Parallel Cost Analysis of Distributed Systems Author's version ⋆

Elvira Albert[1], Jesús Correas[1], Einar Broch Johnsen[2], Guillermo Román-Díez[3]

[1] DSIC, Complutense University of Madrid, Spain
[2] Dept. of Informatics, University of Oslo, Norway
[3] DLSIIS, Technical University of Madrid, Spain

Abstract. We present a novel static analysis to infer the *parallel cost* of distributed systems. Parallel cost differs from the standard notion of *serial cost* by exploiting the truly concurrent execution model of distributed processing to capture the cost of synchronized tasks executing in parallel. It is challenging to analyze parallel cost because one needs to soundly infer the parallelism between tasks while accounting for waiting and idle processor times at the different locations. Our analysis works in three phases: (1) It first performs a *block-level* analysis to estimate the serial costs of the blocks between synchronization points in the program; (2) Next, it constructs a *distributed flow graph* (DFG) to capture the parallelism, the waiting and idle times at the locations of the distributed system; Finally, (3) the parallel cost can be obtained as the path of maximal cost in the DFG. A prototype implementation demonstrates the accuracy and feasibility of the proposed analysis.

1 Introduction

Welcome to the age of distributed and multicore computing, in which software needs to cater for massively parallel execution. Looking beyond parallelism between independent tasks, *regular parallelism* involves tasks which are mutually dependent [17]: synchronization and communication are becoming major bottlenecks for the efficiency of distributed software. This paper is based on a model of computation which separates the asynchronous spawning of new tasks to different locations, 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.

This paper introduces a novel static analysis to study the efficiency of computations in this setting, by approximating how synchronization between blocks of

⋆ This work was funded partially by the EU project FP7-ICT-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu), by the Spanish MINECO project TIN2012-38137, and by the CM project S2013/ICE-3006. Appeared at *22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*. The final publication is available at http://link.springer.com/chapter/10.1007%2F978-3-662-48288-9_16 serial execution influences parallel cost. The analysis builds upon well-established static cost analyses for serial execution [2,8,21]. 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 [19] 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.

Previous work on cost analysis of distributed systems [2] 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, which allows us to represent all possible (equivalence classes of) paths that the execution of the distributed program can take.
3. *Path Expressions.* The problem of finding the parallel cost of executing the program boils down to finding the path of maximal cost in the DFG. Paths in the DFG are computed by means of the single-source path expression problem [18], which finds regular expressions that represent all paths.
4. *Parallel cost with concurrent tasks.* We leverage the previous two steps to the concurrent setting by handling tasks whose execution might suspend and interleave with the execution of other tasks at the same location.

We demonstrate the accuracy and feasibility of the presented cost analysis by implementing a prototype analyzer of parallel cost within the SACO system, a static analyzer for distributed concurrent programs. Preliminary experiments on some typical applications for distributed programs achieve gains up to 29% w.r.t. a serial cost analysis. The tool can be used online from a web interface available at `http://costa.ls.fi.upm.es/web/parallel`.

2 The Model of Distributed Programs

We consider a distributed programming model with explicit locations. Each location represents a processor with a procedure stack and an unordered buffer of pending tasks. Initially all processors are idle. When an idle processor's task buffer is non-empty, some task is selected for execution. Besides accessing its own processor's global storage, each task can post tasks to the buffer of any processor, including its own, and synchronize with the reception of tasks (synchronization will be presented later in Sec. 6). When a task completes, its processor becomes idle again, chooses the next pending task, and so on.

$$(\textsc{newLoc})\ \frac{\text{fresh}(\mathit{lid}'),\ \ l' = l[x \to \mathit{lid}']}{\begin{array}{l} \mathit{loc}(\mathit{lid}, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l, \langle x = \mathsf{newLoc}; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow \\ \mathit{loc}(\mathit{lid}, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l', s)\} \cup \mathcal{Q}) \parallel \mathit{loc}(\mathit{lid}', \bot, \{\}) \end{array}}$$

$$(\textsc{async})\ \frac{l(x) = \mathit{lid}_1,\ \ \text{fresh}(\mathit{tid}_1),\ \ l_1 = \mathit{buildLocals}(\bar{z}, m_1)}{\begin{array}{c} \mathit{loc}(\mathit{lid}, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l, \langle x.m_1(\overline{z}); s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow \\ \mathit{loc}(\mathit{lid}, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l, s)\} \cup \mathcal{Q}) \| \mathit{loc}(\mathit{lid}_1, \_, \{\mathit{tsk}(\mathit{tid}_1, m_1, l_1, \mathit{body}(m_1))\}) \end{array}}$$

$$\begin{array}{cc} (\textsc{select}) & (\textsc{return}) \\ \dfrac{\begin{array}{c} \mathit{select}(\mathcal{Q}) = \mathit{tid}, \\ t = \mathit{tsk}(\mathit{tid}, \_, \_, s) \in \mathcal{Q}, s \neq \epsilon(v) \end{array}}{\mathit{loc}(\mathit{lid}, \bot, \mathcal{Q}) \rightsquigarrow \mathit{loc}(\mathit{lid}, \mathit{tid}, \mathcal{Q})} & \dfrac{v = l(x)}{\begin{array}{c} \mathit{loc}(\mathit{lid}, \mathit{tid}, \{\mathit{tsk}(\mathit{tid}, m, l, \langle \mathsf{return}\ x; \rangle)\} \cup \mathcal{Q}) \rightsquigarrow \\ \mathit{loc}(\mathit{lid}, \bot, \{\mathit{tsk}(\mathit{tid}, m, l, \epsilon(v))\} \cup \mathcal{Q}) \end{array}} \end{array}$$

Fig. 1. Summarized Semantics for Distributed Execution

2.1 Syntax

The number of distributed locations need not be known a priori (e.g., locations may be virtual). Syntactically, a location will therefore be similar to an *object* and can be dynamically created using the instruction newLoc. The program consists of a set of methods of the form $M{::=}T\ m(\overline{T\ x})\{s\}$. Statements s take the form $s{::=}s; s \mid x{=}e \mid \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid \mathsf{return}\ x \mid x{=}\mathsf{newLoc} \mid x.m(\bar{z})$, where e is an expression, x, z are variables and m is a method name. The notation $\bar{z}$ is used as a shorthand for $z_1, \ldots, z_n$, and similarly for other names. 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.

2.2 Semantics

A *program state* S has the form $\mathit{loc}_1 \| \ldots \| \mathit{loc}_n$, denoting the currently existing distributed locations. Each *location* is a term $\mathit{loc}(\mathit{lid}, \mathit{tid}, \mathcal{Q})$ where *lid* is the location identifier, *tid* the identifier of the *active task* which holds the location's lock or $\bot$ if the lock is free, and $\mathcal{Q}$ the set of tasks at the location. Only the task which holds the location's *lock* can be *active* (running) at this location. All other tasks are *pending*, waiting to be executed, or *finished*, if they have terminated and released the lock. A *task* is a term $\mathit{tsk}(\mathit{tid}, m, l, s)$ where *tid* is a unique task identifier, m the name of the method executing in the task, l a mapping from local variables to their values, and s the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated and the return value v is available.

The execution of a program starts from a method m, in an initial state with an initial location with identifier 0 executing task 0 of the form $S_0{=}\mathit{loc}(0, 0, \{\mathit{tsk}(0, \mathsf{m}, l, \mathit{body}(\mathsf{m}))\})$. Here, l maps parameters to their initial values and local references to null (standard initialization), and $\mathit{body}(\mathsf{m})$ refers to the sequence of instructions in the method m. The execution proceeds from S_0 by evaluating *in parallel* the distributed locations. The transition $\to$ denotes a parallel transition W in which we perform an evaluation step $\rightsquigarrow$ (as defined in Fig. 1) at every distributed location loc_i with $i{=}1, \ldots, n$, i.e., $W{\equiv}\mathit{loc}_1 \| \ldots \| \mathit{loc}_n \to \mathit{loc}'_1 \| \ldots \| \mathit{loc}'_m$. If a location is idle and its queue is empty, the evaluation simply returns the same location state. Due to the dynamic creation of distributed locations, we have that $m \geq n$.

The transition relation $\rightsquigarrow$ in Fig. 1 defines the evaluation at each distributed location. The treatment of sequential instructions is standard and thus omitted. In NewLoc, an active task *tid* at location *lid* creates a location *lid'* with a free lock, which extends the program state. This explains that $m \geq n$. Async spawns a new task (the initial state is created by *buildLocals*) with a fresh task identifier tid_1 in a singleton queue for the location lid_1 (which may be *lid*). We here elide the technicalities of remote queue insertion in the parallel transition step, which basically merges locations with the same identifier by taking the union of the queues. Rule Select returns a task that is not finished, and it obtains the lock of the location. When Return is executed, the return value is stored in v. In addition, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding instruction $\epsilon(v)$).

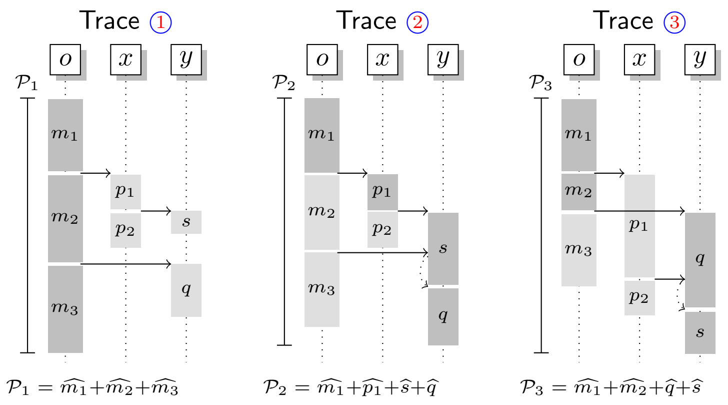
3 Parallel Cost of Distributed Systems

The aim of this paper is to infer an *upper bound* which is an over-approximation of the *parallel cost* of executing a distributed system. Given a parallel transition $W \equiv loc_1 \| \ldots \| loc_n \rightarrow loc'_1 \| \ldots \| loc'_m$, we denote by $\mathcal{P}(W)$ the parallel cost of the transition W. If we are interested in counting the number of executed transitions, then $\mathcal{P}(W) = 1$. If we know the time taken by the transitions, $\mathcal{P}(W)$ refers to the time taken to evaluate all locations. Thus, if two instructions execute in parallel, the parallel cost only accumulates the largest of their times. For simplicity, we assume that all locations execute one instruction in one cost unit. Otherwise, it must be taken into account by the cost analysis of the serial cost (see Sec. 8). Given a trace $t \equiv S_o \rightarrow \ldots \rightarrow S_{n+1}$ of the parallel execution, we define $\mathcal{P}(t) = \sum_{i=0}^{n} \mathcal{P}(W_i)$, where $W_i \equiv S_i \rightarrow S_{i+1}$. Since execution is non-deterministic in the selection of tasks, given a program $P(x)$, multiple (possibly infinite) traces may exist. We use $executions(P(\overline{x}))$ to denote the set of all possible traces for $P(\overline{x})$.

Definition 1 (Parallel cost). *The parallel cost of a program P on input values $\overline{x}$, denoted $\mathcal{P}(P(\overline{x}))$, is defined as $max(\{\mathcal{P}(t) | t \in executions(P(\overline{x}))\})$.*

Example 1. Fig. 2 (left) shows a simple method m that spawns two tasks by calling p and q at locations x and y, resp. In turn, 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. In the sequel we denote by $\widehat{m}$ the cost of block m. $\widehat{m_1}$, $\widehat{m_2}$ and $\widehat{m_3}$ denote, resp., 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 m. $\widehat{p_1}$ and $\widehat{p_2}$ are analogous. Let us assume that the block m_1 contains a loop that performs n iterations (where n is equal to the value of input parameter n if it is positive and otherwise n is 0) and at each iteration it executes 10 instructions, thus $\widehat{m_1}{=}10 * n$. Let us assume that block m_2 contains a loop that divides the value of n by 2 and that it performs at most $log_2(n+1)$ iterations. Assume that at each iteration it executes 20 instructions, thus $\widehat{m_2}{=}20 * log_2(n+1)$. These expressions can be obtained by cost analyzers of serial execution [2]. It is not crucial for the contents of this paper to know how these expressions are obtained,

```
void m (int n) {
  ... // m1
  x.p();
  ... // m2
  y.q();
  ... // m3
}
void p () {
  ... // p1
  y.s();
  ... // p2
}
```



Trace ①
Trace ②
Trace ③
o x y
$\mathcal{P}_1$
$\mathcal{P}_2$
$\mathcal{P}_3$
m_1 m_2 m_3 p_1 p_2 s q
$\mathcal{P}_1 = \widehat{m_1}+\widehat{m_2}+\widehat{m_3}$
$\mathcal{P}_2 = \widehat{m_1}+\widehat{p_1}+\widehat{s}+\widehat{q}$
$\mathcal{P}_3 = \widehat{m_1}+\widehat{m_2}+\widehat{q}+\widehat{s}$

Fig. 2. Motivating example

nor what the cost expressions are for the other blocks and methods. Thus, in the sequel, we simply refer to them in an abstract way as $\widehat{m_1}$, $\widehat{m_2}$, $\widehat{p_1}$, $\widehat{p_2}$ etc. ■

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.

Example 2. Fig. 2 (right) shows three possible traces of the execution of this example (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. It will be the worst (maximum) value of such expressions, that is, $\mathcal{P}=max(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \dots)$. In ② p_1 is shorter than m_2, and `s` executes before `q`. In ③, `q` is scheduled before `s`, resulting in different parallel cost expressions. In ①, the processor of location `y` becomes idle after executing `s` and must wait for task `q` to arrive. ■

In the general case, the inference of parallel cost is complicated because: (1) It is unknown if the processor is available when we spawn a task, as this depends on the duration of the tasks that were already in the queue; e.g., when task `q` is spawned we do not know if the processor is idle (trace ①) or if it is taken (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, we might even have 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 points make the static inference of parallel cost a challenging problem that, to the best of our knowledge, has not been previously addressed. Existing frameworks for the cost analysis of distributed systems [3,2] 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 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 in order to provide costs at the level of the blocks in which the program is partitioned, between synchronization points. In previous work, other extensions have been performed that use costs at the level of specific program points [4] or at the level of complete tasks [3], but the partitioning required by our parallel cost analysis is different. Later, we need to be able to cancel out the cost associated to blocks whose execution occurs in parallel with other blocks that have larger cost. The key notion of the extension is *block-level cost centers*, as defined below.

Block Partitioning. The need to partition the code into blocks will be clear when presenting the second phase of the analysis. Essentially, the subsequent analysis needs to have cost information for the following sets of blocks: $\mathcal{B}_{\mathsf{init}}$, the set of entry blocks for the methods; $\mathcal{B}_{\mathsf{exit}}$, the set of exit blocks for the methods, and $\mathcal{B}_{\mathsf{call}}$, the set of blocks ending with an asynchronous call. Besides these blocks, the standard partitioning of methods into blocks used to build the control flow graph (CFG) for the method is performed (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 block identifier b, $pred(b)$ is the set of blocks from which there are outgoing edges to block b in the CFG. Function $pred$ can also be applied to sets of blocks. We write $pp \in b$ (resp. $i \in b$) to denote that the program point pp (resp. instruction i) belongs to the block b.

Example 3. In Fig. 2, the traces show the partitioning in blocks for the methods m, p, q and s. Note that some of the blocks belong to multiple sets as defined above, namely $\mathcal{B}_{\mathsf{init}} = \{m_1, p_1, s, q\}$, $\mathcal{B}_{\mathsf{exit}}=\{m_3, p_2, s, q\}$, $\mathcal{B}_{\mathsf{call}}=\{m_1, m_2, p_1\}$. For instance, m_1 is both an entry and a call block, and s, as it is not partitioned, is both an entry and exit block. ■

Points-to Analysis. Since locations can be dynamically created, we need an analysis that abstracts them into a *finite* abstract representation, and that tells us which (abstract) location a reference variable is pointing-to. Points-to analysis [2,13,14] solves this problem. It infers 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 more or less accurate precision can be used (our implementation uses [2,13]): (1) $\mathcal{O}$, the set of abstract locations; (2) $\mathcal{M}$, the set of abstract tasks of the form $o.m$ where $o \in \mathcal{O}$ and m is a method name; (3) 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 to.

Example 4. In Fig. 2 we have three different locations, which are pointed to by variables o, x, y. For simplicity, we will use the variable name in italics to refer to the abstract location inferred by the points-to analysis. Thus, $\mathcal{O}=\{o, x, y\}$.

The abstract tasks spawned in the program are $\mathcal{M}=\{o.m, x.p, y.s, y.q\}$. In this example, the points-to abstraction is very simple. However, in general, locations can be reassigned, passed in parameters, have multiple aliases, etc., and it is fundamental to keep track of points-to information in an accurate way. ■

Cost Centers. The notion of cost center is an artifact used to define the granularity of a cost analyzer. In [2], the proposal is to define a cost center for each distributed component; i.e., cost centers are of the form $c(o)$ where $o \in \mathcal{O}$ and $c(\_)$ is the artifact used in the cost expressions to attribute the cost to the different components. Every time the analyzer accounts for the cost of executing an instruction *inst* at program point *pp*, it also checks at which location the instruction is executing. This information is provided by the points-to analysis as $O_{pp} = pt(pp, this)$. The cost of the instruction is accumulated in the cost centers of all elements in O_{pp} as $\sum c(o) * cost(inst), \forall o \in O_{pp}$, where $cost(inst)$ expresses in an abstract way the cost of executing the instruction. If we are counting steps, then $cost(inst) = 1$. If we measure time, $cost(inst)$ refers to the time to execute *inst*. Then, given a method $m(\bar{x})$, the cost analyzer will compute an upper bound for the serial cost of executing m of the form $\mathcal{S}_m(\bar{x}) = \sum_{i=1}^{n} c(o_i) * C_i$, where $o_i \in \mathcal{O}$ and C_i is a cost expression that bounds the cost of the computation carried out by location o_i when executing m. Thus, cost centers allow computing costs at the granularity level of the distributed components. If one is interested in studying the computation performed by one particular component o_j, we simply replace all $c(o_i)$ with $i \neq j$ by 0 and $c(o_j)$ by 1. The idea of using cost centers in an analysis is of general applicability and the different approaches to cost analysis (e.g., cost analysis based on recurrence equations [19], invariants [8], or type systems [9]) can trivially adopt 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 any cost analysis for serial execution.

Example 5. For the code in Fig. 2, we have three cost centers for the three locations that accumulate the costs of the blocks they execute; i.e., we have $\mathcal{S}_m(n) = c(o)*\widehat{m_1} + c(o)*\widehat{m_2} + c(o)*\widehat{m_3} + c(x)*\widehat{p_1} + c(x)*\widehat{p_2} + c(y)*\widehat{s} + c(y)*\widehat{q}$. ■

Block-level Cost Centers. In this paper, we need *block-level* granularity in the analysis. This can be captured in terms of block-level cost centers $\overline{\mathcal{B}}$ which contain all blocks combined with all location names where they can be executed. Thus, $\overline{\mathcal{B}}$ is defined as the set $\{o{:}b \in \mathcal{O} \times \mathcal{B} \mid o \in pt(pp, this) \wedge pp \in b\}$. We define $\overline{\mathcal{B}}_{\mathsf{init}}$ and $\overline{\mathcal{B}}_{\mathsf{exit}}$ analogously. In the motivating example, $\overline{\mathcal{B}} = \{o{:}m_1, o{:}m_2, o{:}m_3, x{:}p_1, x{:}p_2, y{:}q,\ y{:}s\}$. Every time the analyzer accounts for the cost of executing an instruction *inst*, it checks at which location *inst* is executing (e.g., o) and to which block it belongs (e.g., b), and accumulates $c(o{:}b) * cost(inst)$. It is straightforward to modify an existing cost analyzer to include block-level cost centers. Given a method $m(\bar{x})$, the cost analyzer now computes a *block-level upper bound* for the cost of executing m. This upper bound is of the form $\mathcal{S}_m(\bar{x}) = \sum_{i=1}^{n} c(o_i{:}b_i) * C_i$, where $o_i{:}b_i \in \overline{\mathcal{B}}$, and C_i is a cost expression that bounds the cost of the computation carried out by location o_i while executing block b_i. Observe that b_i need not be a block of m because we can have transitive calls from m to other methods; the cost of executing these calls accumulates in $\mathcal{S}_m$. The notation $\mathcal{S}_m(\bar{x})|_{o:b}$ is used to express the cost associated to $c(o{:}b)$ within the cost expression $\mathcal{S}_m(\bar{x})$, 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. Given a set of cost centers $N = \{o_0{:}b_0, \dots, o_k{:}b_k\}$, we let $\mathcal{S}_m(\bar{x})|_N$ refer to the cost obtained by setting to one the cost centers $c(o_i{:}b_i)$ such that $o_i{:}b_i \in N$. We omit m in $\mathcal{S}_m(\bar{x})|_N$ when it is clear from the context.

Example 6. The cost of the program using the blocks in $\mathcal{B}$ as cost centers, is $\mathcal{S}_m(n){=}c(o{:}m_1){*}\widehat{m_1}{+}c(o{:}m_2){*}\widehat{m_2}{+}c(o{:}m_3){*}\widehat{m_3}{+}c(x{:}p_1){*}\widehat{p_1}{+}c(x{:}p_2){*}\widehat{p_2}{+}c(y{:}s){*}\widehat{s}{+}$ $c(y{:}q){*}\widehat{q}$. We can obtain the cost for block $o{:}m_2$ as $\mathcal{S}_m(n)|_{o:m_2} = \widehat{m_2}$. With the serial cost assumed in Sec. 3, we have $\mathcal{S}_m(n)|_{o:m_2} = 20 * log_2(n+1)$. ■

5 Parallel Cost Analysis

This section presents our method to infer the cost of executing the 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 largest cost among them.

5.1 Distributed Flow Graph

The *distributed flow graph* (DFG), introduced below, aims at capturing the different flows of execution that the program can perform. According to the distributed model of Sec. 2, when the processor is released, any pending task of the same location could start executing. We use an existing *may-happen-in-parallel* (MHP) analysis [5,12] to approximate the tasks that could start their execution when the processor is released. This analysis infers pairs of program points (x, y) whose execution *might* happen in parallel. The soundness of the analysis guarantees that if (x, y) is not an MHP pair then there are no instances of the methods to which x or y belong whose program points x and y can run in parallel. The MHP analysis can rely on a points-to analysis in exactly the same way as our overall analysis does. Hence, we can assume that MHP pairs are of the form $(x{:}p_1, y{:}p_2)$ where x and y refer to the locations in which they execute. We use the notation $x{:}b_1 \parallel y{:}b_2$, where b_1 and b_2 are blocks, to denote that the program points of $x{:}b_1$ and $y{:}b_2$ might happen in parallel, and, $x{:}b_1 \nparallel y{:}b_2$ to indicate that they cannot happen in parallel.

Example 7. The MHP analysis of the example shown in Fig. 2 returns that $y{:}s \parallel y{:}q$, indicating that s and q might happen in parallel at location y. In addition, as we only have one instance of m and p, the MHP guarantees that $o{:}m_1 \nparallel o{:}m_3$ and $x{:}p_1 \nparallel x{:}p_2$. ■

The nodes in the DFG are the cost centers which the analysis in Sec. 4 has inferred. The edges represent the control flow in the sequential execution (drawn with normal arrows) and all possible orderings of tasks in the location's queues (drawn with dashed arrows). We use the MHP analysis results to eliminate the dashed arrows that correspond to unfeasible orderings of execution.

Definition 2 (Distributed flow graph). *Given a program P, its block-level cost centers $\overline{\mathcal{B}}$, and its points-to analysis results provided by function pt, we define its* distributed flow graph *as a directed graph $\mathcal{G} = \langle V, E\rangle$ with a set of vertices $V = \overline{\mathcal{B}}$ and a set of edges $E = E_1 \cup E_2 \cup E_3$ defined as follows:*

$$\begin{aligned}
E_1 &= \{o{:}b_1 \rightarrow o{:}b_2 \mid b_1 \rightarrow b_2 \text{ exists in } CFG\}\\
E_2 &= \{o_1{:}b_1 \rightarrow o_2{:}\mathsf{m}_{init} \mid b_1 \in \mathcal{B}_{call}, pp : \mathsf{x.m()} \in b_1, o_2 \in pt(pp, \mathsf{x})\}\\
E_3 &= \{o{:}b_1 \dashrightarrow o{:}b_2 \mid b_1 \in \mathcal{B}_{exit}, b_2 \in \mathcal{B}_{init}, o{:}b_1 \parallel o{:}b_2\}
\end{aligned}$$

Here, E_1 is the set of edges that exist in the CFG, but using the points-to information in $\overline{\mathcal{B}}$ in order to find out at which locations the blocks are executed. E_2 joins 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 know 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). Arrows are drawn for all possible combinations. These arrows capture the parallelism in the execution and allow us to gain precision w.r.t. the serial execution. Intuitively, they allow us to consider the maximal cost of the path that continues the execution and the path that goes over the spawned tasks. Finally, dashed edges E_3 are required for expressing the different orderings of the execution of tasks 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 method m in the queue of pending tasks. Thus, we do not consider this path in E_3 of the DFG.

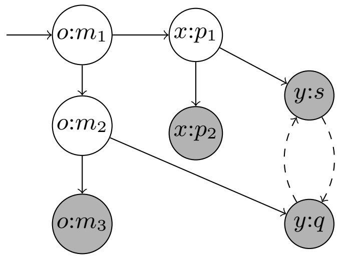


Fig. 3. DFG for Fig. 2

Example 8. Fig. 3 shows the DFG for the program in Fig. 2. The nodes are the cost centers in Ex. 6. Nodes in gray are the nodes in $\overline{\mathcal{B}}_{\mathsf{exit}}$, and it implies that the execution can terminate executing $o{:}m_3$, $x{:}p_2$, $y{:}s$ or $y{:}q$. Solid edges include those existing in the CFG of the sequential program but combined with the location's identity (E_1) and those derived from calls (E_2). Since $y{:}s \parallel y{:}q$ (see Ex. 7), the execution order of s and q at location y is unknown (see Sec. 3). This is modelled by means of the dashed edges (E_3). 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 $x{:}p_1$. ■

5.2 Inference of Parallel Cost

The next phase in our analysis consists of obtaining the maximal parallel cost from all possible executions of the program, based on the DFG. The execution paths in the DFG start in the initial node that corresponds to the entry method of the program, and finish in any node in $\overline{\mathcal{B}}_{\mathsf{exit}}$. The first step for the inference is to compute the set of execution paths by solving the so-called *single-source path expression problem* [18], which finds a regular expression (named *path expression*) for each node $v \in \overline{\mathcal{B}}_{\mathsf{exit}}$ representing all paths from an initial node to v. Given a DFG $\mathcal{G}$, we denote by $\mathit{pexpr}(\mathcal{G})$ the set of path expressions obtained from the initial node to all exit nodes in $\mathcal{G}$.

Example 9. To compute the set *pexpr* for the graph in Fig. 3, we compute the path expressions starting from $o{:}m_1$ and finishing in exit nodes, that is, the 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 $\mathsf{e}_{o:m_3} = o{:}m_1{\cdot}o{:}m_2{\cdot}o{:}m_3$, $\mathsf{e}_{x:p_2} = o{:}m_1{\cdot}x{:}p_1{\cdot}x{:}p_2$, $\mathsf{e}_{y:s} = o{:}m_1{\cdot}(x{:}p_1{\cdot}y{:}s \mid o{:}m_2{\cdot}y{:}q{\cdot}y{:}s){\cdot}(y{:}q{\cdot}y{:}s)^*$ and $\mathsf{e}_{y:q} = o{:}m_1{\cdot}(x{:}p_1{\cdot}y{:}s{\cdot}y{:}q \mid o{:}m_2{\cdot}y{:}q){\cdot}(y{:}s{\cdot}y{:}q)^*$. ■

The key idea to obtain the parallel cost from path expressions 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. Thus, we use sets instead of sequences since the multiplicity of the elements is already taken into account in the cost of the blocks. Given a path expression e, we define $\mathit{sequences}(\mathsf{e})$ as the set of paths produced by e and $\mathit{elements}(p)$ as the set of nodes in a given path p. We use the notions of *sequences* and *elements* to define the set $\mathcal{N}(\mathsf{e})$.

Definition 3. *Given a path expression* e, $\mathcal{N}(\mathsf{e})$ *is the following set of sets:*

$$\{s \mid p \in \mathit{sequences}(\mathsf{e}) \ \wedge \ s = \mathit{elements}(p)\}.$$

In practice, this set $\mathcal{N}(\mathsf{e})$ can be generated by splitting the disjunctions in e into different elements in the usual way, and adding the nodes within the repeatable subexpressions once. Thus, to obtain the parallel cost, it is sufficient to compute $\mathcal{N}^+(\mathsf{e})$, the set of *maximal* elements of $\mathcal{N}(\mathsf{e})$ with respect to set inclusion, i.e., those sets in $\mathcal{N}(\mathsf{e})$ which are not contained in any other set in $\mathcal{N}(\mathsf{e})$. Given a graph $\mathcal{G}$, we denote by $\mathit{paths}(\mathcal{G}) = \bigcup \mathcal{N}^+(\mathsf{e}),\ \mathsf{e} \in \mathit{pexpr}(\mathcal{G})$, i.e., the union of the sets of sets of elements obtained from each path expression.

Example 10. Given the path expressions in Ex. 9, we have the following sets:

$$\mathcal{N}^+(\mathsf{e}_{o:m_3}) = \{\underbrace{\{o{:}m_1, o{:}m_2, o{:}m_3\}}_{N_1}\}, \qquad \mathcal{N}^+(\mathsf{e}_{x:p_2}) = \{\underbrace{\{o{:}m_1, x{:}p_1, x{:}p_2\}}_{N_2}\}$$

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

Observe that these sets represent traces of the program. The execution captured by N_1 corresponds to trace ① of Fig. 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 (not shown in Fig. 2). Therefore, the set *paths* is $\{N_1, N_2, N_3, N_4\}$. ■

Given a set $N \in \mathit{paths}(\mathcal{G})$, we can compute the cost associated to N by using the block-level cost analysis, that is, $\mathcal{S}(\bar{x})|_N$. The parallel cost of the distributed system can be over-approximated by the maximum cost for the paths in $\mathit{paths}(\mathcal{G})$.

Definition 4 (Inferred parallel cost). *The* inferred parallel cost *of a program $P(\bar{x})$ with distributed flow graph $\mathcal{G}$, is defined as $\widehat{\mathcal{P}}(P(\bar{x})) = \max\limits_{N \in \mathit{paths}(\mathcal{G})} \mathcal{S}(\bar{x})|_N$.*

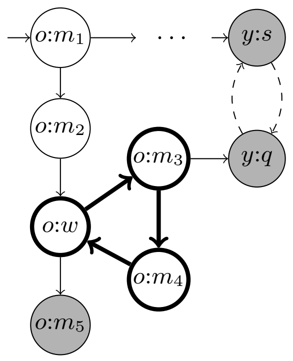
Although we have obtained the parallel cost of the whole program, we can easily obtain the parallel cost associated to a location o of interest, denoted $\widehat{\mathcal{P}}(P(\bar{x}))|_o$, by considering only the paths that lead to the exit nodes of this location. In particular, given a location o, we consider the set of path expressions $\mathit{pexpr}(\mathcal{G}, o)$ which are the subset of $\mathit{pexpr}(\mathcal{G})$ that end in an exit node of o. The above definition simply uses $\mathit{pexpr}(\mathcal{G}, o)$ instead of $\mathit{pexpr}(\mathcal{G})$ in order to obtain $\widehat{\mathcal{P}}(P(\bar{x}))|_o$.

Example 11. The cost is obtained by using the block-level costs for all nodes that compose the sets in *paths*. With the sets computed in Ex. 10, the overall parallel cost is: $\widehat{\mathcal{P}}(\mathsf{m}(n)) = \mathit{max}(\mathcal{S}(n)|_{N_1}, \mathcal{S}(n)|_{N_2}, \mathcal{S}(n)|_{N_3}, \mathcal{S}(n)|_{N_4})$. Importantly, $\widehat{\mathcal{P}}$ is more precise than the serial cost because all paths have at least one missing node. For instance, N_1 does not contain the cost of $x{:}p_1$, $x{:}p_2$, $y{:}s$, $y{:}q$ and N_3 does not contain the cost of $o{:}m_2$, $o{:}m_3$, $x{:}p_2$. Additionally, as $o{:}m_3$ is the only final node for location o, we have that $\widehat{\mathcal{P}}(\mathsf{m}(n))|_o = \mathcal{S}(n)|_{N_1}$. Similarly, for location y we have two exit nodes, $y{:}s$ and $y{:}q$, thus $\widehat{\mathcal{P}}(\mathsf{m}(n))|_y = \mathit{max}(\mathcal{S}(n)|_{N_3}, \mathcal{S}(n)|_{N_4})$. ■

Recall that when there are several calls to a block $o{:}b$ the graph contains only one node $o{:}b$ but the serial cost $\mathcal{S}(\bar{x})|_{o:b}$ 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 to one the corresponding cost center, the serial cost accumulates the cost of all executions of such nodes.

Example 12. The program to the right shows a modification of method m that adds a loop which includes the call $\mathsf{y.q()}$. The DFG for this code contains a cycle caused by the loop, composed by the nodes $o{:}w$, $o{:}m_3$ and $o{:}m_4$, where $o{:}w$ represents the entry block to the while loop. The execution might traverse such nodes multiple times and consequently multiple instances of $y{:}q$ might be spawned.

```
void m (int n) {
  ... // m1 instr
  x. p();
  ... //m2 instr
  while(n > 0) {
    n=n−1;
    ... //m3 instr
    y. q();
    ... //m4 instr
  }
  ... // m5 instr
}
```



A serial cost analyzer (e.g.[2]) infers that the loop is traversed at most n times and obtains a block-level serial cost of the form:

$$\mathcal{S}(n) = c(o{:}m_1)*\widehat{m_1} + c(o{:}m_2)*\widehat{m_2} + n*c(o{:}w)*\widehat{w} + n*c(o{:}m_3)*\widehat{m_3} + n*c(o{:}m_4)*\widehat{m_4} + c(o{:}m_5)*\widehat{m_5} + c(x{:}p_1)*\widehat{p_1} + c(x{:}p_2)*\widehat{p_2} + n*c(y{:}q)*\widehat{q} + c(y{:}s)*\widehat{s}$$

For the DFG we obtain some interesting sets that traverse the loop: $N_1 = \{o{:}m_1, o{:}m_2, o{:}w, o{:}m_3, o{:}m_4, o{:}m_5\}$ and $N_2 = \{o{:}m_1, o{:}m_2, o{:}w, o{:}m_3, o{:}m_4, y{:}q, y{:}s\}$. Observe that N_1 represents a trace that traverses the loop and finishes in $o{:}m_5$ and N_2 represents a trace that reaches $y{:}q$ by traversing the loop. The cost associated to N_1 is computed as $\mathcal{S}(n)|_{N_1} = \widehat{m_1}+\widehat{m_2}+n*\widehat{w}+n*\widehat{m_3}+n*\widehat{m_4}+\widehat{m_5}$. Note that $\mathcal{S}(n)|_{N_1}$ includes the cost of executing the nodes of the loop multiplied by n, capturing the iterations of the loop. Similarly, for N_2 we have $\mathcal{S}(n)|_{N_2}=\widehat{m_1}+\widehat{m_2}+n*\widehat{w}+n*\widehat{m_3}+n*\widehat{m_4}+n*\widehat{q}+\widehat{s}$, which captures that q might be executed n times. ∎

Theorem 1. $\mathcal{P}(P(\bar{x})) \leq \widehat{\mathcal{P}}(P(\bar{x}))$.

6 Parallel Cost Analysis with Cooperative Concurrency

We now extend the language to allow cooperative concurrency between the tasks at each location, in the style of concurrent (or active) object systems such as ABS [11]. The language is extended 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 as follows f=x.p(). The instruction await f? allows synchronizing the execution of the current task with the task p to which the future variable f is pointing; f.get is used to retrieve the value returned by the completed task. The semantics for these instructions is given in Fig. 4. The semantics of Async+Fut differs from Async in Fig. 1 in that it stores the association of the future variable to the task in the local variable table l. In Await1, the future variable we are awaiting points to a finished task and await can be completed. The finished task t_1 is looked up at all locations in the current state (denoted by Locs). Otherwise, Await2 yields the lock so any other task at the same location can take it. In Get1 the return value is retrieved after the task has finished and in Get2 the location is blocked allowing time to pass until the task finishes and the return value can be retrieved.

Handling concurrency in the analysis is challenging because we need to model the fact that we can lose the processor at the await instructions and another pending task can interleave its execution with the current task. The first extension needed is to refine the block partitioning in Sec. 4 with the set of blocks: $\mathcal{B}_{\mathsf{get}}$, the set of blocks starting with a get; and $\mathcal{B}_{\mathsf{await}}$, the set of blocks starting with an await. Such blocks contain edges to the preceding and subsequent blocks as in the standard construction of the CFG (and we assume they are in the set of edges E_1 of Def. 2). Fortunately, task interleavings can be captured in the graph in a clean way by treating await blocks as initial blocks, and their predecessors as ending blocks. Let b be a block which contains a f.get or await f? instruction. Then $awaited(\mathsf{f}, pp)$ 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 tasks a future variable is pointing to. Furthermore, the MHP analysis learns information from the await instructions, since after an await f? we know that the execution of the task to which f is linked is finished and thus it will not happen in parallel with the next tasks spawned at the same location.

$$\textsc{(async+fut)}\quad \frac{l(x) = lid_1,\ \text{fresh}(tid_1),\ l' = l[f \to tid_1],\ l_1 = buildLocals(\bar{z}, m_1)}{\begin{array}{c} loc(lid, tid, \{tsk(tid, m, l, \langle f = x.m_1(\bar{z}); s\rangle\} \cup \mathcal{Q}) \rightsquigarrow \\ loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q}) \parallel loc(lid_1, \_, \{tsk(tid_1, m_1, l_1, body(m_1))\}) \end{array}}$$

$$\textsc{(await1)}\quad \frac{l(f) = tid_1,\ loc(lid_1, \_, \mathcal{Q}_1) \in \mathsf{Locs},\ tsk(tid_1, \_, \_, s_1) \in \mathcal{Q}_1,\ s_1 = \epsilon(v)}{loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l, s)\} \cup \mathcal{Q})}$$

$$\textsc{(await2)}\quad \frac{l(f) = tid_1,\ loc(lid_1, \_, \mathcal{Q}_1) \in \mathsf{Locs},\ tsk(tid_1, \_, \_, s_1) \in \mathcal{Q}_1,\ s_1 \neq \epsilon(v)}{loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, \bot, \{tsk(tid, m, l, \langle \mathsf{await}\ f?; s\rangle)\} \cup \mathcal{Q})}$$

$$\textsc{(get1)}\quad \frac{l(f) = tid_1,\ tsk(tid_1, \_, \_, s_1) \in \mathsf{Locs}, s_1 = \epsilon(v),\ l' = l[x \to v])}{loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{x}{=}f.\mathsf{get}; s\rangle)\} \cup \mathcal{Q}) \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l', s)\} \cup \mathcal{Q})}$$

$$\textsc{(get2)}\quad \frac{l(f) = tid_1,\ tsk(tid_1, \_, \_, s_1) \in \mathsf{Locs}, s_1 \neq \epsilon(v)}{\begin{array}{l} loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{x}{=}f.\mathsf{get}; s\rangle)\} \cup \mathcal{Q}) \\ \rightsquigarrow loc(lid, tid, \{tsk(tid, m, l, \langle \mathsf{x}{=}f.\mathsf{get}; s\rangle)\} \cup \mathcal{Q}) \end{array}}$$

Fig. 4. Summarized Semantics of Concurrent Execution

Definition 5 (DFG with cooperative concurrency). *We extend Def. 2:*
$E_4 = \{o_1{:}\mathit{m_{exit}} \to o_2{:}b_2 \mid$ *either* $pp{:}\mathit{f.get}$ *or* $pp{:}\mathit{await\ f?} \in b_2, \mathit{m_{exit}} \in awaited(\mathit{f}, pp)\}$
$E_5 = \{o{:}b_1 \dashrightarrow o{:}b_2 \mid b_1 \in pred(\mathcal{B}_{\mathit{await}}), b_2 \in \mathcal{B}_{\mathit{await}} \cup \mathcal{B}_{\mathit{init}}, o{:}b_1 \parallel o{:}b_2\}$
$E_6 = \{o{:}b_1 \dashrightarrow o{:}b_2 \mid b_1 \in \mathcal{B}_{\mathit{exit}}, b_2 \in \mathcal{B}_{\mathit{await}}, o{:}b_1 \parallel o{:}b_2\}$

Here, E_4 contains the edges that relate the last block of a method with the corresponding synchronization instruction in the caller method, indicating that the execution can take this path after the method has completed. E_5 and E_6 contain dashed edges that represent the orderings between parts of tasks split by await instructions and thus capture the possible interleavings. E_5 considers the predecessor as an ending block from which we can start to execute another interleaved task (including await blocks). E_6 treats await blocks as initial blocks which can start their execution after another task at the same location finishes. As before, the MHP analysis allows us to discard those edges between blocks that cannot be pending to execute when the processor is released. Theorem 1 also holds for DFG with cooperative concurrency.

Example 13. Fig. 5 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 creates a new edge (the thick one) 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 m waits until p is finished. For the graph in Fig. 5 we have that *paths* is $\{\{o{:}m_1, x{:}p_1, x{:}p_2, o{:}m_3, o{:}q\}, \{o{:}m_1, o{:}m_2, o{:}m_3, o{:}q\}\}$. Observe that the thick edge is crucial for creating the first set in *paths*. The difference between the use of await and get is visible in the edges labelled with $\circledast$, which are only added for await. They capture the traces in which the execution of m waits for the termination of p, and q starts its execution interleaved between $o{:}m_2$ and $o{:}m_3$, postponing the execution of $o{:}m_3$. In this example, the edges labelled with $\circledast$ do not produce new sets in *paths*. ∎

Finally, let us remark that our work is parametric in the underlying points-to and cost analyses for serial execution. Hence, any accuracy improvement in these auxiliary analyses will have an impact on the accuracy of our analysis. In particular, a context-sensitive points-to analysis [15] can lead to big accuracy gains. Context-sensitive points-to analyses use the program point from 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 $o{:}p_1{:}m$ and $o{:}p_2{:}m$. Therefore, instead of representing them by a single node in the graph, we will use two nodes. The advantage of this finer-grained information is that we can be more accurate when considering task parallelism. For instance, we can have one path in the graph which includes a single execution of $o{:}p_1{:}m$ (and none of $o{:}p_2{:}m$). However, if the nodes are merged into a single one, we have to consider either that both or none are executed. There are also techniques to gain precision in points-to analysis in the presence of loops [16] that could improve the precision of our analysis.

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

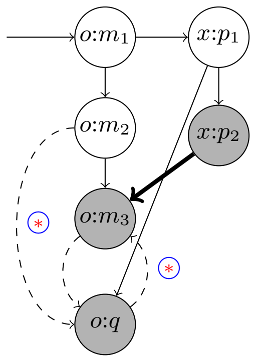


Fig. 5. DCG with synchronization

7 Experimental evaluation

We have implemented our analysis in SACO and applied it to some distributed based systems: BBuffer, the typical bounded-buffer for communicating several producers and consumers; MailServer, which models a distributed mail server system with multiple clients; Chat, which models chat application; DistHT, which implements and uses a distributed hash table; BookShop, which models a web shop client-server application; and P2P, which represents a peer-to-peer network formed by a set of interconnected peers. Experiments have been performed on an Intel Core i7 at 2.0GHz with 8GB of RAM, running Ubuntu 14.04. Table 1 summarizes the results obtained for the benchmarks. Columns Benchmark and loc show, resp., the name and the number of program lines of the benchmark. Columns $\#_{\mathsf{N}}$ and $\#_{\mathsf{E}}$ show the number of nodes and edges of the DFG with concurrency (Def. 5). Columns $\#_{\mathsf{F}}$ and $\#_{\mathsf{P}}$ contain the number of terminal nodes in the DFG and the number of elements in the set *paths*. Columns $\mathbf{T}_{\mathcal{S}}$ and $\mathbf{T}_{\widehat{\mathcal{P}}}$ show, resp., the analysis times for the serial cost analysis and the additional time required by the parallel cost analysis (in milliseconds) to build the DFG graphs and obtain the cost from them. The latter includes a simplification of the DFG to reduce the strongly connected components (SCC) to one node. Such simplification significantly reduces the time in computing the path expressions and we can see that the overall overhead is reasonable.

Column $\%_{\widehat{\mathcal{P}}}$ aims at showing the gain of the parallel cost $\widehat{\mathcal{P}}$ w.r.t. the serial cost $\mathcal{S}$ by evaluating $\widehat{\mathcal{P}}(\bar{e})/\mathcal{S}(\bar{e}) * 100$ for different values of $\bar{e}$. Namely, $\%_{\widehat{\mathcal{P}}}$ is the av-

Benchmark	loc	$\#_{\mathsf{N}}$	$\#_{\mathsf{E}}$	$\#_{\mathsf{F}}$	$\#_{\mathsf{P}}$	$\mathbf{T}_{\mathcal{S}}$	$\mathbf{T}_{\widehat{\mathcal{P}}}$	$\#_{\mathsf{I}}$	$\%_{\mathsf{m}}$	$\%_{\mathsf{a}}$	$\%_{\widehat{\mathcal{P}}}$
BBuffer	105	37	50	7	50	256	26	1000	3.0	19.7	77.4
MailServer	115	28	35	6	36	846	12	1000	61.1	68.6	88.5
Chat	302	84	245	25	476	592	126	625	5.7	56.0	85.4
DistHT	353	38	47	6	124	950	49	625	3.7	25.5	76.3
BookShop	353	60	63	7	68	2183	214	2025	9.2	50.9	71.1
P2P	240	168	533	27	730	84058	1181	512	13.0	85.9	95.2

Table 1. Experimental results (times in ms)

erage of the evaluation of the cost expressions $\widehat{\mathcal{P}}(\bar{e})$ and $\mathcal{S}(\bar{e})$ for different values of the input arguments $\bar{e}$ to the programs. The number of evaluations performed is shown in column $\#_{\mathsf{I}}$. The accuracy gains range from 4.8% in P2P to 28.9% in BookShop. The gain of more than 20% for DistHT, BookShop and BBuffer is explained by the fact that these examples take advantage of parallelism: the different distributed locations execute a similar number of instructions and besides their code mostly runs in parallel. MailServer, Chat and P2P achieve smaller gains because the blocks that are not included in the path (those that are guaranteed to happen in parallel with longer blocks) are non-recursive. Thus, when the number of instructions is increased, the improvements are reduced proportionally. Moreover, Chat and P2P create very dense graphs, and the paths that lead to the maximum cost include almost all nodes of the graph. Column $\%_{\mathsf{m}}$ shows the ratio obtained for the location that achieves the maximal gain w.r.t. the serial cost. In most examples, except in MailServer, such maximal gain is achieved in the location that executes the entry method. MailServer uses synchronization in the entry method that leads to a smaller gain. Column $\%_{\mathsf{a}}$ shows the average of the gains achieved for all locations. The average gain ranges from 80.3% to 31.4%, except for P2P, which has a smaller gain 14.1% due to the density of its graph as mentioned above.

8 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 of the distributed locations. There is recent work on cost analysis for distributed systems which infers the peak of the *serial* cost [3], 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 it 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 [3]. The only common part 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. This work is improved in [4] 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 [10]. This work differs from our approach 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.

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., [8,9,21]). 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 would be simulated in our case by using a get instruction on all spawned tasks). As future work, we plan to incorporate in the analysis information about the scheduling policy used by 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 [7] and its references). In this sense, our DFG's provide a finite representation of all traces that may arise in the distributed system. A multithread 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 limits is gaining attention both for distributed [11] and for multicore systems [6,20], because the amount of interleaving between tasks that must be considered in analyses is restricted to synchronization points which are explicit in the program.

References

1. WCET tools. `http://www.rapitasystems.com/WCET-Tools`, 2012.
2. 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.
3. 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.
4. 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.

5. 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.
6. S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. Broch Johnsen, Ka I Pun, S. Lizeth Tapia Tarifa, T. Wrigstad, and A. Mingkun 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.
7. A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In *POPL*, pages 129–142. ACM, 2013.
8. 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.
9. J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *Proc. of POPL'11*, pages 357–370. ACM, 2011.
10. J. Hoffmann and Z. Shao. Automatic Static Cost Analysis for Parallel Programs. In *Procs. of ESOP'15*, volume 9032 of *LNCS*, pages 132–157. Springer, 2015.
11. E. Broch 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.
12. J. K. Lee, J. Palsberg, and R. Majumdar. Complexity results for may-happen-in-parallel analysis. Manuscript, 2010.
13. 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.
14. 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.
15. 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.
16. M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.
17. Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.
18. R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, July 1981.
19. B. Wegbreit. Mechanical Program Analysis. *Communications ACM*, 18(9):528–539, 1975.
20. 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.
21. 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.