G_1



(b) Parsimonious G_2

Fig. 3: Task graphs for program p_2 in Example 2

B. Epoch-Based Dependence Analysis Algorithm $\mathcal{D}_{\mathsf{epoch}}$

The simplest way to create a sound and complete task graph is to enumerate all pairs of tasks and choose those between dependent tasks, as in the following algorithm $\mathcal{D}_{\mathsf{simple}}$.

Definition 7. $\mathcal{D}_{\mathsf{simple}}(p) \triangleq \langle p, E\rangle$*, where*

$$E = \{\langle t_1, t_2\rangle \mid p \vdash t_1 \succ^+ t_2 \wedge t_1 \Leftrightarrow t_2\}.$$

Lemma 4. $\mathcal{D}_{\mathsf{simple}}(p)$ *is sound and complete for program* p*.*

Proof. Follows directly from the definitions of soundness and completeness. □

However, the $\mathcal{D}_{\mathsf{simple}}$ algorithm is not useful, because it performs many redundant and unnecessary checks to find dependencies that are *transitively* expressed by other dependencies. The resulting graph is almost always not parsimonious because of edges representing those transitive dependencies. Furthermore, the algorithm requires the entire sequence of tasks to construct a task graph, although in reality this sequence can be arbitrarily long as one task stream corresponds to the whole execution of a program.

Example 3. *Running* $\mathcal{D}_{\mathsf{simple}}$ *on program* p_2 *in Example 2 yields the non-parsimonious task graph* G_1 *in Figure 3 (*$\mathcal{D}_{\mathsf{simple}}(p_2) = G_1$*).*

More realistic algorithms find as few transitive dependencies between tasks as possible and use data structures summarizing the task history to avoid querying the entire task stream. One notable example along this line is Legion's epoch-based dependence analysis algorithm [6]. The key idea behind this algorithm is to use the Single-Writer, Multiple-Reader (SWMR) invariant [8] to suppress transitive dependencies. Specifically, the SWMR invariant states that in any given epoch of tasks for a region, there can be only one writer task or multiple reader tasks, and that there are only two possible cases when a new task arrives:

- **C1** If both the current epoch and the new task are readers of the region, then the new task joins the current epoch.
- **C2** If either the current epoch or the new task is a writer of the region, then the new task starts a new epoch and the current epoch becomes the previous epoch.

In both cases, the new task catches dependencies on all the tasks in the previous epoch and any dependencies between this new task and those that are not in the previous epoch are transitive dependencies. Note that tasks still can have transitive

$$\boxed{(E, P, G, p) \xrightarrow{\mathsf{dep}} (E, P, G, p)}$$

$$\frac{\mathsf{rgn}(t) \neq \varnothing \quad R_r = t(\mathsf{rd}) \quad R_w = t(\mathsf{wr}) \quad E, P, G \vdash_{\mathsf{dep}} (t, R_r, \mathsf{rd}) \rightsquigarrow E_1, P_1, G_1 \quad E_1, P_1, G_1 \vdash_{\mathsf{dep}} (t, R_w, \mathsf{wr}) \rightsquigarrow E_2, P_2, G_2}{(E, P, G, t; p) \xrightarrow{\mathsf{dep}} (E_2, P_2, G_2, p)}$$

$$\frac{\mathsf{rgn}(t) = \varnothing \quad G' = \mathsf{insert}(G, \varnothing, t)}{(E, P, G, t; p) \xrightarrow{\mathsf{dep}} (E, P, G', p)}$$

$$\boxed{E, P, G \vdash_{\mathsf{dep}} (t, R, pv) \rightsquigarrow E, P, G}$$

$$\frac{}{E, P, G \vdash_{\mathsf{dep}} (t, \varnothing, pv) \rightsquigarrow E, P, G}$$

$$\frac{r \in R \quad R' = R - \{r\} \quad E, P, G \vdash_{\mathsf{dep}} (t, r, pv) \rightsquigarrow E_1, P_1, G_1 \quad E_1, P_1, G_1 \vdash_{\mathsf{dep}} (t, R', pv) \rightsquigarrow E_2, P_2, G_2}{E, P, G \vdash_{\mathsf{dep}} (t, R, pv) \rightsquigarrow E_2, P_2, G_2}$$

$$\boxed{E, P, G \vdash_{\mathsf{dep}} (t, r, pv) \rightsquigarrow E, P, G}$$

$$\frac{E' = E[r \mapsto \mathsf{swmr}(E(r), P(r), t, pv)] \quad P' = P[r \mapsto pv] \quad G' = \mathsf{insert}(G, E'(r).2, t)}{E, P, G \vdash_{\mathsf{dep}} (t, r, pv) \rightsquigarrow E', P', G'}$$

Fig. 4: Epoch-based dependence analysis algorithm $\mathcal{D}_{\mathsf{epoch}}$

dependencies when they use multiple regions, because the epochs are managed independently for each region. In the rest of this section, we prove that this epoch-based dependence analysis scheme is actually correct.

Figure 4 shows an idealized epoch-based dependence analysis algorithm $\mathcal{D}_{\mathsf{epoch}}$. The relation $\xrightarrow{\mathsf{dep}}$ describes one step of the $\mathcal{D}_{\mathsf{epoch}}$ algorithm for the first task in a program. For each task, the algorithm first analyzes the regions with read privilege and then those with write privilege.

The $\mathcal{D}_{\mathsf{epoch}}$ algorithm maintains the SWMR invariant in two data structures. The epoch privilege P records whether the current epoch of an region is of a writer (wr) or of readers (rd). The epoch map E maps a region to its current and previous epochs.

Epoch Maps	$E : \mathit{Region} \to \wp(\mathit{Task}) \times \wp(\mathit{Task})$
	$E ::= E_\varnothing \mid E[r \mapsto \langle T, T \rangle]$
Epoch Privileges	$P : \mathit{Region} \to \mathit{Privilege}$
	$P ::= P_\varnothing \mid P[r \mapsto pv]$

An empty epoch map $E_\varnothing$ maps every region to a pair of empty epochs and an empty epoch privilege $P_\varnothing$ returns wr for all regions.

The epoch map and epoch privilege must satisfy two conditions for the $\mathcal{D}_{\mathsf{epoch}}$ algorithm to produce a sound task graph. First, they must satisfy the SWMR invariant. (In the following definition, we use these notations: $\langle A, B \rangle \cdot 1 \triangleq A$ and $\langle A, B \rangle \cdot 2 \triangleq B$)

Definition 8. *Epoch map E and epoch privilege P are valid iff they satisfy the following conditions of the SWMR invariant.*

1) *Tasks in the previous or current epoch for a region have privilege for that region.*

$$\forall r. \forall t \in E(r) \cdot 1 \cup E(r) \cdot 2. r \in \mathsf{rgn}(t)$$

2) *No multiple writers exist in any epoch.*

$$\begin{aligned} \forall r. |E(r) \cdot 1| > 1 &\implies \forall t \in E(r) \cdot 1. r \notin t(\mathsf{wr}) \\ \wedge\ \forall r. |E(r) \cdot 2| > 1 &\implies \forall t \in E(r) \cdot 2. r \notin t(\mathsf{wr}) \end{aligned}$$

3) *The epoch privilege represents the privilege of the current epochs.*

$$\forall r. P(r) = pv \implies \forall t \in E(r) \cdot 1. r \in t(pv)$$

4) *If the current epoch for a region is of readers, the previous epoch should be of a writer.*

$$\forall r. P(r) = \mathsf{rd} \implies \forall t \in E(r) \cdot 2. r \notin t(\mathsf{rd}) - t(\mathsf{wr})$$

We write $\mathsf{valid}(E, P)$ *when E and P are valid.*

Second, the epoch map must be *consistent* with the task graph.

Definition 9. *The r-connected subgraph of task graph $\langle T, D \rangle$ is $\langle T_r, D_r \rangle$, where*

$$T_r = \{t \in T \mid r \in \mathsf{rgn}(t)\} \text{ and}$$
$$D_r = \{\langle t_1, t_2 \rangle \in D \mid t_1, t_2 \in T_r\}.$$

We write G_r to denote the r-connected subgraph of G.

Definition 10. *Epoch map E is consistent with task graph G iff $E(r)$ and the r-connected subgraph $G_r = \langle T_r, D_r \rangle$ satisfy the following conditions for every region r.*

1) *Tasks in the current epoch are at the frontier of G_r.*

$$\forall t \in E(r) \cdot 1 \cap T_r.\ \nexists t'. \langle t, t' \rangle \in D_r$$

2) *All tasks in the previous epoch are connected to those in the current epoch.*

$$\forall t \in E(r) \cdot 2 \cap T_r. \forall t' \in E(r) \cdot 1 \cap T_r. \langle t, t' \rangle \in D_r$$

We write $\mathsf{consistent}(E, G)$ *when E is consistent with G.*

Note that these consistency conditions ignore the tasks that no longer exist in the task graph. This means that an epoch map remains consistent with a task graph even after some tasks finish their execution and are removed from that graph. In Section III-C, we extend the $\mathcal{D}_{\mathsf{epoch}}$ algorithm to incorporate task removals during dependence analysis and show that the algorithm is still correct.

The function swmr adds a new task to the epochs while maintaining the SWMR invariant.

$$\mathsf{swmr}(\langle T_1, T_2 \rangle, pv_1, t, pv_2) \triangleq \begin{cases} \langle T_1 \cup \{t\}, T_2 \rangle & \text{when } pv_1 = pv_2 = \mathsf{rd} \\ \langle \{t\}, T_1 \rangle & \text{otherwise} \end{cases}$$

Once the epochs are adjusted, the dependencies between a new task and those in the previous epoch (written as $E'(r) \cdot 2$ in

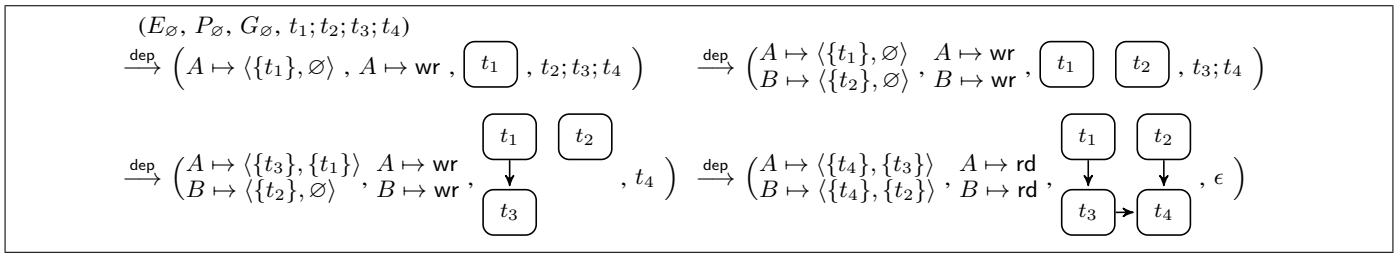


Fig. 5: Anatomy of $\mathcal{D}_{\mathsf{epoch}}(p_2)$

the rules) are registered to the task graph G using the function insert:

$$\mathsf{insert}(\langle T, D\rangle, T', t) \triangleq \langle T \cup \{t\}, D \cup \{\langle t', t\rangle \mid t' \in T' \cap T \wedge t' \neq t\}\rangle$$

At any point of dependence analysis, the $\mathcal{D}_{\mathsf{epoch}}$ algorithm keeps the epoch map and epoch privilege valid, and also keeps the epoch map consistent with the task graph.

Lemma 5.

$$\mathsf{valid}(E,P) \ \wedge\ \mathsf{consistent}(E,G) \wedge (E,P,G,t;p) \xrightarrow{\mathsf{dep}} (E',P',G',p) \Longrightarrow \mathsf{valid}(E',P') \wedge \mathsf{consistent}(E',G')$$

Proof. We proceed by case analysis on the relation $\xrightarrow{\mathsf{dep}}$. Then, the proof directly follows from the facts that the function swmr preserves the validity of the epoch map and epoch privilege, and that the function insert recovers the consistency of the epoch map with the task graph after the new task t is added to the graph. □

The most important property of the $\mathcal{D}_{\mathsf{epoch}}$ algorithm is that it produces a sound and complete task graph.

Definition 11. *$\mathcal{D}_{\mathsf{epoch}}(p) = G$, where G is a task graph that satisfies the following condition:*

$$(E_\varnothing, P_\varnothing, G_\varnothing, p) \xrightarrow{\mathsf{dep}}{}^{*} (E, P, G, \epsilon),$$

where $G_\varnothing$ denotes an empty task graph $\langle\varnothing,\varnothing\rangle$.

Theorem 6. $[\![\mathcal{D}_{\mathsf{epoch}}(p)]\!] = [\![\mathcal{D}_{\mathsf{simple}}(p)]\!]$

Proof. Let $\mathsf{clo}(G)$ be the transitive closure of G; i.e., $\mathsf{clo}(\langle T, D\rangle) = \langle T, D^+\rangle$. It is straightforward to show that

$$\mathsf{clo}(\mathcal{D}_{\mathsf{epoch}}(p)) = \mathsf{clo}(\mathcal{D}_{\mathsf{simple}}(p)) \Longrightarrow [\![\mathcal{D}_{\mathsf{epoch}}(p)]\!] = [\![\mathcal{D}_{\mathsf{simple}}(p)]\!].$$

We first prove $\mathsf{clo}(\mathcal{D}_{\mathsf{epoch}}(p)) \supseteq \mathsf{clo}(\mathcal{D}_{\mathsf{simple}}(p))$. We only need to show that every two dependent tasks in p have a path in $\mathcal{D}_{\mathsf{epoch}}(p)$, because for independent tasks t_1 and t_2, if a path exists from from t_1 to t_2 in $\mathcal{D}_{\mathsf{simple}}(p)$, we can always find another task t_3 on that path such that $t_1 \Leftrightarrow t_3$ and $t_3 \Leftrightarrow t_2$. We proceed by induction on the relation $\xrightarrow{\mathsf{dep}}{}^{*}$. Let t be a new task, $G_1 = \langle T_1, D_1\rangle$ be the task graph before analyzing t, and $G_2 = \langle T_2, D_2\rangle$ be the task graph after analyzing t ($T_2 = T_1 \cup \{t\}$ and $D_2 \supset D_1$). Pick a task t' in G_1 such that $t' \Leftrightarrow t$. If t' is in one of the current epochs, t' is at the frontier of G, and thus by the definition of insert, $\langle t', t\rangle \in D_2$. If t' is in one of the previous epochs, we can always find another task t'' in one of the current epochs such that $\langle t', t''\rangle \in D_1$ and $\langle t'', t\rangle \in D_2$. Therefore, $\langle t', t\rangle \in D_2^+$. Finally, if t' is not in any of the epochs, there must be a task t''' in either one of the epochs such that $t' \Leftrightarrow t'''$, due to the property of valid epochs alternating their privileges. From the induction hypothesis, we have $\langle t', t'''\rangle \in D_1^+$. Then, we show $\langle t''', t\rangle \in D_2^+$ similarly as in the first two cases. Therefore, we conclude $\langle t', t\rangle \in D_2^+$.

Now we prove $\mathsf{clo}(\mathcal{D}_{\mathsf{epoch}}(p)) \subseteq \mathsf{clo}(\mathcal{D}_{\mathsf{simple}}(p))$. Pick two tasks t_1 and t_2 such that a path from t_1 to t_2 exists in $\mathcal{D}_{\mathsf{epoch}}(p)$. We only need to show that a path from t_1 to t_2 also exists in $\mathcal{D}_{\mathsf{simple}}(p)$ when t_1 and t_2 are independent, because otherwise t_1 and t_2 obviously have an edge in $\mathcal{D}_{\mathsf{simple}}(p)$. Suppose t_2 is a new task. Because t_1 and t_2 have a path in $\mathcal{D}_{\mathsf{epoch}}(p)$, they must share at least one region. Furthermore, for one of the common regions, t_1 must not be in any of the epochs, because otherwise there would not be a path between t_1 and t_2, or t_1 and t_2 would be dependent. Then, we can always find another task t_3 in either one of the epochs for this common region such that $t_1 \Leftrightarrow t_3$ and $t_3 \Leftrightarrow t_2$, due to the SWMR invariant. Therefore, we can also find a path from t_1 and t_2 that goes through t_3 in $\mathcal{D}_{\mathsf{simple}}(p)$, which completes the proof. □

Example 4. *Figure 5 shows how $\mathcal{D}_{\mathsf{epoch}}$ analyzes program p_2 in Example 2. Note that $p_2 = t_1;t_2;t_3;t_4$, where*

$$t_1 = \big\langle 1, \{\langle A, \mathsf{wr}\rangle\}\big\rangle, \quad t_2 = \big\langle 2, \{\langle B, \mathsf{wr}\rangle\}\big\rangle,$$
$$t_3 = \big\langle 3, \{\langle A, \mathsf{wr}\rangle\}\big\rangle, \text{ and } t_4 = \big\langle 4, \{\langle A, \mathsf{rd}\rangle, \langle B, \mathsf{rd}\rangle\}\big\rangle.$$

The resulting task graph $\mathcal{D}_{\mathsf{epoch}}(p_2)$ is sound and complete.

To characterize a set of programs for which $\mathcal{D}_{\mathsf{epoch}}$ produces a parsimonious task graph, we must understand when $\mathcal{D}_{\mathsf{epoch}}$ can introduce a transitive edge between tasks. Suppose that $\mathcal{D}_{\mathsf{epoch}}$ adds a transitive edge from task t_1 to task t_3 when the task graph already has two edges $\langle t_1, t_2\rangle$ and $\langle t_2, t_3\rangle$. Then, tasks t_1 and t_3 must have a dependence for a region that is not written by t_2; otherwise, $\mathcal{D}_{\mathsf{epoch}}$ could not connect t_1 directly to t_3, because of the SWMR invariant. In other words, $\mathcal{D}_{\mathsf{epoch}}$ constructs a parsimonious task graph if tasks in the program never form such a "triangle", stated more generally as follows:

Theorem 7. *Task graph $\mathcal{D}_{\mathsf{epoch}}(p)$ is parsimonious if program p satisfies the following condition:*

$$\forall t_1, t_2.p \vdash t_1 \succ^+ t_2 \wedge t_1 \Leftrightarrow t_2 \Longrightarrow p \vdash t_1 \succ t_2 \vee \forall r \in \mathsf{WR}(t_1,t_2).\exists t_3.p \vdash t_1 \succ^+ t_3 \succ^+ t_2 \wedge r \in t_3(\mathsf{wr}),$$

where $\mathsf{WR}(t_1,t_2) = (t_1(\mathsf{wr}) \cap \mathsf{rgn}(t_2)) \cup (\mathsf{rgn}(t_1) \cap t_2(\mathsf{wr}))$.

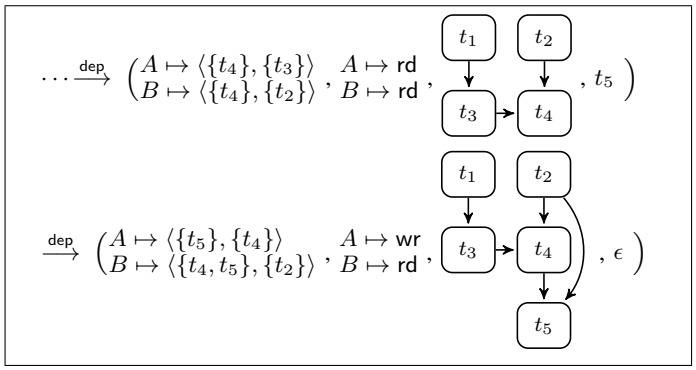


Fig. 6: Last transition of $\mathcal{D}_{\mathsf{epoch}}(p_3)$

Proof. Suppose $\mathcal{D}_{\mathsf{epoch}}(p)$ is not parsimonious despite the condition. Let $\langle t, t'\rangle \in D$ be a transitive edge in $\mathcal{D}_{\mathsf{epoch}}(p)$. Then, there exists a path of edges $\langle t, t_1\rangle$, $\langle t_1, t_2\rangle$, $\ldots$, $\langle t_{n-1}, t_n\rangle$, and $\langle t_n, t'\rangle$ for some n, where $p \vdash t \succ^+ t_1 \succ^+ \cdots \succ^+ t_n \succ^+ t'$. Pick a region r in $\mathsf{WR}(t, t')$. ($\mathsf{WR}(t, t')$ cannot be empty because $t \Leftrightarrow t'$.) Because $\mathcal{D}_{\mathsf{epoch}}$ only connects tasks in epochs, task t' was in the current epoch when t was in the previous epoch. Furthermore, for any task t_i such that $r \in \mathsf{rgn}(t_i)$, we have $r \notin t_i(\mathsf{wr})$ due to the validity of the epoch map. This contradicts the theorem's condition that there must exist a task t'' such that $p \vdash t \succ^+ t'' \succ^+ t'$ and $r \in t''(\mathsf{wr})$. □

Example 5. *Program p_2 in Example 2 satisfies the condition in Theorem 7, and thus task graph $\mathcal{D}_{\mathsf{epoch}}(p_2)$, which is the same as G_2 in Figure 3b, is parsimonious as discussed in Example 2.*

Example 6. *Consider a slightly modified program $p_3 = p_2; t_5$, where*

$$t_5 = \big\langle 5, \{\langle A, \mathsf{wr}\rangle, \langle B, \mathsf{rd}\rangle\}\big\rangle.$$

Note that $t_2(\mathsf{wr}) \cap \mathsf{rgn}(t_5) = \{B\} \not\subseteq \varnothing = t_4(\mathsf{wr})$. Figure 6 shows that $\mathcal{D}_{\mathsf{epoch}}(p_3)$ is not parsimonious, because there exists transitive edge between t_2 and t_5.

The time complexity of $\mathcal{D}_{\mathsf{epoch}}$ for program p is $O(|p||R|)$, where R denotes the set of regions used in p. Since the number of regions used in each task does not typically grow as the program size increases, the effective time complexity is linear in the program size, which is asymptotically better than the quadratic time complexity of $\mathcal{D}_{\mathsf{simple}}$.

C. Extension to Whole-Program Execution

Once a task graph is constructed, tasks in that graph are executed based on the schedule of the graph. We model the execution of tasks in a task graph as a process of finding one of the schedules expressed by that graph.

The following transition relation $\xrightarrow{\mathsf{exec}}$ describes one step of task graph execution.

$$\boxed{(G, sc) \xrightarrow{\mathsf{exec}} (G, sc)}$$

$$\frac{t \in T \quad \nexists t'.(t', t) \in D \qquad T' = T - \{t\} \quad D' = \{(t, t') \in D \mid t, t' \in T'\}}{(\langle T, D\rangle, sc) \xrightarrow{\mathsf{exec}} (\langle T', D'\rangle, sc; t)}$$

$$\boxed{(E, P, G, p, sc) \xrightarrow{\mathsf{prog}} (E, P, G, p, sc)}$$

$$\frac{(E, P, G, p) \xrightarrow{\mathsf{dep}} (E', P', G', p')}{(E, P, G, p, sc) \xrightarrow{\mathsf{prog}} (E', P', G', p', sc)}$$

$$\frac{(G, sc) \xrightarrow{\mathsf{exec}} (G', sc')}{(E, P, G, p, sc) \xrightarrow{\mathsf{prog}} (E, P, G', p, sc')}$$

Fig. 7: Whole-program execution

In each transition, the rule finds a task with no predecessors in the task graph and appends it to the schedule, signifying that the task is executed. The task graph execution terminates once it reaches an empty task graph. A complete execution of a task graph finds one of the schedules expressed by that graph.

Definition 12. *A schedule sc is an execution of task graph G if it satisfies $(G, \varnothing) \xrightarrow{\mathsf{exec}}{}^* (G_\varnothing, sc)$.*

Lemma 8. *Let sc be an execution of task graph G. Then, $sc \in [\![G]\!]$.*

Proof. We proceed by induction on $\xrightarrow{\mathsf{exec}}{}^*$. We define an induction hypothesis that preserves the removed edges and nodes in an accumulated graph, distinct from the graph being executed. The current schedule is defined to be a schedule of the accumulated graph, which when specialized to the definition of initial and final states of execution gives the desired result. □

In a real execution of a program, tasks do not wait until the whole task graph is constructed, and rather the dynamic dependence analysis and the task execution are interleaved, because materializing the graph for the whole task stream is excessive and even infeasible if the stream is long. This interleaving can be modeled by our formalism as a nondeterministic choice between the dynamic dependence analysis and task graph execution at each step of program execution. The transition relation $\xrightarrow{\mathsf{prog}}$ in Figure 7 describes this step of program execution.

The whole-program execution terminates once there is no task to analyze or schedule. We prove that this whole-program execution using the $\mathcal{D}_{\mathsf{epoch}}$ algorithm produces a valid schedule of a program.

Theorem 9.

$$(E_\varnothing, P_\varnothing, G_\varnothing, p, \epsilon) \xrightarrow{\mathsf{prog}}{}^* (E, P, G_\varnothing, \epsilon, sc) \implies p \vdash sc$$

Proof. Follows directly from Lemma 8, Theorem 6, and Lemma 4, and induction on $\xrightarrow{\mathsf{prog}}{}^*$. □

Example 7. *Figure 8 illustrates a sample execution of program p_3 in Example 5.*

IV. Related Work

Dynamic dependence analysis is key to providing sequential semantics for programs in all implicitly parallel tasking systems [1]–[5]. To the best of our knowledge, none of the

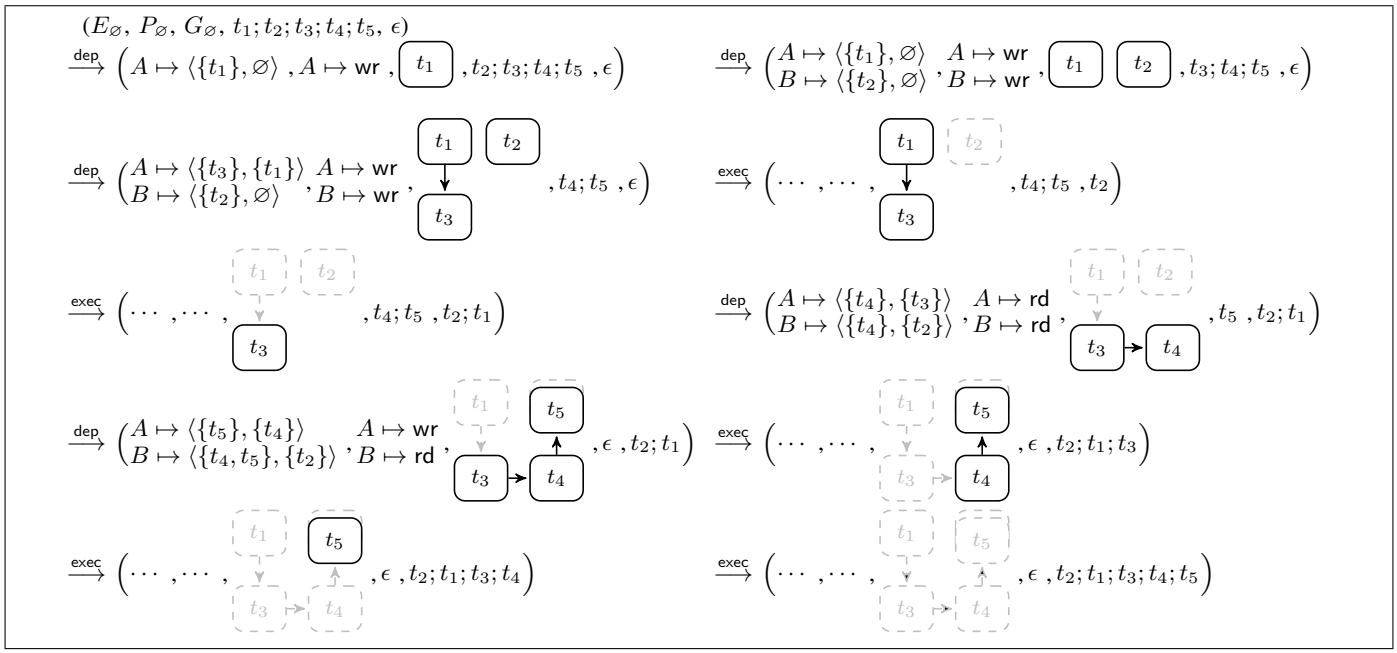


Fig. 8: Execution of p_3 (the epoch map and privilege are elided when they remain the same as in the previous step)

systems have rigorously proven correctness of their dependence analysis algorithms. We believe that our theoretical framework can serve as a foundation for achieving formal correctness of these algorithms.

A series of papers [9]–[11] formalize the semantics of a core BSP (Bulk-Synchronous Parallel) language that models a bulk synchronous subset of MPI. The key contribution of these papers is a proof of deterministic semantics; a BSP program yields the same result regardless of any non-determinism in execution order. However, their semantic determinism is a weak guarantee because it does not imply the correctness of the execution. As demonstrated by Gava and Fortin [9], one must still prove the equivalence between a BSP program and its sequential counterpart, which is a non-trivial work on its own and cannot be reused for other programs. On the other hand, the formal guarantee in this paper is stronger because it provides both determinism and correctness with respect to the sequential semantics.

V. Conclusion

We present a theoretical framework for proving the correctness of dynamic dependence analysis algorithms. We prove that the epoch-based dependence analysis algorithm $\mathcal{D}_{\mathsf{epoch}}$ produces a sound and complete task graph for a program. We also show that the generated task graph is parsimonious when the tasks in that program do not share more than one region.

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, award DE-NA0002373-1 from the Department of Energy National Nuclear Security Administration, NSF grant CCF-1160904, and an internship at Los Alamos National Laboratory.

References

[1] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Supercomputing (SC)*, 2012.

[2] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model," Inria Bordeaux Sud-Ouest ; Bordeaux INP ; CNRS ; Université de Bordeaux ; CEA, Research Report RR-8927, Jun. 2016.

[3] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in parsec: A data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '17, 2017.

[4] M. Tillenius, "Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization," *SIAM J. Scientific Computing*, vol. 37, no. 6, 2015.

[5] A. Zafari, E. Larsson, and M. Tillenius, "Ductteip: An efficient programming model for distributed task based parallel computing," *CoRR*, vol. abs/1801.03578, 2018. [Online]. Available: http://arxiv.org/abs/1801.03578

[6] M. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," Ph.D. dissertation, Stanford University, 2014.

[7] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," in *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[8] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[9] F. Gava and J. Fortin, "Formal semantics of a subset of the paderborn's bsplib," in *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2008, Dunedin, Otago, New Zealand, 1-4 December 2008*, 2008, pp. 269–276.

[10] ——, "Two formal semantics of a subset of the paderborn university bsplib," in *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009, Weimar, Germany, 18-20 Febuary 2009*, 2009, pp. 44–51.

[11] J. Fortin and F. Gava, "Towards mechanised semantics of hpc: The bsp with subgroup synchronisation case," in *Proceedings of the ICA3PP International Workshops and Symposiums on Algorithms and Architectures for Parallel Processing - Volume 9532*, 2015, pp. 222–237.