Parallel Cost Analysis

ELVIRA ALBERT, Complutense University of Madrid
JESÚS CORREAS, Complutense University of Madrid
EINAR BROCH JOHNSEN, University of Oslo
KA I PUN, Western Norway University of Applied Sciences and University of Oslo
GUILLERMO ROMÁN-DÍEZ, Universidad Politécnica de Madrid

This paper presents *parallel cost analysis*, a static cost analysis targeting to over-approximate the cost of parallel execution in distributed systems. In contrast to the standard notion of *serial cost*, parallel cost captures the cost of synchronized tasks executing in parallel by exploiting the true concurrency available in the execution model of distributed processing. True concurrency is challenging for static cost analysis, because the parallelism between tasks needs to be soundly inferred, and the waiting and idle processor times at the different locations need to be accounted for. Parallel cost analysis works in three phases: (1) it performs a *block-level* analysis to estimate the serial costs of the blocks between synchronization points in the program; (2) it then constructs a *distributed flow graph* (DFG) to capture the parallelism, the waiting and idle times at the locations of the distributed system; and (3) the parallel cost can finally be obtained as the *path of maximal cost* in the DFG. We prove the correctness of the proposed parallel cost analysis, and provide a prototype implementation to perform an experimental evaluation of the accuracy and feasibility of the proposed analysis.

CCS Concepts: • **Software and its engineering → Software verification**; **Automated static analysis**; **Distributed programming languages**;

General Terms: Complexity, Languages, Theory.

Additional Key Words and Phrases: Static Analysis, Distributed Systems, Resource Analysis

ACM Reference format:
Elvira Albert, Jesús Correas, Einar Broch Johnsen, Ka I Pun, and Guillermo Román-Díez. 2018. Parallel Cost Analysis. *ACM Trans. Comput. Logic* 1, 1, Article 1 (January 2018), 38 pages.
DOI: 10.1145/3274278

1 INTRODUCTION

Welcome to the age of distributed and multicore computing, in which software must cater for massively parallel execution. Looking beyond parallelism between independent tasks, *regular parallelism* involves tasks which are mutually dependent [47]: synchronization and communication are becoming major bottlenecks for the efficiency of distributed software. We consider a model of computation which separates the asynchronous spawning of new tasks to different locations for task processing from the synchronization between these tasks. The extent to which the software succeeds in exploiting the potential parallelism of the distributed locations depends on its synchronization patterns: synchronization points between dynamically generated parallel tasks restrict concurrency. We use the term *parallel cost* to describe the cost of parallel execution for such distributed software. The parallel cost is related to so-called critical paths in project management [37]: a critical path

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 ACM. 1529-3785/2018/1-ART1 $15.00
DOI: 10.1145/3274278

refers to the sequence of dependent activities across locations which adds up to the longest overall duration. Compared to critical paths in project management, the parallel cost of distributed software needs to deal with aliasing of references to distributed locations and with the dynamic spawning of tasks. Our goal in this paper is to develop static analysis techniques to approximate the parallel cost, i.e., to quantify how synchronization restricts the parallel execution of distributed software.

The concurrency model considered in this paper is based on the formally defined language ABS [34], which completely decouples method calls from synchronization and supports cooperative concurrency [35, 36] between tasks. This decoupling of communication and synchronization leads to very flexible forms of synchronization between tasks running at different locations. This concurrency model is in fact quite general since it subsumes *rendezvous synchronization* as in Ada [17]; actor-based *asynchronous message passing* [3] known from, e.g., Erlang [15] and Akka [29, 51]; *futures* [16] in Argus [39] and MultiLisp [30] and in *active-object languages and frameworks* such as ConcurrentSmalltalk [54], ABCL [55], Eiffel// [21], CJava [22], and ProActive [20]; as well as *fork/join parallelism* [40]. Different concurrency models for actor-based languages, including ABS, have been the subject of a recent survey [24]; compared to standard communication models using futures, ABS additionally supports *cooperative concurrency* by allowing a task to suspend execution if it is blocked waiting for a future to be resolved.

This paper introduces a novel static analysis to study the efficiency of computations in this setting, by approximating how synchronization between blocks of serial execution influences parallel cost. The analysis builds upon well-established static cost analyses for serial execution [5, 28, 56]. We assume that a serial cost analysis returns a "cost" for the serial blocks which measures their efficiency. Traditionally, the metrics used in cost analysis [49] is based on counting the number of execution steps, because this cost model appears as the best abstraction of time for software. Our parallel cost analysis could also be used in combination with worst-case execution time (WCET) analysis [1] by assuming that the cost of the serial blocks is given by a WCET analysis.

1.1 Summary of Contributions

Previous work on cost analysis of distributed systems [5] accumulates costs from different locations, but ignores the parallelism of the distributed execution model. This paper presents, to the best of our knowledge, the first static analysis to infer the parallel cost of distributed systems, which takes into account the parallel execution of code across the locations of the distributed system to infer more accurate bounds on the parallel cost. Our analysis works in the following steps, which are the main contributions of the paper:

(1) *Block-level cost analysis of serial execution.* We extend an existing cost analysis framework for the serial execution of distributed programs in order to infer information at the granularity of synchronization points.

(2) *Distributed flow graph (DFG).* We define the notion of DFG to represent all possible (equivalence classes of) paths that can be taken by the execution of the distributed program.

(3) *Path Expressions.* The problem of finding the parallel cost of executing a distributed program boils down to finding the path of maximal cost in the corresponding DFG. Paths in the DFG are computed by means of the single-source path expression problem [48], which finds regular expressions representing all paths.

(4) *Parallel cost analysis.* By relying on the path expressions in (3) and the block level cost analysis of serial execution in (1), we introduce a novel technique to compute, with three different levels of precision, the parallel cost of the program.

(5) *Implementation and experimental evaluation.* As a practical contribution, we demonstrate the accuracy and feasibility of the presented cost analysis by implementing a prototype analyzer

of parallel cost within the SACO system, [6], a static analyzer for distributed concurrent programs. Experimental evaluation on some typical distributed programs achieve gains up to 50% with respect to a serial cost analysis The tool can be used online from a web interface available at http://costa.fdi.ucm.es/parallelcost.

This article revises and extends a conference paper published in the proceedings of SAS'15 [10] in several crucial aspects: (1) We give an improved semantics of the language and formally prove the soundness of our analysis with respect to this semantics. (2) The notion of distributed flow graph, a key element for the analysis proposed in this article, is improved to clarify the role of synchronization instructions in the parallel program. (3) We have added the notion of *coverage*, as well as Lemma 6.5 and Theorem 6.7 that relate execution traces of the program to paths in the distributed flow graph. A proof of this theorem is provided. (4) Section 7 is new to this article and proposes several ways to compute the cost of parallel programs with different levels of accuracy. (5) Finally, the experimental evaluation of the analysis has been extended, measuring the different approaches described in Section 7.

1.2 Organization of this Article

Section 2 introduces the formal model of distributed programs with explicit locations, Section 3 defines the cost of parallel execution in a distributed system, and Section 4 presents the various components on which the analysis is based. We then show the analysis for serial execution in Section 5, discuss the method used to infer the cost of parallel execution in Section 6, and illustrate the computation of parallel cost and different approaches to enhancing the precision of the analysis in Section 7. Section 8 presents experimental results by applying our analysis to some typical distributed systems and Section 9 discusses related work and concludes the paper.

2 THE MODEL OF DISTRIBUTED PROGRAMS

In this paper, we consider a distributed programming model with explicit locations. Each location represents a processor with an unordered buffer of pending tasks. Initially all processors are idle. When the task buffer of an idle processor is non-empty, some task is selected for execution. Besides accessing the global storage of its own processor, each task can post tasks to the buffer of any processor, including its own, and synchronize with the reception of tasks. When a task completes, its processor becomes idle again, chooses the next pending task, and continues executing.

2.1 Syntax

The number of distributed locations needs not be fixed a priori (e.g., locations may be virtual). Syntactically, a location will therefore be similar to an *object* and can be dynamically created using an instruction newLoc. A program $P(\bar{x})$, with formal parameters $\bar{x}$, consists of a set of methods. The notation $\bar{x}$ is used as a shorthand for $x_1, \ldots, x_n$, and similarly for other names. Methods are of the form M::=$T\ m(\overline{T\ x})\{s\}$. A distinguished method main is used to start the execution of the program with the program's actual parameters. Statements s are defined as follows:

$$s ::= s; s \mid x{=}e \mid \text{if } e \text{ then } s \text{ else } s \mid \text{while } e \text{ do } s \mid \text{return } x \mid x{=}\text{newLoc} \mid x.m(\bar{z}) \mid f{=}x.m(\bar{z}) \mid$$
$$\text{await } f? \mid x{=}f.\text{get}$$

where e is an expression, x, z are variables, f is a future variable and m is a method name. The special location identifier *this* denotes the current location. For the sake of generality, the syntax of expressions e and types T is left open.

$$\frac{\text{fresh}(lid_2),\ l_1' = l_1[x \mapsto lid_2]}{\begin{array}{c} loc(lid_1, tid_1)\ tsk(tid_1, lid_1, m_1, l_1, x = \mathsf{newLoc}; s) \rightsquigarrow \\ loc(lid_1, tid_1)\ tsk(tid_1, lid_1, m_1, l_1', s)\ loc(lid_2, \bot) \end{array}} \quad (\text{NEWLOC})$$

$$\frac{l_1(x) = lid_2,\ \text{fresh}(tid_2),\ l_2 = \text{buildLocals}(\bar{z}, m)}{\begin{array}{l} loc(lid_1, tid_1)\ tsk(tid_1, lid_1, m_1, l_1, x.m(\bar{z}); s) \\ \quad \rightsquigarrow loc(lid_1, tid_1)\ tsk(tid_1, lid_1, m_1, l_1, s)\ tsk(tid_2, lid_2, m, l_2, body(m)) \end{array}} \quad (\text{ASYNC})$$

$$\frac{l(x) = lid_2,\ \text{fresh}(tid_2),\ l_1' = l_1[f \mapsto tid_2],\ l_2 = \text{buildLocals}(\bar{z}, m)}{\begin{array}{l} loc(lid_1, tid_1)\ tsk(tid_1, lid_1, m_1, l_1, f = x.m(\bar{z}); s) \\ \quad \rightsquigarrow loc(lid_1, tid_1)\ tsk(tid_1, lid_1, m_1, l_1', s)\ tsk(tid_2, lid_2, m, l_2, body(m)) \end{array}} \quad (\text{ASYNC+FUT})$$

$$\frac{v = l(x)}{loc(lid, tid)\ tsk(tid, lid, m, l, \mathsf{return}\ x;) \rightsquigarrow loc(lid, \bot)\ tsk(tid, lid, m, l, \epsilon(v))} \quad (\text{RETURN})$$

$$\frac{l(f) = tid_1,\ tsk(tid_1, _, _, _, \epsilon(v)) \in \mathsf{Tsks}}{loc(lid, tid)\ tsk(tid, lid, m, l, \mathsf{await}\ f?; s) \rightsquigarrow loc(lid, tid)\ tsk(tid, lid, m, l, s)} \quad (\text{AWAIT1})$$

$$\frac{l(f) = tid_1,\ tsk(tid_1, _, _, _, s_1) \in \mathsf{Tsks},\ s_1 \neq \epsilon(v)}{loc(lid, tid)\ tsk(tid, lid, m, l, \mathsf{await}\ f?; s) \rightsquigarrow loc(lid, \bot)\ tsk(tid, lid, m, l, \mathsf{await}\ f?; s)} \quad (\text{AWAIT2})$$

$$\frac{l(f) = tid_1,\ tsk(tid_1, _, _, _, \epsilon(v)) \in \mathsf{Tsks},\ l' = l[x \mapsto v]}{loc(lid, tid)\ tsk(tid, lid, m, l, \mathsf{x}{=}f.\text{get}; s) \rightsquigarrow loc(lid, tid)\ tsk(tid, lid, m, l', s)} \quad (\text{GET})$$

$$\frac{\text{select}(\mathsf{Tsks}) = tid,\ tsk(tid, lid, _, _, s_1) \in \mathsf{Tsks},\ s_1 \neq \mathsf{await}\ f?; s,\ s_1 \neq \epsilon(v)}{loc(lid, \bot) \rightsquigarrow loc(lid, tid)} \quad (\text{SELECT1})$$

$$\frac{\text{select}(\mathsf{Tsks}) = tid_1,\ tsk(tid_1, lid, _, l_1, \mathsf{await}\ f?; s_1) \in \mathsf{Tsks}, l_1(f) = tid_2,\ tsk(tid_2, _, _, _, \epsilon(v)) \in \mathsf{Tsks}}{loc(lid, \bot) \rightsquigarrow loc(lid, tid_1)} \quad (\text{SELECT2})$$

Fig. 1. Semantics of Local Execution

2.2 Semantics

A *program state* S has the following form:

$S ::= \mathsf{Locs}\ \mathsf{Tsks}$

where Locs denotes the set of currently existing distributed locations and Tsks the set of tasks at these locations. Each location is a term $loc(lid, tid)$ where lid is the location identifier and tid is the identifier of the *active task* which holds the lock of the location or $\bot$ if the lock is free. Only the task holding the location's *lock* can be *active* (running) at this location. All other tasks spawned at a location are *pending* to be executed, or *finished* if they have terminated and released the lock. A *task* is a term $tsk(tid, lid, m, l, s)$ where tid is a unique task identifier, lid the location that executes

the task, m the name of the method executing in the task, l a mapping from local variables to their values, and s the sequence of statements to be executed or $s = \epsilon(v)$ if the task has terminated and the return value v is available.

Local reduction rules. The transition relation $\rightsquigarrow$ in Figure 1 defines the reduction steps at each distributed location. The treatment of sequential instructions is standard and thus omitted. In NewLoc, an active task tid_1 at location lid_1 creates a location lid_2 with a free lock. Async spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier tid_2 at location lid_2 (which may be lid_1). The language includes constructs to allow cooperative concurrency between the tasks at each location, in the style of concurrent (or active) object systems such as ABS [34]. To this end, the language is equipped with *future variables* which are used to check if the execution of an asynchronous task has finished. In particular, an asynchronous call is associated with a future variable f, denoted as f=x.p(), which is handled by Async+Fut. The rule is analogous to Async, except that the former stores the future associated with the spawned task to a variable in the local variable table l_1. Return stores the return value of a task in v and indicates the *termination* of the task by using $\epsilon(v)$ after the transition. In addition, the lock is released and will never be taken again by the task.

The instruction await f? allows synchronizing the execution of the current task with the task to which the future variable f is pointing; the expression f.get is used to retrieve the value returned by the completed task. In Await1, the task is waiting for a future variable which points to a finished task. Thus, the await statement is consumed. Note that the rule looks for task tid_1 in the set of currently existing tasks (denoted by Tsks). Otherwise, Await2 releases the lock without consuming the await statement; i.e., the future variable points to a task which is not yet finished. Get retrieves the return value of the finished task and stores it in l. Observe that the get expression is *blocking* as it keeps the lock of the location until the task finishes and returns. Finally, rules Select1 and Select2 return a selectable task and it obtains the lock of the location. A selectable task is a task that is not finished and can progress. In other words, a task is not selectable if either it has terminated or the next instruction to execute is an await but the awaited task is not finished.

Parallel reduction rules. In the distributed programs, execution can happen in parallel at different locations. Parallel reduction is captured by the two following rules:

$$\textsc{(Context)}\quad \frac{\begin{array}{ll}\mathsf{Locs}_1\ \mathsf{Tsks}_1 \rightsquigarrow \mathsf{Locs}_1'\ \mathsf{Tsks}_1', & \mathsf{Locs} = \mathsf{Locs}_1 \uplus \mathsf{Locs}_2,\ \mathsf{Tsks} = \mathsf{Tsks}_1 \uplus \mathsf{Tsks}_2,\\ \mathsf{Locs}_2\ \mathsf{Tsks}_2 \not\rightsquigarrow & \mathsf{Locs}' = \mathsf{Locs}_1' \uplus \mathsf{Locs}_2,\ \mathsf{Tsks}' = \mathsf{Tsks}_1' \uplus \mathsf{Tsks}_2,\end{array}}{\mathsf{Locs}\ \mathsf{Tsks} \rightsquigarrow \mathsf{Locs}'\ \mathsf{Tsks}'}$$

$$\textsc{(Parallel)}\quad \frac{\begin{array}{ll}\mathsf{Locs}_1\ \mathsf{Tsks}_1 \rightsquigarrow \mathsf{Locs}_1'\ \mathsf{Tsks}_1', & \mathsf{Locs} = \mathsf{Locs}_1 \uplus \mathsf{Locs}_2,\ \mathsf{Tsks} = \mathsf{Tsks}_1 \uplus \mathsf{Tsks}_2,\\ \mathsf{Locs}_2\ \mathsf{Tsks}_2 \rightsquigarrow \mathsf{Locs}_2'\ \mathsf{Tsks}_2', & \mathsf{Locs}' = \mathsf{Locs}_1' \uplus \mathsf{Locs}_2',\ \mathsf{Tsks}' = \mathsf{Tsks}_1' \uplus \mathsf{Tsks}_2',\end{array}}{\mathsf{Locs}\ \mathsf{Tsks} \rightsquigarrow \mathsf{Locs}'\ \mathsf{Tsks}'}$$

Here, $\uplus$ denotes a disjoint union operator over sets. In the rules, the sets Locs_1 and Locs_2 (resp. Tsks_1 and Tsks_2) are disjoint sets. Note that Locs_i and Tsks_i can be empty or singleton sets. Let $\mathsf{Locs}\ \mathsf{Tsks} \not\rightsquigarrow$ denote that the program state $\mathsf{Locs}\ \mathsf{Tsks}$ is non-reducible by the local reduction rules of Figure 1; i.e., the locations in Locs are either idle, all tasks in a location in Locs are finished, or the tasks are waiting for a future variable pending to be resolved.

The CONTEXT rule distinguishes locations where a reduction step can happen from locations where there is no applicable rule. If a location is idle but there is no task at the location or the task is blocked while waiting at a get expression, the reduction by CONTEXT simply returns the same location state. The PARALLEL rule recursively decomposes the parallel steps into local steps, enabling local transitions defined in Figure 1, denoted as $\mathtt{Locs}_1\ \mathtt{Tsks}_1 \rightsquigarrow \mathtt{Locs}'_1\ \mathtt{Tsks}'_1$. Due to the dynamic creation of distributed locations, we have that $\mathtt{Locs} \subseteq \mathtt{Locs}'$.

Observe that by reordering rule applications, a reduction, which consists of any combination of the rules CONTEXT and PARALLEL and of local reduction rules, is equivalent to a reduction by one application of CONTEXT followed by a series of applications of PARALLEL and local reduction rules applied to the same locations and tasks as before the reordering. We will henceforth work with this sequence as a representative of any such decomposition of parallel locations and denote such a *parallel transition step* by $W \equiv S \rightsquigarrow S'$, i.e., $\mathtt{Locs}\ \mathtt{Tsks} \rightsquigarrow \mathtt{Locs}'\ \mathtt{Tsks}'$ in which we perform a local reduction step $\rightsquigarrow$ (as defined in Figure 1) at every reducible distributed location in $\mathtt{Locs}$, abstracting from the specifics of decomposing the set of locations and tasks by CONTEXT and PARALLEL.

Parallel executions can be represented by execution traces defined as a sequence of parallel transition steps: we use the notation $t \equiv W_0 \longrightarrow W_1 \longrightarrow \cdots \longrightarrow W_i \longrightarrow \cdots$. Given a finite trace $W_0 \longrightarrow \cdots \longrightarrow W_n$ where $W_n \equiv \mathtt{Locs}_n\ \mathtt{Tsks}_n \rightsquigarrow \mathtt{Locs}_{n+1}\ \mathtt{Tsks}_{n+1}$, the last program state $\mathtt{Locs}_{n+1}\ \mathtt{Tsks}_{n+1}$ is called *final* if all tasks in $\mathtt{Tsks}_{n+1}$ have terminated (so their sequences of instructions are $\epsilon(_)$). Observe that if the execution of a program reaches a deadlock, the execution trace is finite but the last state is not final, as tasks at some locations may be mutually blocked in get or await instructions. We use $W_i \in t$ to indicate that W_i is a parallel transition in execution trace t.

Program execution. The execution of a program $P(\bar{x})$ with actual parameters $\bar{v}$ starts by activating the main method; the initial state of the program execution corresponds to the initial state of an initial location with identifier 0, in which there is a task with identifier 0 of the form $S_0 = loc(0,0)\ tsk(0, 0, \mathsf{main}, l[\bar{x} \mapsto \bar{v}], body(\mathsf{main}))$. Here, l maps local references to null (standard initialization) and formal parameters $\bar{x}$ to their initial values $\bar{v}$, while $body(\mathsf{main})$ refers to the sequence of instructions in the method main. The parallel execution proceeds from S_0 by parallel transition steps evaluating tasks at the different distributed locations. Since execution of a program $P(\bar{x})$ is non-deterministic in the selection of tasks, multiple (possibly infinite) traces may exist. We use $traces(P(\bar{x}))$ to denote the set of all possible execution traces for $P(\bar{x})$.

3 PARALLEL COST OF DISTRIBUTED SYSTEMS

The aim of this paper is to infer an *upper bound* that is an over-approximation of the *parallel cost* of executing a distributed system. Given a parallel transition $W \equiv \mathtt{Locs}\ \mathtt{Tsks} \rightsquigarrow \mathtt{Locs}'\ \mathtt{Tsks}'$, we denote by $\mathcal{P}(W)$ the parallel cost of the transition W. If we know the time taken by the transitions, $\mathcal{P}(W)$ refers to the time taken to evaluate all locations that execute an instruction. Thus, if two instructions execute in parallel, the parallel cost only accumulates the longest of their durations. For simplicity, we here assume that all locations execute one instruction in one cost unit, thus $\mathcal{P}(W) = 1$. Otherwise, the cost analysis of the serial cost must take into account the relative speeds of the locations (see Section 9). Given a trace $t \equiv W_0 \longrightarrow \cdots \longrightarrow W_i \longrightarrow \cdots$ of the parallel execution, we define $\mathcal{P}(t) = \sum_{W_i \in t} \mathcal{P}(W_i)$.

We denote by $\widehat{m}$ the cost of a (sequential) block m, for which parallelism does not affect the cost of execution. These cost expressions can be obtained by cost analyzers for serial execution [5]. It is not crucial for the contents of this paper to know how these expressions are obtained, nor what the cost expressions are for the other blocks and methods. Thus, in the sequel, we simply refer to such cost expressions in an abstract way as $\widehat{m_1}$, $\widehat{m_2}$, $\widehat{p_1}$, $\widehat{p_2}$, etc. The cost of blocks can be illustrated by the following example.

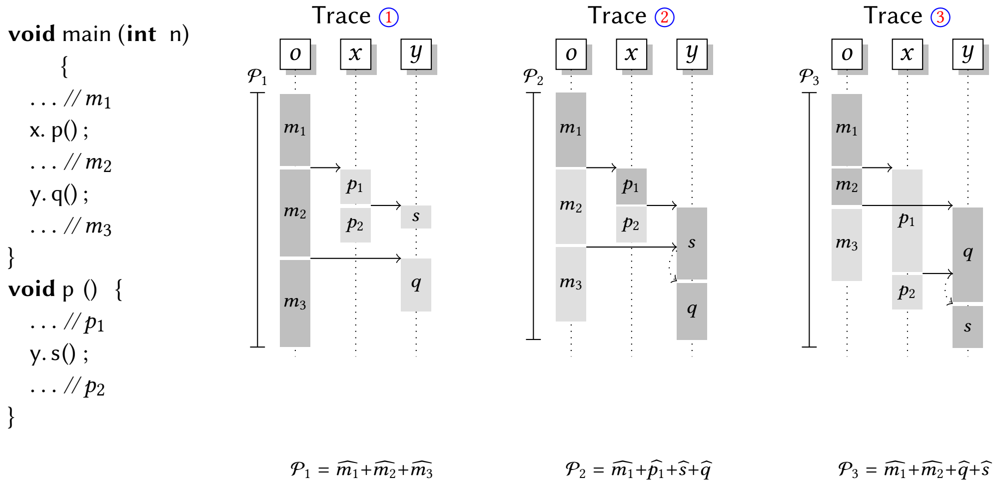


Fig. 2. Motivating example

Example 3.1. Consider a simple method main that spawns two tasks by calling p and q at locations x and y, as shown in Figure 2 (left). In turn, the method p spawns a task by calling s at location y. This program only features distributed execution, concurrent behaviours within the locations are ignored for now. Let $\widehat{m_1}$, $\widehat{m_2}$ and $\widehat{m_3}$ denote the cost from the beginning of m to the call x.p(), the cost between x.p() and y.q(), and the remaining cost of main, respectively, and let the costs $\widehat{p_1}$ and $\widehat{p_2}$ be analogous for method p. Note that the execution of the method main (at location o) proceeds while the spawned tasks are executed at the locations x and y. ■

Given a program $P(\bar{x})$, we now use $traces(P(\bar{x}))$ to formally define the parallel cost of the program:

Definition 3.2 (Parallel cost). The parallel cost of a program P on input values $\bar{x}$, denoted as $\mathcal{P}(P(\bar{x}))$, is defined as $\sup(\{\mathcal{P}(t) | t \in traces(P(\bar{x}))\})$.

The notion of parallel cost $\mathcal{P}$ corresponds to the cost consumed between the first instruction executed by the program at the initial location and the last instruction executed at any location by taking into account the parallel execution of instructions and idle times at the different locations. Note that Definition 3.2 uses the supremum to deal with possible infinite traces of non-terminating executions. The following example illustrates the notion of parallel cost:

Example 3.3. Three possible traces of the execution of Example 3.1 are shown to the right of Figure 2 (more traces are feasible). Below the traces, the expressions $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ show the parallel cost for each trace. The main observation here is that the parallel cost varies depending on the duration of the tasks, and it will be the worst value of such expressions, that is, $\mathcal{P} = \sup(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \ldots)$. In trace ①, the processor of location y becomes idle after executing s and must wait for task q to arrive. In trace ②, p_1 is shorter than m_2 and s executes before q. In trace ③, q is scheduled before s, resulting in different parallel cost expressions. ■

In the general case, the inference of parallel cost is complicated because: (1) When a task is spawned, the availability of the corresponding location is unpredictable, as this depends on whether the queue is empty or not; e.g., when task q is spawned we do not know if the processor is idle (Example 3.3, trace ①) or if it is occupied (Example 3.3, trace ②). Thus, all scenarios must be considered; (2) Locations can be dynamically created, and tasks can be dynamically spawned among

the different locations (e.g., from location o we spawn tasks at two other locations). Besides, tasks can be spawned in a circular way; e.g., task s could make a call back to location x; (3) Tasks can be spawned inside loops, and there might even be non-terminating loops that create an unbounded number of tasks. The analysis must approximate (upper bounds on) the number of tasks that the locations might have in their queues. These complications make the static inference of parallel cost a challenging problem that, to the best of our knowledge, has not been addressed previously. Existing frameworks for the cost analysis of distributed systems [5, 11] rely on a *serial* notion of cost; i.e., the resulting cost accumulates the cost executed by all locations created by the program execution. Thus, we obtain a serial cost that simply adds the costs of all methods: $\widehat{m_1} + \widehat{m_2} + \widehat{m_3} + \widehat{p_1} + \widehat{p_2} + \widehat{q} + \widehat{s}$.

4 BACKGROUND

In this section we briefly describe three crucial components for our parallel cost analysis: (1) a *points-to analysis* to get a finite abstract representation of the locations; (2) a *cost-centers based resource analysis* which defines the granularity of the analysis; and (3) a *may-happen-in-parallel* analysis to approximate those program points that might be simultaneously executing or pending to execute.

Points-to Analysis. Since locations can be dynamically created, we need an analysis that abstracts these locations into a *finite* abstract representation, and that tells us which (abstract) location a reference variable is pointing-to. Points-to analysis [5, 41, 44] addresses this problem by inferring the set of memory locations which a reference variable can *point-to*. Different abstractions can be used and our method is parametric on the chosen abstraction. Any points-to analysis that provides the following information with approximately similar precision can be used (our implementation uses [5, 41]): (1) the set of abstract locations $\mathcal{O}$, and (2) a function $pt(pp, v)$, which for a given program point pp and a variable v, returns the set of abstract locations in $\mathcal{O}$ to which v may point. In addition, given a concrete location $loc(lid, _)$ identified by lid and an abstract location o, we use $lid \in o$ to represent that lid is abstracted by o.

Example 4.1. In Figure 2 we have three different locations that variables o, x, y respectively point to. For simplicity, we will use the variable name in italics to refer to the abstract location inferred by the points-to analysis. Thus, $\mathcal{O}=\{o, x, y\}$. In this example, the points-to abstraction is very simple. However, in general, locations can be reassigned, passed as parameters, and can have multiple aliases, etc. It is fundamental to keep track of points-to information in an accurate way. ■

Resource Analysis with Cost Centers. Resource analysis is a rather complex process that comprises two phases: (1) the generation of cost relations from the source program (see [5, 8]) and (2) the computation of a closed-form upper bound from them (see [7]). Each of the two phases is defined by means of fixed-point static analyses (e.g., size analysis, alias analysis), includes also some program transformations (e.g., transformation into direct recursive form) and uses integer programming tools. Given that we can use the results of an off-the-shelf resource analyzer based on cost centers [5] without requiring any modification to it, we do not include the computation of the upper bounds in this article, and refer to the corresponding articles for details [5, 7].

Resource analyzers, in order to quantify the cost associated to an instruction, use the notion of *cost models*. A cost model is a function that assigns a measure of the cost of executing each instruction in the program. In our approach we consider a platform-independent cost model for bounding the *number of executed instructions* that can be defined as $\mathcal{M}_i(pp) = 1$, for all instructions. The upper bounds obtained by the resource analysis are of the form $\sum_{pp \in P} C_{pp} * \#exec_{pp}$, where $\#exec_{pp}$ bounds the number of times that pp is executed and C_{pp} bounds the cost of executing pp.

The use of such upper bounds to bound the resource consumption of distributed systems does not allow attributing the resource consumption to the location of the distributed system that actually consumes the cost. Instead, it returns a single cost expression that amalgamates the cost consumed by all locations. Cost centers were introduced in [5] for the analysis of distributed systems, such that each cost center is a symbolic expression of the form $c(l)$, where l is an abstract location identifier, used to separate the cost of each distributed location. Essentially, the resource analysis assigns the cost of a program point to the distributed location l by multiplying the cost of executing the instruction by the cost center of the abstract location $c(l)$. This way, the upper bounds that the analysis obtains are of the form:

$$\sum_{pp \in P \ \wedge \ o \in pt(this, pp)} c(o) * C_{pp} * \#exec_{pp}$$

where $C_{pp} * \#exec_{pp}$ bounds the total cost accumulated at program point pp executed by each abstract location o. By means of cost center based upper-bounds, given an abstract location of interest x, its cost can be obtained by replacing $c(o)$ by 1 when $o = x$ and by 0 when $o \neq x$. Analogously, it can be done for a set of abstract locations L, setting $c(o)$ to 1 when $o \in L$ an to 0 when $o \notin L$.

The idea of using cost centers in an analysis is of general applicability, and different approaches to cost analysis (e.g., cost analysis based on recurrence equations [49], invariants [28], or type systems [31]) can use this idea in order to extend their frameworks to a distributed setting. This is the only assumption that we make about the cost analyzer. Thus, we argue that our method can work in combination with other cost analyses for serial execution.

Example 4.2. For the code in Figure 2, we have three cost centers for the three locations that accumulate the costs of the blocks they execute; i.e., we have the expression $c(o)*\widehat{m_1} + c(o)*\widehat{m_2} + c(o)*\widehat{m_3} + c(x)*\widehat{p_1} + c(x)*\widehat{p_2} + c(y)*\widehat{s} + c(y)*\widehat{q}$. ■

May-happen-in-parallel. There exist *may-happen-in-parallel* (MHP) analyses for a wide variety of concurrency models (see, e.g., [25] for C Programs, [2] for X10 programs, [18] for Java programs, and [13, 38] for the ABS language). We use the existing MHP analysis for ABS in [13, 38] to approximate the tasks that could start their execution when the processor is released. This analysis infers pairs of program points whose execution *might* happen in parallel, or in an interleaved way, according to the following definition from [14]. Function $\mathit{first_pp}(s)$ returns the program point of the first instruction in a sequence of instructions s.

Definition 4.3 (Concrete MHP [14][1]). The concrete MHP set of P is defined as

$$\mathcal{E}_P = \bigcup \{ \ \mathcal{E}_{\mathsf{Tsks}} \mid \mathsf{Locs\ Tsks} \in t \wedge t \in traces(P(\bar{z})) \ \}$$

where

$$\begin{aligned}\mathcal{E}_{\mathsf{Tsks}} = \{(lid_1{:}\mathit{first_pp}(s_1), lid_2{:}\mathit{first_pp}(s_2)) \mid\ & tsk(tid_1, lid_1, _, _, s_1) \in \mathsf{Tsks} \ \wedge \\ & tsk(tid_2, lid_2, _, _, s_2) \in \mathsf{Tsks} \ \wedge \\ & tid_1 \neq tid_2\}\end{aligned}$$

The soundness of the analysis guarantees that if $(lid_1{:}pp_1, lid_2{:}pp_2)$ is not a MHP pair, there are no tasks executing pp_1 and pp_2 in parallel, or in an interleaved way, at locations lid_1 and lid_2 respectively. The MHP analysis can rely on a points-to analysis in exactly the same way as our overall analysis does. Hence, the results of the MHP are pairs of the form $(o_1{:}pp_1, o_2{:}pp_2)$ where $lid_1 \in o_1$ and $lid_2 \in o_2$ refer to the abstract locations in which pp_1 and pp_2 execute. Theorem 4.16

[1] We have adapted Definition 3.2 of [14] to the notation and the semantics used in this work.

of [14] guarantees that $\{(o_1{:}pp_1, o_2{:}pp_2) \mid (lid_1{:}pp_1, lid_2{:}pp_2) \in \mathcal{E}_P \wedge lid_1 \in o_1 \wedge lid_2 \in o_2\} \subseteq \widetilde{\mathcal{E}}_P$, where $\widetilde{\mathcal{E}}_P$ is the set of pairs inferred by the MHP analysis.

Example 4.4. The MHP analysis of the example shown in Figure 2 returns that all program points of p and q might execute in an interleaved way, that is, $\{(y{:}pp_s, y{:}pp_q) \mid pp_s \in s \wedge pp_q \in q\} \subseteq \widetilde{\mathcal{E}}_P$. Furthermore, as we only have one instance of main and p, the MHP analysis guarantees that $\{(o{:}pp_1, o{:}pp_2) \mid pp_1 \in \texttt{main} \wedge pp_2 \in \texttt{main}\} \cap \widetilde{\mathcal{E}}_P = \emptyset$ and that $\{(x{:}pp_1, x{:}pp_2) \mid pp_1 \in \texttt{p} \wedge pp_2 \in \texttt{p}\} \cap \widetilde{\mathcal{E}}_P = \emptyset$. ■

Although the MHP analyses of [5, 14], described in this section, are based on semantics that serialize the execution of the tasks, they can be applied to our parallel semantics. Recall the reduction rules of parallel transition steps from Section 2.2: Context distinguishes locations in which a reduction step can happen from those where there are no selectable tasks; and Parallel recursively decomposes the parallel steps at reducible locations into local steps defined in Figure 1. By means of these two rules, multiple local steps can be performed in one single parallel transition. Thus, we can serialize the semantics described in Section 2.2 by simply removing Context and Parallel rules from the semantics and selecting local rules non-deterministically. Given a parallel execution trace t and a parallel transition $W_i \in t$, where $W_i \equiv \texttt{Locs}_i\ \texttt{Tsks}_i \rightsquigarrow \texttt{Locs}_{i+1}\ \texttt{Tsks}_{i+1}$, we can always find a fragment of a sequential execution trace that mimics W_i, consisting of a collection of the local rules applicable to $\texttt{Locs}_i\ \texttt{Tsks}_i$ and applying them in the following order: first we apply rules Await1, Await2, Select1, Select2 and Get, and then the remaining rules in any order. Observe that the set of applicable rules contains, and consequently applies, at most one applicable rule per location. Consequently, it is sound to apply the results produced by the MHP analyses of [5, 14] to our parallel semantics.

5 BLOCK-LEVEL COST ANALYSIS OF SERIAL EXECUTION

The first phase of our method is to perform a *block-level* cost analysis of *serial* execution. This is a simple extension of an existing analysis [5] that provides costs at the level of the blocks, instead of locations, in which the program is partitioned. In previous work, other extensions have been performed to use costs at the level of specific program points [12] or at the level of complete tasks [11], but the partitioning required by our parallel cost analysis is different. Later in our approach, we need to be able to cancel out the cost associated to blocks whose execution occurs in parallel with other blocks that have higher cost. The key notion of the extension is *block-level cost centers*, as described below.

Block Partitioning. The need to partition the code into blocks will be clear when the second phase of the analysis is presented. We use the standard definition of basic block [4]:

Definition 5.1 (Basic block). Given a program P, a *basic block* (or *block*) is a maximal sequence of straight-line consecutive code in P with the properties that

(a) the flow of control can only enter the basic block through the first instruction in the block; and
(b) control will leave the block without halting or branching, except possibly at the last instruction in the block.

The block partition of a program is generated by conditional instructions and while loops, as in the standard construction of the control flow graph (CFG); and by asynchronous calls, which end the block where they are located; as well as by get and await instructions that start and end their single-instruction block. We define the following sets of blocks for their use in the subsequent analyses: $\mathcal{B}_{\mathsf{init}}$ the set of entry blocks for the methods; $\mathcal{B}_{\mathsf{exit}}$ the set of exit blocks for the methods;

$\mathcal{B}_{\text{call}}$ the set of blocks ending with an asynchronous call; and $\mathcal{B}_{\text{get}}$ and $\mathcal{B}_{\text{await}}$ the sets of single-instruction blocks respectively containing get and await instructions. In addition to these blocks, the standard partitioning of methods into blocks used for building the CFG for the method is carried out (e.g., conditional statement and loops introduce blocks for evaluating the conditions, edges to the continuations, etc.). We use $\mathcal{B}$ to refer to all block identifiers in the program. Given a program point pp, the function $block(pp)$ returns the identifier of the block that contains pp. Given a block identifier b, $pred(b)$ is the set of blocks from which there are outgoing edges to block b in the CFG.

Example 5.2. In Figure 2, the traces show the block partitioning for the methods m, p, q and s. Note that some of the blocks belong to multiple sets as defined above, namely $\mathcal{B}_{\text{init}} = \{m_1, p_1, s, q\}$, $\mathcal{B}_{\text{exit}}=\{m_3, p_2, s, q\}$, $\mathcal{B}_{\text{call}}=\{m_1, m_2, p_1\}$. For instance, m_1 is both an entry and a call block, and s is both an entry and an exit block as it is not partitioned. ■

As our analysis needs points-to information, we define the set $\overline{\mathcal{B}}$ as $\{o{:}b \mid o{:}b \in \mathcal{O} \times \mathcal{B} \wedge o \in pt(pp, this) \wedge pp \in b\}$. We define $\overline{\mathcal{B}}_{\text{call}} = \{o{:}b \mid o{:}b \in \overline{\mathcal{B}} \wedge b \in \mathcal{B}_{\text{call}}\}$, and $\overline{\mathcal{B}}_{\text{init}}$, $\overline{\mathcal{B}}_{\text{exit}}$, $\overline{\mathcal{B}}_{\text{get}}$ and $\overline{\mathcal{B}}_{\text{await}}$ analogously.

Example 5.3. In the example of Figure 2, we have that $\overline{\mathcal{B}} = \{o{:}m_1, o{:}m_2, o{:}m_3, x{:}p_1, x{:}p_2, y{:}q, y{:}s\}$. Similarly, we have that $\overline{\mathcal{B}}_{\text{call}} = \{o{:}m_1, o{:}m_2, x{:}p_1\}$, $\overline{\mathcal{B}}_{\text{init}} = \{o{:}m_1, x{:}p_1, y{:}q, y{:}s\}$ and $\overline{\mathcal{B}}_{\text{exit}} = \{o{:}m_3, x{:}p_2, y{:}q, y{:}s\}$. ■

Regarding the MHP analysis, although its results are given at the level of program points, they can be easily lifted to the level of blocks. We write $pp \in b$ (respectively, $i \in b$) to denote that the program point pp (respectively, instruction i) belongs to the block b. We use the notation $o_1{:}b_1 \parallel o_2{:}b_2$ to denote that the program points of $pp_1 \in b_1$ executing at o_1 and $pp_2 \in b_2$ executing at o_2 might happen in parallel, and, $o_1{:}b_1 \not\parallel o_2{:}b_2$ to indicate that they cannot happen in parallel.

Example 5.4. According to Example 4.4, the MHP analysis of the example shown in Figure 2 returns, among other results, that $y{:}s \parallel y{:}q$, $o{:}m_1 \not\parallel o{:}m_3$ and $x{:}p_1 \not\parallel x{:}p_2$. ■

Block-level Cost Centers. For our analysis, we need *block-level* granularity in the resource analysis. To achieve this granularity, we extend the notion of cost center in order to include two pieces of information: (1) the abstract location that might execute an instruction; and (2) the block that contains the instruction whose cost we are accounting. The first one is directly obtained by the resource analysis described in Section 4. The second piece can be obtained by using the following cost model, $\mathcal{M}_b(pp) = c(block(pp)) * 1$, which attributes one cost unit per instruction to its corresponding block. The *block-level upper-bound*, which can be obtained by the cost center-based cost analyzer [5] using $\mathcal{M}_b$, will be an expression of the form:

$$\mathcal{S}(P(\bar{x})) = \sum_{o \in pt(this, first_pp(b)) \,\wedge\, b \in \mathcal{B}} \overbrace{c(o){*}c(b)}^{c(o:b)} {*} C_b {*} \#exec_b,$$

where C_b is a cost expression that bounds the number of instructions executed for running all instructions in block b once and $\#exec_b$ bounds the number of times block b is executed by the abstract location o. In what follows, we use the notation $c(o{:}b)$ to refer to the multiplication $c(o){*}c(b)$.

With this block-level upper-bound, the serial cost can be distributed in terms of *block-level cost centers*, that is, the blocks in the program combined with the abstract locations where they can be executed. To obtain an upper bound on the number of instructions executed by an abstract location o running a given block b, we use the notation $\mathcal{S}(P(\bar{x}))|_{o:b}$, which returns the cost associated to the cost center $c(o{:}b)$ within the cost expression $\mathcal{S}(P(\bar{x}))$, i.e., the cost obtained by setting all $c(o'{:}b')$ to

0 (for $o' \neq o$ or $b' \neq b$) and setting $c(o{:}b)$ to 1. Analogously, given a set of block-level cost centers $N = \{o_0{:}b_0, \ldots, o_k{:}b_k\}$, we let $\mathcal{S}(\bar{x})|_N$ refer to the cost obtained by setting the cost centers $c(o_i{:}b_i)$ to: 1 when $o_i{:}b_i \in N$; and 0 otherwise. We omit P in cost expressions when it is clear from the context.

Example 5.5. The block-level upper-bound for the program shown in Figure 2 is:

$$\mathcal{S}(n){=}c(o{:}m_1){*}\widehat{m_1}{+}c(o{:}m_2){*}\widehat{m_2}{+}c(o{:}m_3){*}\widehat{m_3}{+}c(x{:}p_1){*}\widehat{p_1}{+}c(x{:}p_2){*}\widehat{p_2}{+}c(y{:}s){*}\widehat{s}{+}c(y{:}q){*}\widehat{q}.$$

Using this upper bound, we are able to obtain the cost for one block $o{:}m_2$, by replacing the cost center $c(o{:}m_2)$ by 1 and the other cost centers by 0, that is, $\mathcal{S}(n)|_{o:m_2} = \widehat{m_2}$. Similarly, we can obtain the cost for a set of blocks; e.g., given $B = \{o{:}m_1, o{:}m_2\}$, we get $\mathcal{S}(n)|_B = \widehat{m_1} + \widehat{m_2}$. ■

We use the resource analysis of [5] to obtain other relevant information needed by our analysis. By applying the analysis with a *cost model* that accounts cost 1 each time a block is executed, we can get a bound on the number of times each block is executed, that is, $C_b = 1$.

$$\mathcal{S}^t(P(\bar{x})) = \sum_{o \in pt(this, first_pp(b)) \,\wedge\, b \in \mathcal{B}} \overbrace{c(o){*}c(b)}^{c(o:b)} {*}1{*}\#exec_b,$$

As before, we can use the cost centers to parametrize the upper bound expression, thus, get the number of times a block is executed. We use $\mathcal{S}^t(P(\bar{x}))$ to refer to this upper bound. We can also restrict this upper bound to a particular set of blocks B, denoted $\mathcal{S}^t(P(\bar{x}))|_B$.

Example 5.6. Given the program shown in Figure 2, we get the expression

$$\mathcal{S}^t(n){=}c(o{:}m_1){*}1{+}c(o{:}m_2){*}1{+}c(o{:}m_3){*}1{+}c(x{:}p_1){*}1{+}c(x{:}p_2){*}1{+}c(y{:}s){*}1{+}c(y{:}q){*}1.$$

Analogously to Example 5.5, we get the upper bound of the number of times that $y{:}q$ is executed by $\mathcal{S}^t(n)|_{y:q}{=}1$. ■

6 PARALLEL COST ANALYSIS

This section presents our method to infer the execution cost of running on distributed system by taking advantage of the fact that certain blocks of code *must* execute in parallel. Thus, we only need to account for the highest cost among them. The analysis is based on the construction of a distributed flow graph that captures the parallelism in all possible paths that the execution can take, and uses this information with the block-level resource analysis.

6.1 Distributed Flow Graph

The *distributed flow graph* (DFG) aims at capturing the different flows of execution that the program can take. According to the distributed model of Section 4, when the processor is released, any task pending at the same location could start executing.

The nodes in the DFG are blocks in $\overline{\mathcal{B}}$ in the analysis of Section 5. The edges represent the control flow in the sequential execution (indicated with normal arrows) and all possible orderings of tasks in the location's queues (indicated with dashed arrows). We only differentiate solid and dashed arrows for presentation purposes, in the computation of the parallel cost, dashed edges are not treated differently from solid edges. We use the MHP analysis results to eliminate those dashed arrows that correspond to infeasible execution orders.

Definition 6.1 (Distributed Flow Graph). Given a program P, its control flow graph $CFG = \langle V_c, E_c \rangle$ with vertices V_c and edges E_c, its block-level cost centers $\overline{\mathcal{B}}$, and its points-to analysis results

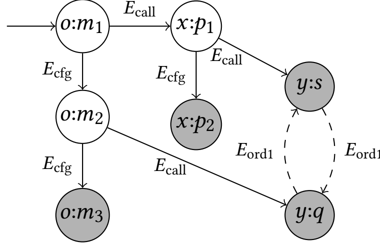


Fig. 3. DFG for Figure 2. Nodes in gray are the nodes in $\overline{\mathcal{B}}_{\text{exit}}$. Edges are labelled with the set of Def 6.1 that produces them.

provided by the function pt, we define the *distributed flow graph* of P as a directed graph $\mathcal{G} = \langle V, E\rangle$ with a set of vertices $V = \overline{\mathcal{B}}$ and a set of edges $E = E_{\text{cfg}} \cup E_{\text{call}} \cup E_{\text{ord1}} \cup E_{\text{sync}} \cup E_{\text{await}} \cup E_{\text{ord2}}$:

$$\begin{array}{rcl}
E_{\text{cfg}} & = & \{o{:}b_1 \longrightarrow o{:}b_2 \mid b_1 \longrightarrow b_2 \in E_c \wedge o{:}b_1 \in \overline{\mathcal{B}} \wedge o{:}b_2 \in \overline{\mathcal{B}}\} \\
E_{\text{call}} & = & \{o_1{:}b_1 \longrightarrow o_2{:}\mathsf{m}_{\mathsf{init}} \mid o_1{:}b_1 \in \overline{\mathcal{B}}_{\text{call}} \wedge pp : \mathsf{x.m()} \in b_1 \wedge o_2 \in pt(pp, \mathsf{x})\} \\
E_{\text{ord1}} & = & \{o{:}b_1 \dashrightarrow o{:}b_2 \mid o{:}b_1 \in \overline{\mathcal{B}}_{\text{exit}} \wedge o{:}b_2 \in \overline{\mathcal{B}}_{\text{init}} \wedge o{:}b_1 \| o{:}b_2\} \\
E_{\text{sync}} & = & \{o_1{:}\mathsf{m}_{\mathsf{exit}} \longrightarrow o_2{:}b_2 \mid \text{either } pp{:}\mathsf{f.get} \text{ or } pp{:}\mathsf{await\ f?} \in b_2 \wedge o_1{:}\mathsf{m}_{\mathsf{exit}} \in awaited(\mathsf{f}, pp)\} \\
E_{\text{await}} & = & \{o{:}b_1 \dashrightarrow o{:}b_2 \mid o{:}b_1 \in \overline{\mathcal{B}}_{\text{await}} \wedge o{:}b_2 \in (\overline{\mathcal{B}}_{\text{await}} \cup \overline{\mathcal{B}}_{\text{init}}) \wedge o{:}b_1 \neq o{:}b_2 \wedge o{:}b_1 \| o{:}b_2\} \\
E_{\text{ord2}} & = & \{o{:}b_1 \dashrightarrow o{:}b_2 \mid o{:}b_1 \in \overline{\mathcal{B}}_{\text{exit}} \wedge o{:}b_2 \in \overline{\mathcal{B}}_{\text{await}} \wedge o{:}b_1 \| o{:}b_2\}
\end{array}$$

Let us start by explaining the edges produced by the sets E_{cfg}, E_{call}, E_{ord1}:

(1) The set E_{cfg} contains the edges that exist in the CFG, but using the points-to information to find out at which abstract locations the blocks are executed.
(2) The set E_{call} associates each block that contains a method invocation with the initial block $\mathsf{m}_{\mathsf{init}}$ of the invoked method. Again, points-to information is used to find out all possible locations from which the calls originate (named o_1 above) and also the locations where the tasks are sent (named o_2 above). Since $pt(pp, \mathsf{x})$ returns a set of abstract locations, arrows are drawn for all possible combinations. Note that the elements $o_1{:}b_1 \in \overline{\mathcal{B}}_{\text{call}}$ are the blocks ending with asynchronous calls and the abstract object that could execute them, which are obtained by means of the function $pt(pp, \mathsf{this})$ in the definition of $\overline{\mathcal{B}}$. These arrows capture the parallelism in the execution and allow us to gain precision in our parallel cost analysis with respect to the serial execution. Intuitively, they enable us to consider the maximal cost of the path that continues the execution in the same location and the path that proceeds with execution of the spawned tasks.
(3) Dashed edges from E_{ord1} are required for expressing the different orderings of the task executions within each abstract location. Without further knowledge, the exit blocks of methods must be joined with the entry blocks of others tasks that execute at the same location. With the MHP analysis we can avoid some dashed edges in the DFG in the following way: given two methods m, whose initial block is m_1, and p, whose final block is p_2, if we know that m_1 cannot happen in parallel with p_2, then we do not need to add a dashed edge between them. This is because the MHP guarantees that when the execution of p finishes there is no instance of block $o{:}m_1$ in the queue of pending tasks. Thus, we do not consider this path in E_{ord1} of the DFG.

Example 6.2. Figure 3 shows the DFG for the program in Figure 2. The nodes are the cost centers in Example 5.5. Nodes in gray are the nodes in $\overline{\mathcal{B}}_{\text{exit}}$, and they indicate that the execution of the

program can terminate by executing $o{:}m_3$, $x{:}p_2$, $y{:}s$ or $y{:}q$. Solid edges include the existing edges in the CFG of the sequential program but combined with the location's identity (E_{cfg}) and those derived from method calls (E_{call}). Since $y{:}s \parallel y{:}q$ (see Example 5.4), the execution order of s and q at location y is unknown (see Section 3). This is modelled by means of the dashed edges (E_{ord1}). In contrast, since $o{:}m_1 \nparallel o{:}m_3$ and $x{:}p_1 \nparallel x{:}p_2$, we neither add a dashed edge from $o{:}m_3$ to $o{:}m_1$ nor from $x{:}p_2$ to $x{:}p_1$. ■

Given a program P, the set of nodes in the *DFG* of P, always contains a selected node from $\overline{\mathcal{B}}_{\text{init}}$ which corresponds to the first block of method main, $o_0{:}\text{main}_{\text{init}}$, where o_0 is the abstract location that represents the initial location, that we refer to as the *initial node* of the graph. It is represented in the examples as a node with an incoming arrow, as for instance node $o{:}m_1$ in Figure 3.

Let us continue with the explanation of the edges produced by E_{sync}, E_{await} and E_{ord2}, which are needed to treat the cooperative concurrency of the program caused by await and get instructions. Handling cooperative concurrency in the analysis is challenging because we need to model the situation where we can lose the processor at await instructions and another pending task can interleave its execution with the current task. Fortunately, task interleavings can be captured in the graph in a clean way by treating the elements in $\overline{\mathcal{B}}_{\text{await}}$ as both initial and ending blocks. Let b be a block which contains a f.get or await f? instruction. Then, *awaited*(f, *pp*) returns the (set of) exit blocks to which the future variable f can be linked at program point *pp*. We use the points-to analysis results to find the set of tasks a future variable might be related to. Furthermore, the MHP analysis obtains information from the await instructions. The execution of the task to which f is linked is finished after an await f? is executed. Thus, this task will not happen in parallel with the next tasks spawned at the same location. Let us see how the DFG deals with the cooperative concurrency:

(4) E_{sync} contains those edges that relate the last block of a method to the corresponding synchronization instruction in the caller method (await and get), indicating that the execution can take this path after the method has completed.
(5) E_{await} considers the elements in $\overline{\mathcal{B}}_{\text{await}}$ as ending blocks from which we can start to execute another interleaved task, and therefore, produce edges from them to any initial block of the same location.
(6) E_{ord2} treats blocks in $\overline{\mathcal{B}}_{\text{await}}$ as initial blocks which can start their execution after another task at the same location finishes or releases the processor. As before, the MHP analysis enables us to discard those edges between blocks that cannot be pending to execute when the processor is released.

Note that sets E_{await} and E_{ord2} contain dashed edges that represent the orderings between parts of tasks split by await instructions, and thus capture the possible interleavings. Observe that such interleavings do not happen when the synchronization is done using get instructions. The following example illustrate how the DFG is produced in the presence of await and get instructions.

Example 6.3. Figure 4 shows an example where the call to method p is synchronized by using either await or get. Method p then calls method q at location o. The synchronization (set E_{sync}) produces a new edge from $x{:}p_2$ to the synchronization point in block $o{:}m_3$. This edge adds a new path to reach $o{:}m_3$ that represents a trace in which the execution of main waits until p is finished. The difference between the use of await and get in main is visible in the edges labelled with $\circledast$, which are only added for await. The edge from $o{:}m_3$ to $o{:}q$ is produced by E_{await}, whereas the one that goes from $o{:}q$ to $o{:}m_3$ is produced by E_{ord2}. These edges capture the traces in which the execution of main waits for the termination of p, and q starts its execution before executing $o{:}m_4$,

```
void main () {
  ...                    // m1
  f = x.p(this);         // m1
  ...                    // m2
  await f? | f.get       // m3
  ...                    // m4
}
void p (Loc o) {
  ...                    // p1
  o.q();                 // p1
  ...                    // p2
}
```

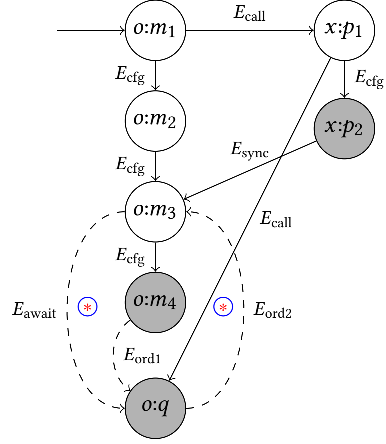


Fig. 4. DFG of a progam with cooperative concurrency

postponing its execution. As this trace is only feasible when the processor is released by an await instruction, we do not add the edges for get instructions. ■

Since we need to capture the parallel execution of the program, the relation between the execution traces and the DFG is not straightforward. In the following, we detail how to relate semantic rules to nodes and edges and then find a way to represent an execution trace by means of a path in the DFG. We introduce the notion of *coverage* to relate the evaluation of semantics rules in a trace to elements of the DFG. We will use $loc \in o$ to represent that loc is abstracted by o, obtained by the points-to analysis.

Definition 6.4 (Coverage). Let t be an execution trace of a program P and $\mathcal{G}$ be the *DFG* of P. Each parallel transition $W_i \in t$ is of the form $W_i \equiv \texttt{Locs}_i\ \texttt{Tsks}_i \rightsquigarrow \texttt{Locs}_{i+1}\ \texttt{Tsks}_{i+1}$, where $\texttt{Locs}_i = \{loc_1, \ldots loc_{m_i}\}$ and $\texttt{Locs}_{i+1} = \{loc'_1, \ldots loc'_{m_{i+1}}\}$. We define the coverage of W_i as follows:

(1) Node $o{:}b$ *covers* W_i *at location* $loc_j \in \texttt{Locs}_i$ if loc_j is of the form $loc(lid, tid)$ and $tsk(tid, lid, m, l, inst; s) \in \texttt{Tsks}_i$, where $inst$ is in block b and $lid \in o$, and performs an evaluation step in transition W_i of any rule executing instruction $inst$.
(2) Node $o{:}b$ *covers* W_i *at location* $loc_j \in \texttt{Locs}_i$ if $loc_j \equiv loc(lid_1, \bot)$ performs an evaluation step by either Select1 or Select2, and $o{:}b$ covers W_{i+1} at location $loc'_k \in \texttt{Locs}_{i+1}$ where $loc'_k \equiv loc(lid_2, tid_2)$ and $lid_1 = lid_2$.
(3) Node $o{:}b$ *covers* W_i if there exists a location $loc_j \in \texttt{Locs}_i$ such that $o{:}b$ covers W_i at loc_j.

We also define the coverage of an execution trace t as follows: A path p in $\mathcal{G}$ *covers* t if for every transition W_i in t there exists a node $o{:}b$ in p such that $o{:}b$ covers W_i.

Observe that the definition of coverage for rules Select1 and Select2 includes the case of the instruction being executed at the next transition in the execution trace. This will allow us to reason about dependencies between tasks at different locations, as well as the sequential execution of different tasks queued at the same location. The following lemma guarantees that any transition in an execution trace is represented in the DFG.

Lemma 6.5. *Given a program P, let t be an execution trace of P and let $\mathcal{G}$ be the DFG of P. For any transition $W_i \in t$ there exists a node in $\mathcal{G}$ that covers W_i.*

Since the transitions of the trace are evaluated in parallel for all distributed locations, there might be several nodes in the DFG that cover the same transition. Furthermore, those nodes will very likely be unconnected in the DFG, as they correspond to the parallel execution of different blocks of code at different locations. Furthermore, we can relate execution traces with paths in the DFG that end in a node in $\overline{\mathcal{B}}_{\mathsf{exit}}$:

Lemma 6.6. *Given a program P, let $\mathcal{G} = \langle V, E\rangle$ be the DFG of P. For any node $o{:}b \in V$, either $o{:}b \in \overline{\mathcal{B}}_{exit}$ or there exists a path p in $\mathcal{G}$ from $o{:}b$ to a node in $\overline{\mathcal{B}}_{exit}$.*

We can always guarantee the following:

Theorem 6.7. *Given a program P, let t be an execution trace of P and $\mathcal{G}$ be the DFG of P. There exists a path p in the graph $\mathcal{G}$ from the initial node to a node in $\overline{\mathcal{B}}_{exit}$ that covers all transitions in t.*

We refer to such paths as *execution paths*. There may be several execution paths for a given execution trace. Note that this theorem includes not only terminating traces, but also non-terminating and deadlock traces. We will see in the next section that we do not need to generate the (possibly infinite) set of execution paths to obtain an accurate upper bound of the parallel cost for any execution trace.

6.2 Inference of Parallel Cost

The next phase in our analysis is about obtaining the maximal parallel cost of all possible execution traces of the program, based on the DFG. We can use Theorem 6.7 to reason about execution traces by means of selected paths in the DFG. We use $paths(\mathcal{G})$ to denote the set of all execution paths that can be obtained from $\mathcal{G}$. Observe that some paths in $paths(\mathcal{G})$ might correspond to infeasible traces.

Definition 6.8 (Set of inferred execution paths). Given a program P and its distributed flow graph $\mathcal{G}$, the *set of inferred execution paths* in $\mathcal{G}$, denoted as $paths(\mathcal{G})$, is defined as

$$paths(\mathcal{G}) = \{p \mid p \text{ is a path in } \mathcal{G} \wedge first_node(p) = o_0{:}\mathsf{main}_{\mathsf{init}} \wedge last_node(p) \in \overline{\mathcal{B}}_{\mathsf{exit}}\}.$$

Example 6.9. Let us compute the set $paths(\mathcal{G})$ for the DFG shown in Figure 3. All paths start from $o{:}m_1$ and finish in a node in $\overline{\mathcal{B}}_{\mathsf{exit}}=\{o{:}m_3, x{:}p_2, y{:}s, y{:}q\}$. Some of these paths are:

$$\begin{array}{l}
o{:}m_1 \rightarrow o{:}m_2 \rightarrow o{:}m_3\\
o{:}m_1 \rightarrow x{:}p_1 \rightarrow x{:}p_2\\
o{:}m_1 \rightarrow x{:}p_1 \rightarrow y{:}s\\
o{:}m_1 \rightarrow x{:}p_1 \rightarrow y{:}s \rightarrow y{:}q\\
o{:}m_1 \rightarrow x{:}p_1 \rightarrow y{:}s \rightarrow y{:}q \rightarrow y{:}s \ldots\\
o{:}m_1 \rightarrow x{:}p_1 \rightarrow y{:}s \rightarrow y{:}q \rightarrow y{:}s \rightarrow y{:}q \ldots\\
\ldots\\
o{:}m_1 \rightarrow o{:}m_2 \rightarrow y{:}q\\
o{:}m_1 \rightarrow o{:}m_2 \rightarrow y{:}q \rightarrow y{:}s \ldots\\
\ldots
\end{array}$$

Note that the cycle between $y{:}s$ and $y{:}q$ produces an infinite number of inferred execution paths. ■

The key idea to obtain the parallel cost from DFG execution paths is that the cost of each block (obtained by using the block-level cost analysis) contains not only the cost of the block itself but this cost is multiplied by the number of times the block is visited (see Section 4). Thus, we use the sets of nodes in the inferred execution paths of the DFG instead of the actual paths since the

multiplicity of the elements is already taken into account in the cost associated to each block. We use *elements*(p) to refer to the set of nodes in a given path p, and to define the set $\mathcal{N}(\mathcal{G})$.

Definition 6.10. Given a program P and its distributed flow graph $\mathcal{G}$, $\mathcal{N}(\mathcal{G})$ is the following set of sets:

$$\mathcal{N}(\mathcal{G}) = \{elements(p) \mid p \in paths(\mathcal{G})\}$$

We define the set $\mathcal{N}^+(\mathcal{G})$ as the set of *maximal* elements of $\mathcal{N}(\mathcal{G})$ with respect to set inclusion, i.e., those sets in $\mathcal{N}(\mathcal{G})$ which are not contained in any other set in $\mathcal{N}(\mathcal{G})$.

The set $\mathcal{N}(\mathcal{G})$ contains the sets of nodes in the paths of $\mathcal{G}$. Although cycles in the graph produce an infinite number of paths, $\mathcal{N}(\mathcal{G})$ is finite. It may nevertheless happen that some elements in $\mathcal{N}(\mathcal{G})$ are subsumed by other elements in $\mathcal{N}(\mathcal{G})$, representing those execution paths whose nodes are fully included in other execution paths. Thus, to obtain an upper bound for the parallel cost, it is sufficient to compute $\mathcal{N}^+(\mathcal{G})$. Let us illustrate this construction with an example.

Example 6.11. For the graph in Figure 3 we have that:

$$\begin{aligned}\mathcal{N}(\mathcal{G}) = \{&\{o{:}m_1, o{:}m_2, o{:}m_3\},\{o{:}m_1, x{:}p_1, x{:}p_2\},\{o{:}m_1, x{:}p_1, y{:}s\},\{o{:}m_1, x{:}p_1, y{:}s, y{:}q\},\{o{:}m_1, o{:}m_2, y{:}q\},\\ &\{o{:}m_1, o{:}m_2, y{:}q, y{:}s\}\}\\ \mathcal{N}^+(\mathcal{G}) = \{&\underbrace{\{o{:}m_1, o{:}m_2, o{:}m_3\}}_{N_1}, \underbrace{\{o{:}m_1, x{:}p_1, x{:}p_2\}}_{N_2}, \underbrace{\{o{:}m_1, x{:}p_1, y{:}s, y{:}q\}}_{N_3}, \underbrace{\{o{:}m_1, o{:}m_2, y{:}s, y{:}q\}}_{N_4}\}\end{aligned}$$

Note that the sets in $\mathcal{N}^+(\mathcal{G})$ represent traces of the program. The execution captured by N_1 corresponds to trace ① in Figure 2. In this trace, the code executed at location o leads to the maximal cost. Similarly, the set N_3 corresponds to trace ② and N_4 corresponds to trace ③. The set N_2 corresponds to a trace where $x{:}p_2$ leads to the maximal cost, which is not shown in Figure 2. ∎

The parallel cost of the distributed system can be over-approximated by the maximum cost for the sets in $\mathcal{N}^+(\mathcal{G})$.

Definition 6.12 (Inferred parallel cost). The *inferred parallel cost* of a program $P(\bar{x})$ with distributed flow graph $\mathcal{G}$, is defined as $\widehat{\mathcal{P}}(P(\bar{x})) = \max_{N \in \mathcal{N}^+(\mathcal{G})} \mathcal{S}(P(\bar{x}))|_N$.

Example 6.13. Given the set $\mathcal{N}^+(\mathcal{G})$ computed in Example 6.11, we have that the parallel cost is obtained by the following expression:

$$\widehat{\mathcal{P}}(n) = max(\mathcal{S}(n)|_{N_1}, \mathcal{S}(n)|_{N_2}, \mathcal{S}(n)|_{N_3}, \mathcal{S}(n)|_{N_4})$$

∎

As the inferred parallel cost considers the maximal sets in $\mathcal{N}^+(\mathcal{G})$, which cover all the execution traces, it computes the worst possible case of executing the associated paths. Thus, we can guarantee that $\widehat{\mathcal{P}}$ is an over-approximation of $\mathcal{P}$.

Theorem 6.14. $\mathcal{P}(P(\bar{x})) \leq \widehat{\mathcal{P}}(P(\bar{x}))$.

Observe that Theorem 6.7 guarantees that there exists a path that covers any trace, and the costs associated to the nodes in the paths are used for computing $\widehat{\mathcal{P}}(P(\bar{x}))$. As the block-level resource analysis based on [12] gives an unbounded cost expression for non-terminating programs, our analysis returns an unbounded result. Note that deadlock traces, which always have a finite parallel cost, are indeed bounded by $\widehat{\mathcal{P}}(P(\bar{x}))$. According to Lemma 6.5, all instructions executing a transition, including the deadlock instructions, are covered by a node in the DFG and, according to

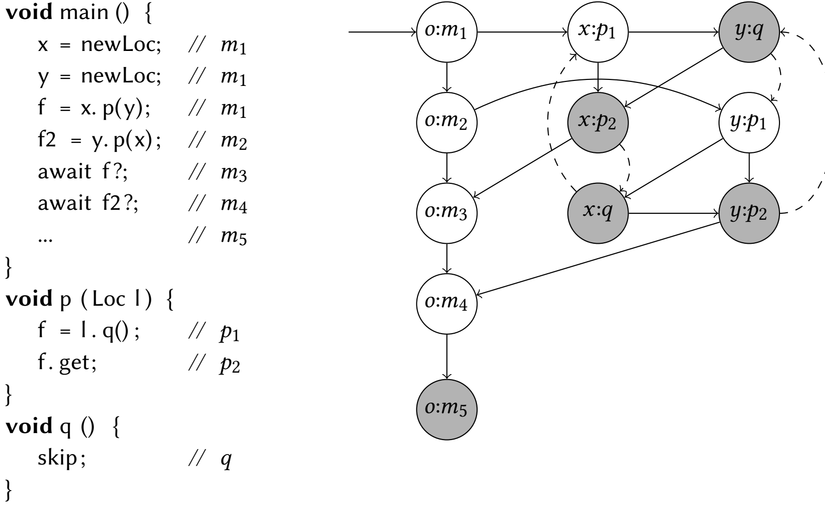


Fig. 5. Parallel cost of a deadlock program

Lemma 6.6, there exists a path from all nodes to some node in $\overline{\mathcal{B}}_{\text{exit}}$. As the parallel cost computes the cost of all possible paths in the DFG, the parallel cost over-approximates the concrete cost of the deadlock traces, as we illustrate with the following example.

Example 6.15. To the left of Figure 5 we show a typical program which produces a deadlock trace. The invocations to method p at locations x and y mutually depend on each other and, as locations are not released at f.get, the execution deadlocks. To the right, Figure 5 shows the DFG of the program. The deadlock trace will be blocked in nodes: $x{:}p_2$ for location x, and, $y{:}p_2$ for location y. The deadlock trace produces the following concrete cost:

$$\mathcal{P} = max(\widehat{m_1} + \widehat{p_1} + \widehat{p_2}, \widehat{m_1} + \widehat{m_2} + \widehat{p_1} + \widehat{p_2})$$

.

As there exists a path in the DFG that traverses all nodes of the DFG, the parallel cost analysis returns an expression the contains the cost of all nodes:

$$\mathcal{N}^+(\mathcal{G}) = \{\underbrace{\{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, o{:}m_5, x{:}p_1, x{:}p_2, x{:}q, y{:}p_1, y{:}p_2, y{:}q\}}_{N_0}\}$$

$$\widehat{\mathcal{P}}() = \mathcal{S}()|_{N_0} = \widehat{m_1} + \widehat{m_2} + \widehat{m_3} + \widehat{m_4} + \widehat{m_5} + \widehat{p_1} + \widehat{p_2} + \widehat{q} + \widehat{p_1} + \widehat{p_2} + \widehat{q}$$

which over-approximates the concrete cost, and therefore $\widehat{\mathcal{P}} \geq \mathcal{P}$.

Although we have obtained the parallel cost of the whole program, we can easily over-approximate the parallel cost of traces that end in a location of interest o, denoted as $\widehat{\mathcal{P}}(P(\bar{x}))|_o$. To do so, we consider only the paths that lead to the exit nodes of this location. Observe that those paths may contain intermediate nodes of any location. In particular, given a location o, we consider the set of paths $paths(\mathcal{G}, o)$ which is the subset of $paths(\mathcal{G})$ ending in an exit node of o. The computation of

the inferred parallel cost ending in a given abstract location o uses

$$paths(\mathcal{G}, o) = \{p \mid p \text{ is a path in } \mathcal{G} \wedge first_node(p) = o_0{:}\mathsf{main}_{\mathsf{init}} \wedge last_node(p) = o{:}m \wedge o{:}m \in \overline{\mathcal{B}}_{\mathsf{exit}}\}$$

instead of $paths(\mathcal{G})$. Analogously, with $paths(\mathcal{G}$,o), we compute $\mathcal{N}^+(\mathcal{G}, o)$ to obtain $\widehat{\mathcal{P}}(P(\bar{x}))|_o$. Let us illustrate it with an example.

Example 6.16. Given the program of Figure 2, we can over-approximate the parallel cost associated to location y by using

$$\mathcal{N}^+(\mathcal{G}, y) = \{\underbrace{\{o{:}m_1, x{:}p_1, y{:}s, y{:}q\}}_{N_3}, \underbrace{\{o{:}m_1, o{:}m_2, y{:}s, y{:}q\}}_{N_4}\}$$

to compute the upper-bound restricted to location y

$$\widehat{\mathcal{P}}(n)|_y = max(\mathcal{S}(n)|_{N_3}, \mathcal{S}(n)|_{N_4})$$

Note that, in the computation of $\widehat{\mathcal{P}}(n)|_y$, we only consider N_3 and N_4 with respect to the computation of $\widehat{\mathcal{P}}(n)$ of Example 6.13, discarding N_1 and N_2. ■

7 COMPUTATION OF PARALLEL COST

In this section we present three different techniques, with different levels of precision, for computing the set $\mathcal{N}^+$ and consequently, different levels of precision for the computation of the inferred parallel cost. We start the section by showing the notion of path expressions [48], which are regular expressions that represent all paths between two nodes and are used in the three techniques. After describing such techniques, we additionally include a discussion about how the analysis of the context-sensitive version of the program could provide significant precision improvements in the computation of the inferred parallel cost.

7.1 Path expressions

The first step for the inference is to compute $\mathcal{N}^+(\mathcal{G})$ by solving the so-called *single-source path expression problem* [48], which looks for a regular expression (named *path expression*) representing all paths from an initial node n_1 and an ending node n_2. Given a DFG and the initial node of the program $o_0{:}\mathsf{main}_{\mathsf{init}}$, we compute all path expressions starting from $o_0{:}\mathsf{main}_{\mathsf{init}}$ and ending in a node $o_2{:}\mathsf{m}_{\mathsf{exit}} \in \overline{\mathcal{B}}_{\mathsf{exit}}$. Given a DFG $\mathcal{G}$, we denote by $pexpr(\mathcal{G})$ the set of path expressions obtained from the initial node to all nodes in $\overline{\mathcal{B}}_{\mathsf{exit}}$.

Example 7.1. To compute the set $pexpr(\mathcal{G})$ for the graph in Figure 3, we compute the path expressions starting from $o{:}m_1$ and finishing in exit nodes in $\overline{\mathcal{B}}_{\mathsf{exit}}$. In path expressions, we use $o{:}m_1{\cdot}o{:}m_2$ to represent the edge from $o{:}m_1$ to $o{:}m_2$. Thus, for the nodes in $\overline{\mathcal{B}}_{\mathsf{exit}}$ we have

$$\begin{array}{rcl} \mathsf{e}_{o:m_3} & = & o{:}m_1{\cdot}o{:}m_2{\cdot}o{:}m_3 \\ \mathsf{e}_{x:p_2} & = & o{:}m_1{\cdot}x{:}p_1{\cdot}x{:}p_2 \\ \mathsf{e}_{y:s} & = & o{:}m_1{\cdot}(x{:}p_1{\cdot}y{:}s \mid o{:}m_2{\cdot}y{:}q{\cdot}y{:}s){\cdot}(y{:}q{\cdot}y{:}s)^* \\ \mathsf{e}_{y:q} & = & o{:}m_1{\cdot}(x{:}p_1{\cdot}y{:}s{\cdot}y{:}q \mid o{:}m_2{\cdot}y{:}q){\cdot}(y{:}s{\cdot}y{:}q)^* \end{array}$$

Thus, we have that $pexpr(\mathcal{G}) = \{\mathsf{e}_{o:m_3}, \mathsf{e}_{x:p_2}, \mathsf{e}_{y:s}, \mathsf{e}_{y:q}\}$. ■

Note that the path expressions contain disjunctions, expressed by |, and multiplicity, expressed by $(_)^*$. By exploiting the path expressions we can set different levels of precision, resulting in different over-approximations of $\mathcal{N}^+(\mathcal{G})$. In order to compute $\mathcal{N}^+(\mathcal{G})$ from the path expressions, we overload the definition of $\mathcal{N}^+$ by adding the ending node as a parameter: given an ending node $o{:}b$, which produces, $\mathsf{e}_{o:b}$, we use $\mathcal{N}^+(\mathcal{G}, o{:}b)$ to denote the set of maximal elements of the

subset of $\mathcal{N}(\mathcal{G})$ that are produced by the path expression $\mathsf{e}_{o:b}$. Then, $\mathcal{N}^+(\mathcal{G})$ can be obtained by $\bigcup^{max}_{o:b_i \in \overline{\mathcal{B}}_{\text{exit}}} \mathcal{N}^+(\mathcal{G}, o{:}b_i)$. The operation $\mathcal{N}_1^+ \cup^{max} \mathcal{N}_2^+$, where $\mathcal{N}_1^+$ and $\mathcal{N}_2^+$ are sets of sets, first applies $\mathcal{N}^+ = \mathcal{N}_1^+ \cup \mathcal{N}_2^+$, then removes those sets in $\mathcal{N}^+$ that are contained in another set in $\mathcal{N}^+$.

In the following subsections we present several over-approximations with different levels of precision for computing $\mathcal{N}^+(\mathcal{G})$.

7.1.1 Naïve approach. As a straightforward but naïve approach, for each node in $o{:}b \in \overline{\mathcal{B}}_{\text{exit}}$, which produces $\mathsf{e}_{o:b}$ in *pexpr*($\mathcal{G}$), we compute the set of maximal elements collecting all nodes involved in the path expression, denoted as $\mathcal{N}_0^+(\mathcal{G}, o{:}b)$.

Example 7.2. Given the path expressions from Example 7.1, we compute the following sets with the naïve approach:

$$\begin{array}{rcl} \mathcal{N}_0^+(\mathcal{G}, o{:}m_3) & = & \{\underbrace{\{o{:}m_1, o{:}m_2, o{:}m_3\}}_{N_1^0}\} \\ \mathcal{N}_0^+(\mathcal{G}, x{:}p_2) & = & \{\underbrace{\{o{:}m_1, x{:}p_1, x{:}p_2\}}_{N_2^0}\} \\ \mathcal{N}_0^+(\mathcal{G}, y{:}s) \;=\; \mathcal{N}_0^+(\mathcal{G}, y{:}q) & = & \{\underbrace{\{o{:}m_1, o{:}m_2, x{:}p_1, y{:}s, y{:}q\}}_{N_3^0}\} \end{array}$$

Observe that the naïve approach ignores the case where $o{:}m_2$ and $x{:}p_1$ can be executed in parallel on different objects, namely o and x, and includes both nodes in the set N_3^0. Consequently, the cost will be calculated as if both nodes were executed sequentially. Thus, we have $\mathcal{N}_0^+(\mathcal{G}) = \{N_1^0, N_2^0, N_3^0\}$. ∎

We can now compute the cost associated to each $N \in \mathcal{N}^+(\mathcal{G})$ over-approximated by $\mathcal{N}_0^+(\mathcal{G})$ using the block-level cost analysis, that is, $\mathcal{S}(P(\bar{x}))|_N$, as it is illustrated in the following example.

Example 7.3. The cost is obtained by using the block-level costs for all nodes that compose the sets in *paths*. With $\mathcal{N}_0^+(\mathcal{G})$, the overall parallel cost is:

$$\widehat{\mathcal{P}}(n) = max(\mathcal{S}(n)|_{N_1^0}, \mathcal{S}(n)|_{N_2^0}, \mathcal{S}(n)|_{N_3^0})$$

Importantly, observe that $\widehat{\mathcal{P}}$ is more precise than the serial cost because all paths have at least one missing node. For instance, N_1^0 does not contain the cost of $x{:}p_1$, $x{:}p_2$, $y{:}s$, $y{:}q$, or N_3^0 does not contain the cost of $o{:}m_3$, $x{:}p_2$. ∎

7.1.2 Unfolding disjunctions. The set $\mathcal{N}^+(\mathcal{G})$ can be over-approximated by unfolding the disjunctions of path expressions in *pexpr*($\mathcal{G}$) into different elements in the usual way, and by adding the nodes within the iterative subexpressions once. We refer to this set as $\mathcal{N}_e^+$.

Example 7.4. Let us detail the computation of $\mathsf{e}_{y:s}$ computed in Example 7.1. We first add the iterative elements only once:

$$o{:}m_1{\cdot}(x{:}p_1{\cdot}y{:}s \mid o{:}m_2{\cdot}y{:}q{\cdot}y{:}s){\cdot}y{:}q{\cdot}y{:}s$$

and then, we unfold the disjunctions, producing two different paths and consequently two different sets of nodes:

$$\begin{array}{rcl} o{:}m_1{\cdot}x{:}p_1{\cdot}y{:}s{\cdot}y{:}q{\cdot}y{:}s & \Rightarrow & \{o{:}m_1, x{:}p_1, y{:}s, y{:}q\} \\ o{:}m_1{\cdot}o{:}m_2{\cdot}y{:}q{\cdot}y{:}s{\cdot}y{:}q{\cdot}y{:}s & \Rightarrow & \{o{:}m_1, o{:}m_2, y{:}q, y{:}s\} \end{array}$$

Thus, given the path expressions in Example 7.1, we compute the following sets:

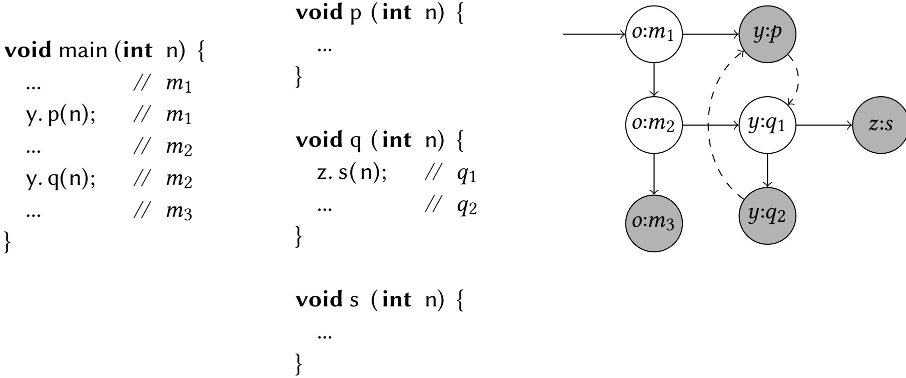


Fig. 6. Filtering paths example

$$
\begin{array}{rcl}
\mathcal{N}_e^+(\mathcal{G}, o{:}m_3) & = & \{\underbrace{\{o{:}m_1, o{:}m_2, o{:}m_3\}}_{N_1^e}\} \\
\mathcal{N}_e^+(\mathcal{G}, x{:}p_2) & = & \{\underbrace{\{o{:}m_1, x{:}p_1, x{:}p_2\}}_{N_2^e}\} \\
\mathcal{N}_e^+(\mathcal{G}, y{:}s) = \mathcal{N}_e^+(\mathcal{G}, y{:}q) & = & \{\underbrace{\{o{:}m_1, x{:}p_1, y{:}s, y{:}q\}}_{N_3^e}, \underbrace{\{o{:}m_1, o{:}m_2, y{:}s, y{:}q\}}_{N_4^e}\}
\end{array}
$$

Observe that these sets represent traces of the program. The execution captured by N_1^e corresponds to trace ① of Figure 2. In this trace, the code executed at location o leads to the maximal cost. Similarly, the set N_3^e corresponds to trace ② and N_4^e corresponds to trace ③. The set N_2^e corresponds to a trace where $x{:}p_2$ leads to the maximal cost (not shown in Figure 2). Therefore, given $\mathcal{N}_e^+(\mathcal{G}) = \{N_1^e, N_2^e, N_3^e, N_4^e\}$, the overall parallel cost is:

$$\widehat{\mathcal{P}}(n) = max(\mathcal{S}(n)|_{N_1^e}, \mathcal{S}(n)|_{N_2^e}, \mathcal{S}(n)|_{N_3^e}, \mathcal{S}(n)|_{N_4^e})$$

Importantly, using $\mathcal{N}_e^+(\mathcal{G})$ instead of $\mathcal{N}_0^+(\mathcal{G})$, $\widehat{\mathcal{P}}$ gets more precise results as the set N_3^0 is split into N_3^e and N_4^e, containing less nodes than N_3^0. Additionally, for instance we can restrict the parallel cost to location o by restricting the upper bound to o. As $o{:}m_3$ is the only final node for location o, we have that $\widehat{\mathcal{P}}(n)|_o = \mathcal{S}(n)|_{N_1^e}$. Similarly, for location y we have two exit nodes, $y{:}s$ and $y{:}q$, thus $\widehat{\mathcal{P}}(n)|_y = max(\mathcal{S}(n)|_{N_3^e}, \mathcal{S}(n)|_{N_4^e})$. ■

Recall that when there are several calls to a block $o{:}b$, the graph contains only one node $o{:}b$ but the serial cost $\mathcal{S}(P(\bar{x}))|_{o:b}$ accumulates the cost of all calls. This is also the case for loops or recursion. The nodes within an iterative construct form a cycle in the DFG and by setting the corresponding cost center to 1, the serial cost accumulates the cost of all executions of such nodes.

7.1.3 Using $\mathcal{S}^t(P(\bar{x}))$ to gain further accuracy. By means of the path expressions, we can compute all possible paths from the initial node to any ending node in the DFG. However, some of these computed paths might be infeasible in actual executions and thus lead to pessimistic over-approximations of the parallel cost. In the presence of cycles, caused either by loops or by dashed

edges, nodes can appear multiple times in an execution path. Let us illustrate this issue with an example.

Example 7.5. Figure 6 shows a program and its corresponding $\mathcal{G}$. For this program, we compute the following sets:

$$\begin{array}{rcccl} \mathcal{N}_0^+(\mathcal{G}, o{:}m_3) & = & \mathcal{N}_e^+(\mathcal{G}, o{:}m_3) & = & \{\underbrace{\{o{:}m_1, o{:}m_2, o{:}m_3\}}_{E_1}\} \\ \mathcal{N}_0^+(\mathcal{G}, z{:}s) & = & \mathcal{N}_e^+(\mathcal{G}, z{:}s) & = & \{\underbrace{\{o{:}m_1, o{:}m_2, y{:}q_1, y{:}q_2, y{:}p, z{:}s\}}_{E_2}\} \\ \mathcal{N}_0^+(\mathcal{G}, y{:}p) & = & \mathcal{N}_e^+(\mathcal{G}, y{:}p) & = & \{\{o{:}m_1, o{:}m_2, y{:}q_1, y{:}q_2, y{:}p\}\} \end{array}$$

It can be seen that we have two sets to consider, the one in $\mathcal{N}_e^+(\mathcal{G}, o{:}m_3)$ and the one in $\mathcal{N}_e^+(\mathcal{G}, z{:}s)$. The set in $\mathcal{N}_e^+(\mathcal{G}, y{:}p)$ is subsumed by the set of $\mathcal{N}_e^+(\mathcal{G}, z{:}s)$. With these sets, we get the upper bound:

$$\widehat{\mathcal{P}}(n) = max(\mathcal{S}(n)|_{E_1}, \mathcal{S}(n)|_{E_2})$$

Note that the set E_2 comes from the path $o{:}m_1{\cdot}o{:}m_2{\cdot}y{:}q_1{\cdot}y{:}q_2{\cdot}y{:}p{\cdot}y{:}q_1{\cdot}z{:}s$, which visits $y{:}q_1$ twice. As $y{:}q_1$ is only executed once in the program, this is indeed an infeasible execution path of the program, which leads to a less precise upper-bound. ■

As we have seen in Theorem 6.7, a path in $\mathcal{G}$ might not represent actual executions in the semantics of the program. When we have a node that is executed at most once in the concrete semantics for all possible execution traces, the paths in $\mathcal{G}$ that include this node more than once represent infeasible execution traces. Consequently, we do not have to consider them in the inference of the parallel cost. We refer to nodes that are only executed once as *singleton* nodes. We define $singleton(P(\bar{x}))$ as the set of blocks that can be executed only once and we use $\mathcal{S}^t(P(\bar{x}))$ to compute such set:

Definition 7.6 (Singleton blocks). Given a program $P(\bar{x})$ and its corresponding set of blocks $\overline{\mathcal{B}}$, we define

$$singleton(P(\bar{x})) = \{o{:}b \mid o{:}b \in \overline{\mathcal{B}} \wedge \mathcal{S}^t(P(\bar{x}))|_{o:b} = 1\}$$

Example 7.7. Given the program shown in Figure 6, we get the following upper bound of the number of times each block is executed:

$$\mathcal{S}^t(n) = c(o{:}m_1) + c(o{:}m_2) + c(o{:}m_3) + c(y{:}p) + c(y{:}q_1) + c(y{:}q_2) + c(z{:}s)$$

As $\mathcal{S}^t(n)|_{z:b} = 1$ for all $z{:}b \in \overline{\mathcal{B}}$, it captures that all blocks are executed only once. Thus, we have that $singleton(P(\bar{x}))$ contains all nodes in the DFG. For the program in Figure 6, $singleton(P(\bar{x})) = \{o{:}m_1, o{:}m_2, o{:}m_3, y{:}p, y{:}q_1, y{:}q_2, z{:}s\}$. ■

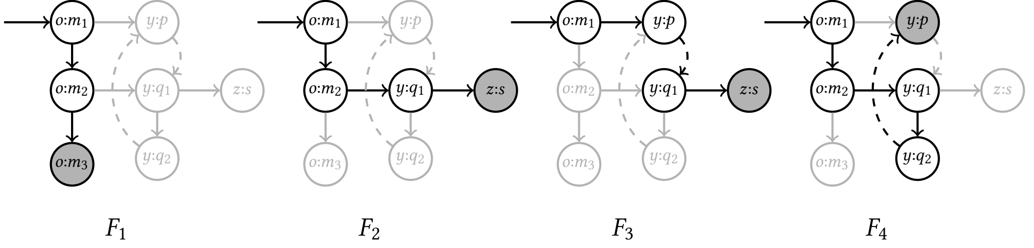
The next step involves exploiting the fact that paths containing the elements in $singleton(P(\bar{x}))$ more than once are infeasible, and consequently, we do not have to consider them in the computation of $\mathcal{N}^+(\mathcal{G})$. We refer to this over-approximation of $\mathcal{N}^+(\mathcal{G})$ as $\mathcal{N}_{\mathcal{F}}^+(\mathcal{G})$. Let us illustrate this issue with the example in Figure 6:

Example 7.8. Given the set $singleton(P(\bar{x}))$ computed at Example 7.7, we can discard those paths containing any singleton node more than once. For instance, the path $o{:}m_1{\cdot}o{:}m_2{\cdot}y{:}q_1{\cdot}y{:}q_2{\cdot}y{:}p{\cdot}y{:}q_1{\cdot}z{:}s$, which leads to the set $\mathcal{N}_e^+(\mathcal{G}, z{:}s)$, as seen in Example 7.5, is not feasible. We should not consider this path in the computation of $\mathcal{N}$. Using $singleton(P(\bar{x}))$, as all nodes must appear at most once,

we can ignore those paths that contain repeated nodes. With this improvement, we get:

$$\begin{aligned}
\mathcal{N}^+_{\mathcal{F}}(\mathcal{G}, o{:}m_3) &= \{\underbrace{\{o{:}m_1, o{:}m_2, o{:}m_3\}}_{F_1}\} \\
\mathcal{N}^+_{\mathcal{F}}(\mathcal{G}, z{:}s) &= \{\underbrace{\{o{:}m_1, o{:}m_2, y{:}q_1, z{:}s\}}_{F_2}, \underbrace{\{o{:}m_1, y{:}p, y{:}q_1, z{:}s\}}_{F_3}\} \\
\mathcal{N}^+_{\mathcal{F}}(\mathcal{G}, y{:}p) &= \{\underbrace{\{o{:}m_1, o{:}m_2, y{:}q_1, y{:}q_2, y{:}p\}}_{F_4}\} \\
\mathcal{N}^+_{\mathcal{F}}(\mathcal{G}, y{:}q_2) &= \{\{o{:}m_1, y{:}p, y{:}q_1, y{:}q_2\}, \{o{:}m_1, o{:}m_2, y{:}q_1, y{:}q_2\}\}
\end{aligned}$$

These sets are obtained by means of the following paths computed from their corresponding path expression.



Note that $\mathcal{N}^+_{\mathcal{F}}(\mathcal{G}, y{:}q_2)$ does not lead to maximal sets as they are subsumed by $\mathcal{N}^+_{\mathcal{F}}(\mathcal{G}, y{:}p)$. Thus, by filtering infeasible paths, instead of $\mathcal{N}^+_e(\mathcal{G}) = \{E_1, E_2\}$ (see Example 7.5), we get $\mathcal{N}^+_{\mathcal{F}}(\mathcal{G}) = \{F_1, F_2, F_3, F_4\}$. From E_2, which contains all nodes except $o{:}m_3$, now we get three different sets: F_1, F_2, F_3, which have less nodes than E_2. This improvement leads to a more precise parallel cost:

$$\widehat{\mathcal{P}}(n) = max(\mathcal{S}(n)|_{F_1}, \mathcal{S}(n)|_{F_2}, \mathcal{S}(n)|_{F_3}, \mathcal{S}(n)|_{F_4})$$

■

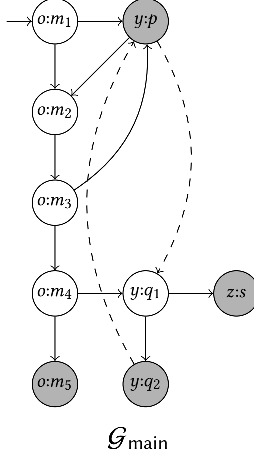
In order to filter those paths that are infeasible according to the nodes contained in $singleton(P(\bar{x}))$, we replace each multiplicity element of the form $(exp)^*$ found in the path expression by the disjunction $(exp|nil)$, that is, we remove the multiplicity and include a disjunction such that the nodes appear, that is, exp, or not appear, nil. As before, the computation of the maximal paths is performed by unfolding all disjunctions of the resulting path expressions.

7.2 Context sensitivity

Remark that the presented work is parametric in the underlying points-to, may-happen in parallel, and cost analyses for serial execution. Hence, any accuracy improvement in these auxiliary analyses will have an impact on the precision of our analysis. We can use standard techniques for context-sensitive analysis [50], e.g., method replication. For instance, a context-sensitive points-to analysis [45] can lead to big accuracy gains. Context-sensitive analyses use the program point at which tasks are spawned as context information. This means that two different calls $o.m$, one from program point p_1 and another from p_2 (where $p_1 \neq p_2$) are distinguished in the analysis as $o{:}m_{p_1}$ and $o{:}m_{p_2}$. Therefore, instead of representing them by a single node in the graph, we will use two different nodes. The advantage of having this finer-grained information is that the analysis regarding task parallelism can be more accurate. For instance, the context-sensitivity allows us to have one path

Program P

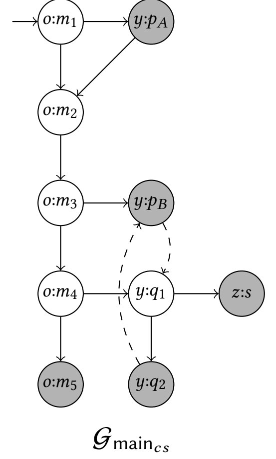
```
main (int n) {
  ...            // m1
  f = y.p(n);    // m1
  await f?;      // m2
  ...            // m3
  y.p(n);        // m3
  ...            // m4
  y.q(n);        // m4
  ...            // m5
}
void p (int n) {
  . . .
}
void q (int n) {
  z.s(n);        // q1
  ...            // q2
}
void s (int n) {
  . . .
}
```



$\mathcal{G}_{\mathsf{main}}$

Program P_{cs}

```
    void main_cs (int n) {
      ...            // m1
Ⓐ    f = y.p_A(n);  // m1
      await f?;      // m2
                     // m3
Ⓑ    y.p_B(n);      // m3
      ...            // m4
      y.q(n);        // m4
      ...            // m5
    }
⊕   void p_A (int n) {
⊕     . . .
⊕   }
⊕   void p_B (int n) {
⊕     . . .
⊕   }
    void q (int n) {
      z.s(n);        // q1
      ...            // q2
    }
    void s (int n) {
      . . .
    }
```



$\mathcal{G}_{\mathsf{main}_{cs}}$

Fig. 7. Context-sensitive example

in the graph which includes a single execution of $o{:}m_{p_1}$ (and none of $o{:}m_{p_2}$). However, if these two nodes are merged into a single one $o{:}m$, we have to consider that both instances of the method are executed. There are also techniques to gain precision in points-to analysis in the presence of loops [46] that could improve the precision of the parallel cost analysis.

The following example aims at illustrating the improvements we can get by analyzing the context-sensitive version of the program.

Example 7.9. Figure 7 (left) shows a program P that invokes method p twice (at location y) and its corresponding DFG. Figure 7 (right) shows its context-sensitive version P_{cs} and its corresponding DFG. The differences between the programs are marked with ⊕, Ⓐ and Ⓑ. As p is called from two different program points, we replicate its code and we get p_A and p_B. The other difference is in main_{cs}, where we replace the calls to p at program points Ⓐ and Ⓑ by two different calls to p_A and p_B, respectively. These changes produce some relevant improvements in the computation of the parallel cost:

(1) Observe that in the DFG of main_{cs} in Figure 7, we have two nodes, $y{:}p_A$ and $y{:}p_B$ instead of $y{:}p$. Additionally, the MHP results allow us to avoid dashed edges between $y{:}p_A$ and $y{:}p_B$ and dashed edges between $y{:}p_A$ and $y{:}q_1$.

(2) The serial upper bound can now compute more precise cost centers as we have two different cost centers for the two calls to method p:

$$\mathcal{S}(P(n)) = c(o{:}m_1)*\widehat{m_1} + c(o{:}m_2)*\widehat{m_2} + c(o{:}m_3)*\widehat{m_3} + c(o{:}m_4)*\widehat{m_4} + c(o{:}m_5)*\widehat{m_5} + \underline{c(y{:}p)*\widehat{p} + c(y{:}p)*\widehat{p}} + c(y{:}q_1)*\widehat{q_1} + c(y{:}q_2)*\widehat{q_2} + c(z{:}s)*\widehat{s}$$

$$\mathcal{S}(P_{cs}(n)) = c(o{:}m_1)*\widehat{m_1} + c(o{:}m_2)*\widehat{m_2} + c(o{:}m_3)*\widehat{m_3} + c(o{:}m_4)*\widehat{m_4} + c(o{:}m_5)*\widehat{m_5} + \underline{c(y{:}p_A)*\widehat{p_A} + c(y{:}p_B)*\widehat{p_B}} + c(y{:}q_1)*\widehat{q_1} + c(y{:}q_2)*\widehat{q_2} + c(z{:}s)*\widehat{s}$$

Observe that in $\mathcal{S}(P_{cs}(n))$ we have $c(y{:}p_A)$ and $c(y{:}p_B)$. Thus, we can separate the cost of both executions of p, whereas in $\mathcal{S}(P(n))$ it is not possible.

(3) The MHP analysis gets more precise results. The analysis of method main returns that $y{:}p \Vert y{:}q_1$ and $y{:}p \Vert y{:}q_2$, that is, both executions of p might be executing in parallel with q. However, the first execution of p is finished when q is invoked as it is captured by the MHP analysis of P_{cs}: $y{:}p_A \not\Vert y{:}q_1$, $y{:}p_A \not\Vert y{:}q_2$, which means that the first call to p and q will not happen in parallel. On the contrary, as the second call to p is not awaited, we have $y{:}p_B \Vert y{:}q_1$ and $y{:}p_B \Vert y{:}q_2$. Additionally, we have $y{:}p_A \not\Vert y{:}p_B$.

(4) By means of $\mathcal{S}^t(P_{cs}(n))$ we get that all blocks in main$_{cs}$ are singleton blocks, that is:

$$singleton(P_{cs}(n)) = \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, o{:}m_5, y{:}p_A, y{:}p_B, y{:}q_1, y{:}q_2, z{:}s\}$$

while in the original program, block $y{:}p$ is not a singleton block as it is executed twice.

$$singleton(P(n)) = \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, o{:}m_5, y{:}q_1, y{:}q_2, z{:}s\}$$

■

The improvement in precision gained by using context-sensitivity leads to a relevant improvement in the computation of the set $\mathcal{N}^+$, and consequently, a more accurate over-approximation of the parallel cost. Let us illustrate such improvements with an example.

Example 7.10. The following table shows the sets $\mathcal{N}_e^+(\mathcal{G}_{\mathsf{main}})$ and $\mathcal{N}_{\mathcal{F}}^+(\mathcal{G}_{\mathsf{main}_{cs}})$ for the original program of Figure 7 and for its context-sensitive version.

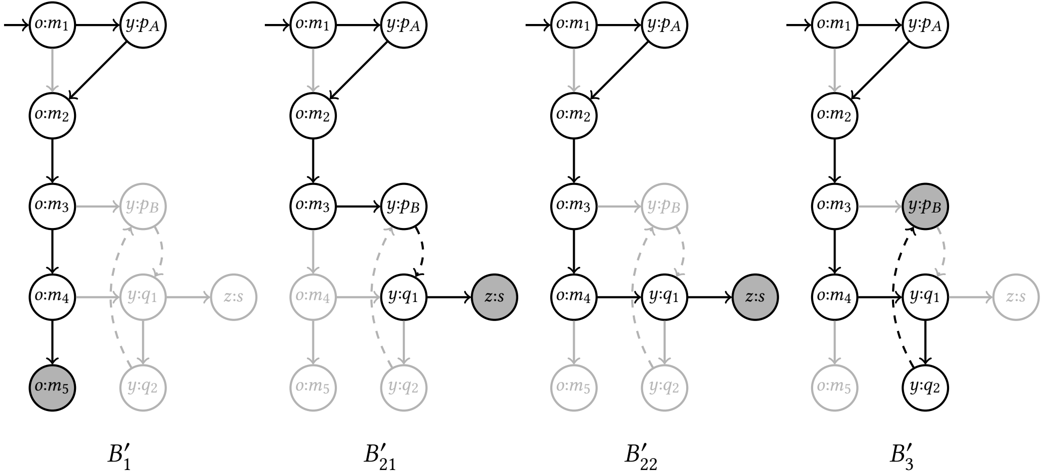
Original program (main)

$$\begin{array}{lcll}
\mathcal{N}_e^+(\mathcal{G}_{\mathsf{main}}) & = \{ & \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, o{:}m_5, y{:}p, y{:}q_1, y{:}q_2\}, & A_1 \\
 & & \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, y{:}p, y{:}q_1, y{:}q_2, z{:}s\} \quad \} & A_2 \\
\mathcal{N}_{\mathcal{F}}^+(\mathcal{G}_{\mathsf{main}}) & = \{ & \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, o{:}m_5, y{:}p\}, & B_1 \\
 & & \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, y{:}p, y{:}q_1, z{:}s\} & B_2 \\
 & & \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, y{:}p, y{:}q_1, y{:}q_2\} \quad \} & B_3
\end{array}$$

Context-sensitive version (main$_{cs}$)

$$\begin{array}{lcll}
\mathcal{N}_e^+(\mathcal{G}_{\mathsf{main}_{cs}}) & = \{ & \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, o{:}m_5, y{:}p_A\}, & A'_1 \\
 & & \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, y{:}p_A, y{:}p_B, y{:}q_1, y{:}q_2, z{:}s\} \quad \} & A'_2 \\
\mathcal{N}_{\mathcal{F}}^+(\mathcal{G}_{\mathsf{main}_{cs}}) & = \{ & \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, o{:}m_5, y{:}p_A\}, & B'_1 \\
 & & \{o{:}m_1, o{:}m_2, o{:}m_3, y{:}p_A, y{:}p_B, y{:}q_1, z{:}s\} & B'_{21} \\
 & & \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, y{:}p_A, y{:}q_1, z{:}s\} & B'_{22} \\
 & & \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}m_4, y{:}p_A, y{:}q_1, y{:}q_2, y{:}p_B\} \quad \} & B'_3
\end{array}$$

Let us graphically show the paths that produce the sets obtained for $\mathcal{N}_{\mathcal{F}}^+(\mathcal{G}_{\mathsf{main}_{cs}})$.



The most relevant differences between the results of the original program and its context-sensitive version are:

(1) For the original program, $\mathcal{N}_e^+(\mathcal{G}_{\text{main}})$ only contains two sets, A_1 and A_2 which contain all nodes except one missing node, $z{:}s$ for A_1 and $o{:}m_5$ for A_2. For the context-sensitive version, $\mathcal{N}_e^+(\mathcal{G}_{\text{main}_{cs}})$ also contains two sets, but one of them has less nodes: A'_1 does not contain $y{:}q_1$ and $y{:}q_2$. Note that there is no difference between A_2 and A'_2 apart from the different nodes produced by context sensitivity.

(2) The analysis of the context-sensitive version of the program allows us to capture four different sets in $\mathcal{N}_{\mathcal{F}}^+(\mathcal{G}_{\text{main}_{cs}})$. The set B_2 is split into two sets, B_{21} and B_{22} in $\mathcal{N}_{\mathcal{F}}^+(\mathcal{G}_{\text{main}_{cs}})$. Note that one of them contains $o{:}m_4$ and the other one contains $y{:}p_B$, but both are not together in the same set.

(3) Another relevant issue is the cost accounted for those methods that are replicated in the context-sensitive version. Observe that B_1 and A'_1 contain the same nodes except $y{:}p$, which is $y{:}p_A$ in A'_1. This change implies that the cost of executing p, that is $\widehat{p}$, is added twice when we use $\mathcal{N}_{\mathcal{F}}^+(\mathcal{G}_{\text{main}})$ but only once if we use $\mathcal{N}_e^+(\mathcal{G}_{\text{main}_{cs}})$.

■

8 EXPERIMENTAL EVALUATION

We have implemented our analysis in SACO [6] and it can be accessed online at http://costa.fdi.ucm.es/parallelcost. We have applied it to some typical distributed systems: BBuffer, which is the typical bounded-buffer for communicating between several producers and consumers; MailServer, which models a distributed mail server system with multiple clients; Chat, which models a chat application; DistHT, which implements and uses a distributed hash table; BookShop, which models a web shop client-server application; and PeerToPeer, which represents a peer-to-peer network formed by a set of interconnected peers. Experiments have been performed on an Intel(R) Core(TM) i7-7700K CPU@4.20GHz with 64GB of RAM, running Ubuntu Server 16.04. Table 1 shows global information obtained for each benchmark and its corresponding context-sensitive version. The columns Benchmark and loc show the name and the number of program lines for each benchmark, respectively. The columns $\#_{\mathsf{N}}$ and $\#_{\mathsf{E}}$ state the number of nodes and edges of the DFG (Definition 6.1). The columns $\#_{\mathsf{F}}$ contain the number of terminal nodes in the DFG. Our experimental evaluation

			Original Program				Context-sensitive			
Benchmark	loc	$\#_\mathsf{I}$	$\#_\mathsf{N}$	$\#_\mathsf{E}$	$\#_\mathsf{F}$	$\mathbf{T}_\mathcal{S}$	$\#_\mathsf{N}$	$\#_\mathsf{E}$	$\#_\mathsf{F}$	$\mathbf{T}_\mathcal{S}$
BBuffer	105	1000	37	50	6	148	47	102	6	201
MailServer	115	1000	28	35	6	436	42	57	9	552
DistHT	353	1000	38	47	6	493	101	174	15	1798
Chat	302	100	133	437	14	1055	215	722	24	6323
BookShop	423	10000	63	67	7	1077	219	342	15	5794
PeerToPeer	240	100	154	465	10	52258	443	2534	11	151675

Table 1. Experimental results (times in ms)

aims at comparing the precision of the upper bound expressions obtained by the different versions of the parallel cost analysis described in the paper. For this purpose, we evaluate the upper bound for all possible combinations of the input arguments $\bar{e}$ ranging each of them between [1-10] and compare the obtained results.

The number of evaluations performed for each benchmark is shown in column $\#_\mathsf{I}$. Finally, column $\mathbf{T}_\mathcal{S}$ shows the execution times for the serial cost analysis.

The information in Table 1 indicates that the execution times of the serial cost analysis are acceptable, even for the context-sensitive version of the programs. PeerToPeer is the benchmark that presents the worst execution time, taking more than 50 seconds to analyze the original problem and 150 seconds for the context-sensitive version of the program. Observe that the size of the graphs increases significantly when analyzing the context-sensitive version of the programs. Again, PeerToPeer presents the most complex graph, with 443 nodes and 2534 edges.

As described in Section 7, once the path expressions have been computed, we can compute the set $\mathcal{N}^+(\mathcal{G})$ with different levels of precision: (a) the *naïve approach*, using the set of nodes contained in the path expressions, (b) *unfolding all disjunctions* of the path expression; and (c) *filtering infeasible paths* by using $\mathcal{S}^t(\bar{x})$. Table 2 shows the results of the comparison between these three precision levels of upper bounds with respect to the serial cost of the program. The superscripts a, b, and c in the column headers indicate the level of precision used. The column $\mathbf{T}_{\widehat{\mathcal{P}}}$ shows the execution time required by the parallel cost analysis (in milliseconds) to build the DFG graphs, to compute the path expressions and to derive the set $\mathcal{N}^+(\mathcal{G})$ from the path expressions. The computation of the parallel cost includes a simplification of the DFG to reduce the strongly connected components (SCC) to one node. Such a simplification significantly reduces the time needed in computing the path expressions and $\mathcal{N}^+(\mathcal{G})$. The columns $\#_\mathsf{P}$ show the number of elements contained in the set $\mathcal{N}^+(\mathcal{G})$. The columns $\%_\mathsf{m}$, $\%_\mathsf{a}$ and $\%_{\widehat{\mathcal{P}}}$ show the ratio of different measures with respect to the serial cost. Columns $\%_{\widehat{\mathcal{P}}}$ compare the parallel cost $\widehat{\mathcal{P}}(\bar{e})$ to the serial cost $\mathcal{S}(\bar{e})$ by evaluating $\widehat{\mathcal{P}}(\bar{e})/\mathcal{S}(\bar{e})*100$ for different input values for $\bar{e}$, $\%_\mathsf{m}$ show the ratio obtained for the location that achieves the maximal gain with respect to the serial cost, and $\%_\mathsf{a}$ show the average of the gains achieved for all locations.

We now explain the results, which are obtained by applying the different levels of precision (a), (b), and (c) to the *original programs*, that is, the benchmarks without the subscript $_{cs}$ in Table 2. The execution times taken by the parallel cost analysis for the three approaches, $\mathbf{T}^a_{\widehat{\mathcal{P}}}$, $\mathbf{T}^b_{\widehat{\mathcal{P}}}$ and $\mathbf{T}^c_{\widehat{\mathcal{P}}}$, are very similar. The columns $\#^a_\mathsf{P}$, $\#^b_\mathsf{P}$ and $\#^c_\mathsf{P}$ show that the number of elements in $\mathcal{N}^+(\mathcal{G})$ are roughly the same for all benchmarks, except for BookShop, which goes from 6 in case (a) to 12 in (b) and (c). Regarding precision, we see that the precision gained by the naïve approach (column $\mathbf{T}^a_{\widehat{\mathcal{P}}}$) ranges from 91.7% (PeerToPeer) to 75.3% (in DistHT), which is a significant gain (24.7%) with respect to

	Naïve approach					*Unfolding disjunctions*					*Filtering infeasible paths*				
Benchmark	$\mathbf{T}^{a}_{\widehat{\mathcal{P}}}$	$\#^{a}_{\mathrm{P}}$	$\%^{a}_{\mathrm{m}}$	$\%^{a}_{\mathrm{a}}$	$\%^{a}_{\widehat{\mathcal{P}}}$	$\mathbf{T}^{b}_{\widehat{\mathcal{P}}}$	$\#^{b}_{\mathrm{P}}$	$\%^{b}_{\mathrm{m}}$	$\%^{b}_{\mathrm{a}}$	$\%^{b}_{\widehat{\mathcal{P}}}$	$\mathbf{T}^{c}_{\widehat{\mathcal{P}}}$	$\#^{c}_{\mathrm{P}}$	$\%^{c}_{\mathrm{m}}$	$\%^{c}_{\mathrm{a}}$	$\%^{c}_{\widehat{\mathcal{P}}}$
BBuffer	10	6	3.0	22.7	95.7	11	9	3.0	19.7	77.4	11	9	3.0	19.7	77.4
BBuffer$_{cs}$	11	6	3.0	22.7	95.7	12	9	3.0	19.7	77.4	12	9	3.0	19.7	77.4
MailServer	5	5	62.4	71.1	88.8	6	5	62.4	71.1	88.8	5	5	62.4	71.1	88.8
MailServer$_{cs}$	14	5	22.1	53.3	75.2	16	10	21.7	53.0	74.9	17	10	21.7	53.0	74.9
DistHT	19	6	4.0	25.8	76.1	21	8	4.0	25.6	75.3	21	8	4.0	25.6	75.3
DistHT$_{cs}$	300	13	4.0	34.4	64.8	362	46	4.0	29.6	55.1	359	37	4.0	25.8	47.5
Chat	78	5	4.4	62.6	82.9	93	8	4.4	61.6	81.6	95	8	4.4	61.6	81.6
Chat$_{cs}$	516	10	4.4	64.4	87.4	681	53	4.4	64.0	81.9	657	39	4.4	57.4	76.8
BookShop	105	6	2.0	35.0	87.0	110	12	2.0	34.9	87.0	109	12	2.0	34.9	87.0
BookShop$_{cs}$	188	13	92.7	93.7	95.5	32462	1395	2.0	54.2	87.4	8309	554	2.0	44.8	54.9
PeerToPeer	42	2	19.4	58.3	97.1	46	2	19.4	58.3	97.1	49	2	19.4	58.3	97.1
PeerToPeer$_{cs}$	123	2	94.0	94.1	94.2	130	2	94.0	94.1	94.2	137	2	94.0	94.1	94.2

Table 2. Experimental results (times in ms)

the serial cost. However, the gains obtained by expanding and filtering paths, $\%^{b}_{\widehat{\mathcal{P}}}$ and $\%^{c}_{\widehat{\mathcal{P}}}$, are not relevant with respect to the gains obtained by the naïve approach ($\%^{a}_{\widehat{\mathcal{P}}}$).

The DFGs of the original programs have, at least, one path that contains most nodes of the program due to big cycles in the DFGs, and this path leads to the maximal cost for the three approaches. Besides, these paths include nodes multiple times as they cannot be filtered because the upper bound indicates that they might be executed multiple times. This situation is improved by analyzing the context-sensitive version of the program. Observing $\%^{a}_{\mathrm{a}}$, $\%^{b}_{\mathrm{a}}$ and $\%^{c}_{\mathrm{a}}$ we can see that, on average, the improvements obtained for each location separately are significant, ranging from 19.7% of BBuffer to 71.1% of MailServer. In the best case, one particular location can gain up to 97% (100 - 3 %) with respect to the serial cost of the program (see columns $\%^{a}_{\mathrm{m}}$, $\%^{b}_{\mathrm{m}}$ and $\%^{c}_{\mathrm{m}}$ for BBuffer).

Now let us focus on the results obtained by analyzing the context-sensitive versions of the programs, that is, the benchmarks with the subscript $_{cs}$. Due to the increased size and complexity of the obtained DFGs, we see that the execution times for applying the analysis are significantly increased, especially for BookShop, which goes from 188ms in the analysis of the original programs to 32462 ms in the column $\mathbf{T}^{b}_{\widehat{\mathcal{P}}}$. The results obtained from the analysis of the context-sensitive version of the program show huge differences between the different benchmarks. For instance, in PeerToPeer, the number of elements in $\mathcal{N}^{+}(\mathcal{G})$ is 2 for all approaches (see columns $\#^{a}_{\mathrm{P}}$, $\#^{b}_{\mathrm{P}}$ and $\#^{c}_{\mathrm{P}}$). It is due to the high density of the graph: it contains 5.7 edges per node on average, which leads to a maximal path with almost all nodes of the graph, resulting in an upper bound close to the serial one. Consequently, PeerToPeer does not gain any precision improvement from approach (a) to (c). On the contrary, we have BookShop, which starts with 13 elements in $\#^{a}_{\mathrm{P}}$ and increases to 1395 in $\#^{b}_{\mathrm{P}}$. Such difference leads to a precision gain, from 95.5% of $\%^{a}_{\widehat{\mathcal{P}}}$ to 87.4% of $\%^{b}_{\widehat{\mathcal{P}}}$. Interestingly, approach (c) reduces the number of paths from 1395 to 554, resulting in bigger increment of precision from 87.4% to 54.9% in $\%^{c}_{\widehat{\mathcal{P}}}$. Chat shows a similar behaviour, the precision gained by analyzing the context-sensitive version of the program is also relevant, from 87.4% of $\%^{a}_{\widehat{\mathcal{P}}}$ to 76.8% $\%^{c}_{\widehat{\mathcal{P}}}$. Finally DistHT also presents a significant precision improvement for the context-sensitive version, from 64.8% of (a) to 47.5% of (c). There are several factors that contribute to the improvements obtained

	Context Insensitive			*Context Sensitive*		
Benchmark	$\%_r^a$	$\%_r^b$	$\%_r^c$	$\%_r^a$	$\%_r^b$	$\%_r^c$
BBuffer	46.29	50.33	50.33	46.29	50.33	50.33
MailServer	61.67	61.67	61.67	62.16	62.20	62.20
DistHT	50.89	51.62	51.62	50.53	62.94	79.06
Chat	33.33	34.19	34.19	31.75	33.52	35.82
BookShop	52.85	52.85	52.85	50.09	53.34	91.47
PeerToPeer	30.57	30.57	30.57	31.86	31.86	31.86

Table 3. Experimental comparison with real executions

by the context-sensitive versions of the benchmarks: (1) it produces a more precise DFG; (2) MHP results are more precise; and (3) the information obtained by $\mathcal{S}^t(\bar{x})$ is able to detect more nodes that can be executed only once and consequently more paths are filtered, giving more precise upper bounds.

All in all, we argue that our experimental evaluation shows that parallel cost analysis is feasible and accurate. From these results, we conclude that the parallel cost analysis can achieve significant improvements with respect to the serial cost analysis, reaching the gains of 52.3% (100 - 47.5%). These results suggest that the naïve approach could be enough for the analysis of the original program, while the analysis of the context-sensitive version of the program can lead to significant improvements combined with the filtering of infeasible paths.

Finally, we want to assess the accuracy of our parallel cost analysis. We do that by comparing the actual parallel cost of real runs, obtained by using the aPET system [9] as profiler, against the estimated parallel cost obtained by evaluating the generated parallel upper bounds. This comparison has required the extension of the aPET system to include the computation of the parallel cost. Table 3 summarizes the results of comparing the actual parallel cost with respect to the three levels of precision of the parallel cost analysis described above, (a) the *naïve* approach, (b) unfolding disjunctions, and (c) filtering paths. We show the percentage that the actual executions represent with respect to the different upper bounds computed by our analysis $(\mathcal{P}(\bar{z})/\widehat{\mathcal{P}}(\bar{z})) * 100$. Let us discuss the most interesting aspects. We can see that Chat and PeerToPeer are the benchmarks for which we obtain the most imprecise results, where the actual cost is around 35% of the upper bound obtained. This can be explained by the high density of the DFGs, both the context sensitive and the context-insensitive versions (see Table 1). Due to this fact, there exists at least one path that traverses most nodes of the DFG, and consequently most of the serial cost is added to the parallel cost. On the contrary, we have BookShop, DistHT and MailServer whose DFGs are not as dense as the graphs of Chat and PeerToPeer, and whose parallel cost upper bound is quite tight with respect to the real execution, especially in the case of the BookShop. We also want to highlight the accuracy obtained for BookShop, as our analysis is able to discard a significant number of paths using precision (c), ending in more precise results in comparison with (b).

Additionally we have applied our analysis to a bigger program –consisting of 1400 lines of code– called TradingSystem, which models a supermarket cash desk line: it includes the processes at a single cash desk (e.g., scanning products using a bar code scanner, paying by cash or by credit card); it also handles bill printing, as well as other administrative tasks. A store consists of an arbitrary number of cash desks. Each of them is connected to the store server, holding store-local product data such as inventory stock, prices, etc. The system is divided into two main parts, the CashDeskInstallation and the CashDeskEnvironment. The CashDeskInstallation contains those classes that are in charge of modeling the hardware behaviour and the CashDeskEnvironment,

which models at higher level the behaviour of the system. Furthermore, the TradingSystem includes a class for modeling a bank implementation and another one that models an inventory system. The application of the *naïve approach* and the *unfolding disjunctions approach* of our analysis is performed in 11 and 13 minutes, and the paths are computed in 3.5 seconds. However, the DFG has 352 nodes and the size of the cost expressions including the block-level cost centers is huge. Thus, the time taken for the *filtering paths approach* is much longer, 31 minutes, because it requires a significant amount of time to evaluate the upper bound for all nodes in order to get the number of times they are executed. The results show that the *naïve approach* and the *unfolding disjunctions approach* are 75% and 73.4% of the sequential upper bound, respectively. The *filtering paths approach* shows the same precision as the *unfolding disjunctions approach*. The computation of the context-sensitive version of the TradingSystem produces a timeout after 5 hours of analysis. Nevertheless, we argue that our implementation, in spite of being a prototype, works reasonably well even for big and complex programs, but its application requires a trade-off between the precision that can be achieved and the computational time to obtain the upper bound.

9 CONCLUSIONS AND RELATED WORK

We have presented what is, to the best of our knowledge, the first static cost analysis for distributed systems which exploits the parallelism among distributed locations in order to infer a more precise estimation of the parallel cost. Our experimental results show that parallel cost analysis can be of great use to know if an application succeeds in exploiting the parallelism available at distributed locations. There is recent work on cost analysis for distributed systems which infers the peak of the *serial* cost [11], i.e., the maximal amount of resources that a distributed component might need along its execution. This notion is different to the parallel cost that we infer since it is still serial; i.e., it accumulates the resource consumption in each component and does not exploit the overall parallelism as we do. Thus, the techniques used to obtain the peak cost are also different: the peak cost is obtained by abstracting the information in the queues of the different locations using graphs and finding the cliques in such graphs [11]. The only part in common with our analysis is that both rely on an underlying resource analysis for the serial execution that uses cost centers and on a MHP analysis, but the methods used to infer each notion of cost are fundamentally different. The work in [11] is improved in [12] to infer the peak for non-cumulative resources that increase and decrease along the execution (e.g., memory usage in the presence of garbage collection). In this sense, the notion of parallel cost makes sense only for cumulative resources since its whole purpose is to observe the efficiency gained by parallelizing the program in terms of resources used (and accumulated) in parallel by distributed components. Recent work has applied type-based amortized analysis for deriving bounds of parallel first-order functional programs [32]. This differs from our work in the concurrent programming model, as they do not allow explicit references to locations in the programs and there is no distinction between blocking and non-blocking synchronization. The cost measure is also quite different from the one used in our approach.

Dynamic analysis (or profiling) gathers information from running a system on specific input data, typically gathering counts of statements or procedure calls, or gathering timing information [27]. Closest to the proposed parallel cost analysis, critical path analysis identifies the path through the program's execution that consumed the most time. Critical path analysis is used in profiling tools such as IPS [43] and IPS-2 [42] to help the programmer identify bottlenecks in parallel programs, whereas Slack [33] introduces a metric to estimate the potential gain in execution time by improving procedures along the critical path. Causal profiling [23, 53] is a technique directly addressing asymptotic parallelism, the potential speed-up of a program executed on a large number of processors. This technique uses virtual speedup (introducing delays in concurrently executing

code) to automatically identify opportunities for optimizing code for parallel execution. Compared to static techniques such as parallel cost analysis, profiling is limited to the inspection of behavior on the selection of input and it can miss the anomalous (costly) behavior that only happens on some specific input.

To simplify the presentation, we have assumed that the different locations execute one instruction in one cost unit. This is without loss of generality because if they execute at a different speed we can weight their block-level costs according to their relative speeds. We argue that our work is of wide applicability as it can be used in combination with any cost analysis for serial execution which provides us with cost information at the level of the required fragments of code (e.g., [28, 31, 56]). It can also be directly adopted to infer the cost of parallel programs which spawn several tasks to different processors and then use a join operator to synchronize with the termination of all of them (the latter correspond to a get instruction on all spawned tasks in our setting). As future work, we plan to incorporate in the analysis information about the scheduling policy used at the locations (observe that each location could use a different scheduler). In particular, we aim at inferring (partial) orderings among the tasks of each location by means of static analysis.

Analysis and verification techniques for concurrent programs seek finite representations of the program traces to avoid an exponential explosion in the number of traces (see [26] and its references). In this sense, our DFGs provide a finite representation of all traces that may arise in the distributed system. A *multi-threaded concurrency model* entails an exponential explosion in the number of traces because task scheduling is preemptive. In contrast, *cooperative concurrency* as studied in this paper is gaining attention both for distributed [34] and for multicore systems [19, 52], because the amount of interleaving between tasks that must be considered in analyses is restricted to synchronization points which are explicit in the program.

ACKNOWLEDGMENTS

This work was funded partially by the Spanish MINECO project TIN2015-69175-C4-2-R, by the CM project S2013/ICE-3006 and by the SIRIUS Centre for Scalable Data Access (www.sirius-labs.no).

REFERENCES

[1] WCET tools. http://www.rapitasystems.com/WCET-Tools, 2012.
[2] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 183–193, New York, NY, USA, 2007. ACM.
[3] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.
[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2006.
[5] E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, and G. Román-Díez. Object-Sensitive Cost Analysis for Concurrent Objects. *Software Testing, Verification and Reliability*, 25(3):218–271, 2015.
[6] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In *Proc. of TACAS'14*, volume 8413 of *LNCS*, pages 562–567. Springer, 2014.
[7] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.
[8] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science (Special Issue on Quantitative Aspects of Programming Languages)*, 413(1):142–159, 2012.
[9] E. Albert, P. Arenas, M. Gómez-Zamalloa, and P. Y. Wong. aPET: A Test Case Generation Tool for Concurrent Objects. In B. Meyer, L. Baresi, and M. Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 595–598. ACM, 2013.
[10] E. Albert, J. Correas, E. B. Johnsen, and G. Román-Díez. Parallel Cost Analysis of Distributed Systems. In *Procs. of SAS'15*, volume 9291 of *LNCS*, pages 275–292. Springer, 2015.

[11] E. Albert, J. Correas, and G. Román-Díez. Peak Cost Analysis of Distributed Systems. In *Proc. of SAS'14*, volume 8723 of *LNCS*, pages 18–33, 2014.
[12] E. Albert, J. Correas, and G. Román-Díez. Non-Cumulative Resource Analysis. In *Procs. of TACAS'15*, volume 9035 of *LNCS*, pages 85–100. Springer, 2015.
[13] E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of May-Happen-in-Parallel in Concurrent Objects. In *Proc. of FORTE'12*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.
[14] E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. May-Happen-in-Parallel Analysis for Actor-based Concurrency. *ACM Transactions on Computational Logic*, 17(2):11:1–11:39, Mar. 2016.
[15] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
[16] H. G. Baker Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proc. Symp. on Artificial Intelligence and Programming Languages*, pages 55–59. ACM Press, New York, NY, USA, 1977.
[17] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.
[18] R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised Selected Papers*, volume 4339 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2005.
[19] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I. Pun, S. L. Tapia Tarifa, T. Wrigstad, and A. M. Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*, pages 1–56. Springer, 2015.
[20] D. Caromel and L. Henrio. *A Theory of Distributed Object.* Springer, 2005.
[21] D. Caromel and Y. Roudier. Reactive programming in Eiffel//. In J. Briot, J. Geib, and A. Yonezawa, editors, *Proc. Conf. on Object-Based Parallel and Distributed Computation (OBPDC '95)*, volume 1107 of *Lecture Notes in Computer Science*, pages 125–147. Springer, 1996.
[22] G. Cugola and C. Ghezzi. CJava: Introducing concurrent objects in Java. In M. E. Orlowska and R. Zicari, editors, *4th Intl. Conf. on Object Oriented Information Systems (OOIS'97)*, pages 504–514. Springer, 1997.
[23] C. Curtsinger and E. D. Berger. Coz: finding code that counts with causal profiling. In *Proc. 25th Symposium on Operating Systems Principles (SOSP 2015)*, pages 184–197. ACM, 2015.
[24] F. S. de Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017.
[25] P. Di, Y. Sui, D. Ye, and J. Xue. Region-based may-happen-in-parallel analysis for C programs. In *44th International Conference on Parallel Processing, ICPP 2015*, pages 889–898. IEEE Computer Society, 2015.
[26] A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In *POPL*, pages 129–142. ACM, 2013.
[27] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler (with retrospective). In K. S. McKinley, editor, *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*, pages 49–57. ACM, 1982.
[28] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL'09*, pages 127–139. ACM, 2009.
[29] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Comput. Sci.*, 410(2-3):202–220, 2009.
[30] R. H. Halstead, Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Prog. Lang. Syst.*, 7(4):501–538, Oct. 1985.
[31] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *Proc. of POPL'11*, pages 357–370. ACM, 2011.
[32] J. Hoffmann and Z. Shao. Automatic Static Cost Analysis for Parallel Programs. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015*, volume 9032 of *LNCS*, pages 132–157. Springer, 2015.
[33] J. K. Hollingsworth and B. P. Miller. Slack: A new performance metric for parallel programs. Technical report, University of Maryland and University of Wisconsin-Madison, 1994.
[34] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Procs. of FMCO'10*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
[35] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, Mar. 2007.
[36] E. B. Johnsen, O. Owe, and M. Arnestad. Combining active and reactive behavior in concurrent objects. In D. Langmyhr, editor, *Proc. of the Norwegian Informatics Conference (NIK'03)*, pages 193–204. Tapir Academic Publisher, Nov. 2003.
[37] J. E. Kelley, Jr. Critical-path planning and scheduling: Mathematical basis. *Operations Research*, 9:296–320, 1961.
[38] J. K. Lee, J. Palsberg, and R. Majumdar. Complexity results for may-happen-in-parallel analysis. Manuscript, 2010.

[39] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In R. L. Wexelblat, editor, *Proc. Conf. of Programming Language design and Implementation (PLDI)*, pages 260–267. ACM Press, New York, NY, USA, 1988.

[40] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann Publishers Inc., 2012.

[41] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, 2005.

[42] B. P. Miller, M. Clark, J. K. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. IPS-2: the second generation of a parallel program measurement system. *IEEE Trans. Parallel Distrib. Syst.*, 1(2):206–217, 1990.

[43] B. P. Miller and C. Yang. IPS: an interactive and automatic performance measurement tool for parallel and distributed programs. In *Proc. 7th International Conference on Distributed Computing Systems*, pages 482–489. IEEE Computer Society, 1987.

[44] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *POPL'97: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997. ACM.

[45] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your Contexts Well: Understanding Object-Sensitivity. In *In Proc. of POPL'11*, pages 17–30. ACM, 2011.

[46] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[47] H. Sutter and J. R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[48] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, July 1981.

[49] B. Wegbreit. Mechanical Program Analysis. *Communications ACM*, 18(9):528–539, 1975.

[50] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[51] D. Wyatt. *Akka Concurrency.* Artima, 2013.

[52] J. Yi, C. Sadowski, S. N. Freund, and C. Flanagan. Cooperative concurrency for a multicore world - (extended abstract). In *Procs. of RV'11*, volume 7186 of *LNCS*, pages 342–344. Springer, 2012.

[53] A. Yoga and S. Nagarakatte. A fast causal profiler for task parallel programs. In *Proc. 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, pages 15–26. ACM, 2017.

[54] Y. Yokote and M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, 1987.

[55] A. Yonezawa, J. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In N. K. Meyrowitz, editor, *ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'86)*, pages 258–268. ACM Press, Nov. 1986.

[56] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.

10 PROOF FOR THEOREM 6.7

We will first prove Lemma 6.5 that states that any transition in a trace is covered by a node in $\mathcal{G}$.

> *[Lemma 6.5] Given a program P, let t be an execution trace of P and let $\mathcal{G}$ be the DFG of P. For any transition $W_i \in t$ there exists a node in $\mathcal{G}$ that covers W_i.*

Proof. The soundness of the points-to analysis [5] guarantees that the set of abstract locations O is finite and it is a partition of the set of concrete locations. We use $loc \in o$ to represent that loc is abstracted by o.

Let $\mathcal{G}$ be of the form $\mathcal{G} = \langle V, E \rangle$. We study the relation between two running tasks at a parallel transition $W_i \equiv \mathtt{Locs}_i\ \mathtt{Tsks}_i \rightsquigarrow \mathtt{Locs}_{i+1}\ \mathtt{Tsks}_{i+1}$. This lemma can be proved by considering all possible local reduction rules in the semantics shown in Figure 1. The cases are considered as follows:

- We assume that the rules for standard sequential instructions are correctly generated in the CFG and are therefore covered in $\mathcal{G}$ by means of edges in E_{cfg} or nodes in V (see Definition 6.1).
- **[Rules** NewLoc, Async, Async + Fut, Return, Get, Await1 **and** Await2] Let $loc(lid, tid) \in \mathtt{Locs}_i$ be a location that executes a task $tsk(tid, lid, _, _, inst; s) \in \mathtt{Tsks}_i$. If $lid \in o$ and $inst$ is the instruction to be executed by the task that corresponds to any of these rules and it is in block b, then $o{:}b$ covers transition W_i (item (1) of Definition 6.4).
- **[Rules** Select1 **and** Select2] Let $loc(lid, \perp) \in \mathtt{Locs}_i$ be a location, $tsk(tid, lid, _, _, inst; s) \in \mathtt{Tsks}_i$ be a task that satisfies the pre-conditions of the rule Select1 or Select2 applied in transition W_i, and function select($\mathtt{Tsks}_i$) returns tid. There are two situations in which a task is selected for execution: either a fresh task has just been spawned on an idle location from another location, or the location's processor has just been released in the previous transition and the tasks queue contains some selectable tasks. If $lid \in o$ and the rule to be applied to execute $inst$ in transition W_{i+1} is in block b, node $o{:}b$ covers both transitions W_i and W_{i+1} (item (2) of Definition 6.4).

These cases cover all possible applications of the transition relation $\rightsquigarrow$ for local reduction rules shown in Figure 1.

Observe that in a parallel transition W_i there may be some locations that do not apply any local reduction rule, handled by rule Context. This occurs due to one of the following causes:

- When the location is idle and every task in the location's queue either (1) is terminated, i.e., its sequence of instructions is of the form $\epsilon(v)$, or (2) is awaiting another task that is not terminated, i.e., it has executed Await2 in a previous transition.
- When the location is waiting at a Get instruction for a future variable.

If $\mathtt{Tsks}_{i+1}$ contains non-terminated tasks and all of them are in one of these cases, the trace has reached a deadlock. □

The proof of Theorem 6.7 is built upon two additional lemmata. First, let us study the situations in which a task starts or resumes execution at a location.

Lemma 10.1. *Given a program P, let $W_i \equiv \mathtt{Locs}_i\ \mathtt{Tsks}_i \rightsquigarrow \mathtt{Locs}_{i+1}\ \mathtt{Tsks}_{i+1}$ and $W_{i+1} \equiv \mathtt{Locs}_{i+1}\ \mathtt{Tsks}_{i+1} \rightsquigarrow \mathtt{Locs}_{i+2}\ \mathtt{Tsks}_{i+2}$ be two consecutive parallel transitions of an execution trace t of P. The following holds for any location in $\mathtt{Locs}_i$ with identifier lid:*

(a) *If $loc(lid, \perp) \in \mathtt{Locs}_i$ applies rule* Select1 *or* Select2 *in transition W_i, then either the rule applied to $loc(lid, _)$ in transition W_{i+1} corresponds to the first instruction of the initial block of a method, or it is a rule* Await1.

(b) *If $loc(lid, _) \in \mathsf{Locs}_i$ does not apply any rule in transition W_i, then either no rule can be applied in transition W_{i+1}, or the only rule that can be applied to $loc(lid, _)$ in transition W_{i+1} is a rule* Select1 *or* Select2 *or* Get.

Proof.

(a) Rules Select1 and Select2 are executed only if the location's processor is idle. When a task starts executing at a location, the only instructions that release the location's processor are Return and Await2. Then, when the processor is released, the tasks in the location's queue can only be either completed or awaiting the completion of another task. Therefore, when a task from the queue is selected for execution, the next instruction to execute corresponds to the beginning of a method (if it has been spawned from another task) or it is an await instruction.

(b) A location $loc(lid, tid) \in \mathsf{Locs}_i$ does not apply any rule in transition W_i in two cases:
   - If $tid = \bot$ and there are no selectable tasks for location lid in Locs_i, i.e., the pre-conditions for rules Select1 or Select2 do not hold. The only rule to be applied by this location in transition W_{i+1} is any of these rules, if its corresponding pre-conditions become true, or the location continues idle otherwise.
   - If $tid \neq \bot$ and there are two tasks $tsk(tid, lid, _, _, \mathtt{x}{=}f.\mathrm{get}; s), tsk(tid_1, _, _, _, s_1) \in \mathsf{Tsks}_i$ such that $s_1 \neq \epsilon(_)$. In transition W_{i+1} location lid can only apply rule Get if task tid_1 terminates in transition W_i, or continues waiting for its termination otherwise.

□

We now prove Lemma 6.5 that states that there is a path from any any node in the DFG to a node in $\overline{\mathcal{B}}_{\mathsf{exit}}$:

> *[Lemma 6.6] Given a program P, let $\mathcal{G} = \langle V, E \rangle$ be the DFG of P. For any node $o{:}b \in V$, either $o{:}b \in \overline{\mathcal{B}}_{exit}$ or there exists a path p in $\mathcal{G}$ from $o{:}b$ to a node in $\overline{\mathcal{B}}_{exit}$.*

sketch. $\mathcal{G}$ is constructed in Definition 6.1 using the edges of the CFG extended with points-to information to produce the set E_{cfg} . As the language considered in this paper does not contain unconditional jump statements, any control structure of the language has a continuation edge in the CFG, in particular while loops and conditionals. It is easy to prove that the CFG of a well-formed program contains a path from any node to a node in $\mathcal{B}_{\mathsf{exit}}$, and this can be lifted to $\mathcal{G}$ and $\overline{\mathcal{B}}_{\mathsf{exit}}$. □

Now we prove Theorem 6.7:

> *[Theorem 6.7] Given a program P, let t be an execution trace of P and $\mathcal{G}$ be the DFG of P. There exists a path p in the graph $\mathcal{G}$ from the initial node to a node in $\overline{\mathcal{B}}_{exit}$ that covers all transitions in t.*

Proof. We first prove the following statement by induction on the length of the trace:

($\star$) *Given a DFG $\mathcal{G}$ and a trace $t {\equiv} W_0 \longrightarrow \cdots \longrightarrow W_i$, with $W_i \equiv \mathsf{Locs}_i\ \mathsf{Tsks}_i \rightsquigarrow \mathsf{Locs}_{i+1}\ \mathsf{Tsks}_{i+1}$, then for every location $loc \in \mathsf{Locs}_i$ the following holds:*
   If there exists a node $o{:}b$ in $\mathcal{G}$ that covers W_i at location loc, then there exists a path in $\mathcal{G}$ from the initial node to $o{:}b$ that covers all transitions in t.

- **Base Case:** The statement ($\star$) trivially holds as the first parallel transition corresponds to the evaluation of the rule for the first instruction in main, and it is covered by the initial node $o_0{:}\mathsf{main}_{\mathsf{init}}$, where o_0 is the abstract location that starts the program.

- **Induction Hypothesis:** Let us consider that $(\star)$ holds for a trace $t \equiv W_0 \longrightarrow \cdots \longrightarrow W_{i-1}$ of length i.
- **Inductive Case:** Consider a trace t of length $i+1$ and a parallel transition $W_i \equiv \mathtt{Locs}_i\ \mathtt{Tsks}_i \rightsquigarrow \mathtt{Locs}_{i+1}\ \mathtt{Tsks}_{i+1}$, $\mathtt{Locs}_i$ has the form $\{loc_1, \ldots, loc_m\}$ and $\mathtt{Locs}_{i+1}$ has the form $\{loc'_1, \ldots, loc'_{m'}\}$. We focus on the evaluation step of a location $loc_j \in \mathtt{Locs}_i$. Let $W_{i-1} \equiv \mathtt{Locs}_{i-1}\ \mathtt{Tsks}_{i-1} \rightsquigarrow \mathtt{Locs}_i\ \mathtt{Tsks}_i$ be the previous parallel transition. The following cases may occur, according to the local reduction rules showed in Figure 1:
  - $(\spadesuit)$ We assume that the rules for standard sequential instructions are correctly generated in the CFG and are therefore covered in $\mathcal{G}$ by nodes in V (see Definition 6.1).
    In particular, if the rule corresponds to the first instruction of the continuation block b_i after a method invocation, there is an edge produced by E_{call} (see Definition 6.1), and the statement $(\star)$ holds for W_i.
  - [**Rules** NewLoc, Async, Async + Fut **and** Return] These rules correspond to location creation, asynchronous method invocations, and method exit. Any of these instructions either do not partition the code into separate blocks, or they correspond to the last instruction of a block. There exists a location $loc''_j \in \mathtt{Locs}_{i-1}$ of the form $loc(lid''_j, _)$ with $lid''_j = lid_j$. By Lemma 10.1 (b), loc''_j must apply a rule in transition W_{i-1}. In all cases, W_i is either covered by the same node as W_{i-1} or there is an edge induced by the CFG (see $\spadesuit$), or it is the first instruction of a method and W_{i-1} executes a rule Select1 or Select2 at location loc_j (by Lemma 10.1 (a)). In any case, $(\star)$ holds for W_i.
  - [**Rule** Get] This rule corresponds to the case in which an instruction f.get retrieves the value returned by a completed task referenced by f. The get instruction is the first and the only instruction of a block $b \in \overline{\mathcal{B}}_{\text{get}}$ and $lid_j \in o_2$, where $loc_j \equiv loc(lid_j, _)$. There exists a location $loc''_j \in \mathtt{Locs}_{i-1}$ of the form $loc(lid''_j, _)$ with $lid''_j = lid_j$. The following two cases may occur:
    * If loc''_j applies a rule in the previous transition W_{i-1}, it means that the task referenced by f in the instruction f.get has already terminated before the get instruction is evaluated. The set E_{cfg} contains an edge from $pred(b)$ to the block b that contains the get instruction, produced by the block partitioning of the code using the CFG and the set $\overline{\mathcal{B}}_{\text{get}}$ (see $\spadesuit$).
    * If loc''_j does not apply any rule in transition W_{i-1}, it is because there is a task at $loc(lid''_k, _) \in \mathtt{Locs}_{i-1}$ such that $lid''_k \in o_1$, which the get instruction is synchronized with that has just terminated by applying rule Return in W_{i-1}. Let $o_1{:}\mathsf{m}_{\text{exit}}$ be the node that covers W_{i-1} at location loc''_k. The set E_{sync} defined in Definition 6.1 contains an edge from the exit blocks of the task in loc_k to the node $o_2{:}b$.

    In both cases, there exists an edge from a node that covers W_{i-1} to node $o_2{:}b$ that covers W_i, thus $(\star)$ holds for W_i.
  - [**Rule** Await1] This rule corresponds to the case in which an instruction await f retrieves the value returned by a completed task referenced by f. The await instruction is the first and the only instruction of a block $b \in \overline{\mathcal{B}}_{\text{await}}$ and $lid_j \in o$, where $loc_j \equiv loc(lid_j, _)$. There exists a location $loc''_j \in \mathtt{Locs}_{i-1}$ of the form $loc(lid''_j, _)$ with $lid''_j = lid_j$. By Lemma 10.1 (b), loc''_j must apply a rule in transition W_{i-1}. The following two cases may occur:
    * If loc''_j applies in W_{i-1} a rule different from Select2, it means that the task referenced by f in the instruction await f is completed before this rule is evaluated. The set E_{cfg} contains an edge from $pred(b)$ to the block b that contains the await

instruction, produced by the block partitioning of the code using the CFG and the set $\overline{\mathcal{B}}_{\mathsf{await}}$ (see ♠).

    * If loc''_j applies in W_{i-1} the rule Select2 (Lemma 10.1 (a)), then by Lemma 6.5 and Definition 6.4 both transitions W_{i-1} and W_i are covered by $o{:}b$ at location loc_j.

  In both cases, ($\star$) holds for W_i.
- [**Rule** Await2] This rule corresponds to the case in which an await instruction checks that the awaited task has not terminated and thus releases the location's processor. Let $lid_j \in o$, where $loc_j \equiv loc(lid_j, _)$. The await instruction is the only instruction of a block $b \in \overline{\mathcal{B}}_{\mathsf{await}}$ and the DFG contains edges to the preceding and subsequent blocks as in the standard construction of the CFG. There exists a location $loc''_j \in \mathtt{Locs}_{i-1}$ of the form $loc(lid''_j, _)$ with $lid''_j = lid_j$. By Lemma 10.1 (b), loc''_j must apply a rule in transition W_{i-1} and, by the pre-conditions of rules Select1 and Select2, loc''_j cannot apply any of these rules. Any other local rule of the semantics does not change the active task of loc''_j to a different task. Therefore, loc_j must execute in transition W_{i-1} the last instruction of the block $pred(b)$. E_{cfg} contains an edge induced by the CFG (see ♠) connecting $o{:}pred(b)$ and $o{:}b$, and thus ($\star$) holds for W_i.
- [**Rules** Select1 **and** Select2] These rules correspond to choosing a selectable task from the task queue to be executed at location loc_j. Let $W_{i-1} \equiv \mathtt{Locs}_{i-1}\ \mathtt{Tsks}_{i-1} \rightsquigarrow \mathtt{Locs}_i\ \mathtt{Tsks}_i$ be the previous transition, there is a location $loc''_j \in \mathtt{Locs}_{i-1}$ of the form $loc(lid''_j, _)$ with $lid''_j = lid_j$. When any of these rules is evaluated in transition W_i, the following two cases may occur:
    * loc''_j applies in transition W_{i-1} a rule Return or Await2 in the active task at this location and released the processor. If there exists some other tasks waiting at the location's queue, one of them may be selected. Since the MHP analysis returns correct results, these tasks exist in the location's queue at the same time, and therefore may-happen-in-parallel. The set $E_{\mathrm{ord1}} \cup E_{\mathrm{await}} \cup E_{\mathrm{ord2}}$ contains edges that represent this.
    * loc''_j is idle in transition W_{i-1} and it does not execute any rule. If loc_j does not evaluate any rule in W_{i-1}, and in particular it does not evaluate rule Select1 or Select2, then it is because in transition W_{i-1} there were no selectable tasks in the location's queue, but a task is selectable in transition W_i. This may happen in two cases:
        (a) A fresh task has been spawned in W_{i-1} at location loc_j from another task executing at any location.
        (b) There is a task in the location's queue waiting for the termination of another task, and the awaited task has terminated in transition W_{i-1}.

      The sets E_{call} and E_{ord2} contain edges for both cases.

In all cases, ($\star$) holds for any transition W_i of the trace t and thus Theorem 6.7 holds.

Theorem follows from the proof of Statement ($\star$), as it is possible to find a path that covers t, and Lemma 6.6, that guarantees that this path can be extended to reach a node in $\overline{\mathcal{B}}_{\mathsf{exit}}$.

Note that in the case of a non-terminating trace, the path p generated by the proof of Statement ($\star$) is infinite. Nevertheless, since $\mathcal{G}$ has a finite number of vertices and edges, we can always find a finite path that traverses the set of nodes that are traversed by p. For instance, we can use [48] to generate the path expressions in $pexpr(\mathcal{G})$, and then unfold disjunctions and include iterative subexpressions once in every path expression, as it is done in Section 7.1.2. Given an infinite path p,

the resulting set of finite paths contains a path that traverses the nodes traversed by p. Therefore, we can always find a finite extended path to a node in $\overline{\mathcal{B}}_{\text{exit}}$. □

11 PROOF FOR THEOREM 6.14

[Theorem 6.14] $\mathcal{P}(P(\bar{x})) \leq \widehat{\mathcal{P}}(P(\bar{x}))$.

PROOF. We will use $covered(t, o{:}b)$ to refer to the set of transitions in trace t that are covered by $o{:}b$. The soundness of the serial cost analysis [5] guarantees that $\mathcal{S}|_{o:b}(\bar{x})$ is an over-approximation of the parallel cost of the transitions executing instructions that belong to block b by tasks executed in any location abstracted by o. In particular, the following holds:

$$\sum_{W_i \,\in\, covered(t,o:b)} \mathcal{P}(W_i) \leq \mathcal{S}(\bar{x})|_{o:b} \tag{1}$$

Definition 3.2 states that $\mathcal{P}(P(\bar{x}))$ is defined as $\sup(\{\mathcal{P}(t) | t \in traces(P(\bar{x}))\})$ where $\mathcal{P}(P(t)) = \sum_{i=0}^{n} \mathcal{P}(W_i)$. By Theorem 6.7, given a trace $t \equiv W_0 \longrightarrow \cdots W_i \cdots \longrightarrow W_n$, where $W_n \equiv \texttt{Locs}_n\ \texttt{Tsks}_n \leadsto \texttt{Locs}_{n+1}\ \texttt{Tsks}_{n+1}$ and $\texttt{Locs}_{n+1}\ \texttt{Tsks}_{n+1}$ is a final state, there exists an execution path p that covers all transitions in t, and in fact some transitions might be covered by several nodes in p. Let s be the set $elements(p)$. By definition 6.10, $s \in \mathcal{N}(\mathcal{G})$. Since the cost of each block (obtained by using the block-level cost analysis) contains not only the cost of the block itself but this cost is multiplied by the number of times the block is visited, the following holds:

$$\sum_{i=0}^{n} \mathcal{P}(W_i) \leq \sum_{o:b \in s} \left(\sum_{W_i \,\in\, covered(t,o:b)} \mathcal{P}(W_i) \right) \leq \sum_{o:b \in s} \mathcal{S}(\bar{x})|_{o:b} \leq \mathcal{S}(\bar{x})|_s \tag{2}$$

The set $\mathcal{N}^+(\mathcal{G})$ is the set of *maximal* elements of $\mathcal{N}(\mathcal{G})$ with respect to set inclusion. Therefore, exists $N \in \mathcal{N}^+(\mathcal{G})$ such that $s \subseteq N$. Then, the following holds:

$$\sum_{i=0}^{n} \mathcal{P}(W_i) \leq \mathcal{S}(\bar{x})|_s \leq \mathcal{S}(\bar{x})|_N \leq \max_{N \in \mathcal{N}^+(\mathcal{G})} \mathcal{S}(\bar{x})|_N \leq \widehat{\mathcal{P}}(P(\bar{x})) \tag{3}$$

As (3) holds for any execution trace t, Theorem 6.14 also holds. □