Turning Nondeterminism into Parallelism

Omer Tripp
Tel Aviv University
omertrip@post.tau.ac.il

Eric Koskinen
New York University
ejk@cims.nyu.edu

Mooly Sagiv
Tel Aviv University
msagiv@post.tau.ac.il

Abstract

Nondeterminism is a useful and prevalent concept in the design and implementation of software systems. An important property of nondeterminism is its latent parallelism: A nondeterministic action can evaluate to multiple behaviors. If at least one of these behaviors does not conflict with concurrent tasks, then there is an admissible execution of the action in parallel with these tasks. Unfortunately, existing implementations of the atomic paradigm—optimistic as well as pessimistic—are unable to fully exhaust the parallelism potential of nondeterministic actions, lacking the means to guide concurrent tasks toward nondeterministic choices that minimize interference.

This paper investigates the problem of utilizing parallelism due to nondeterminism. We observe that nondeterminism occurs in many real-world codes. We explain why existing approaches fail to exploit this source of parallelism. Namely, what is missing is a communication mechanism that enables nondeterministic tasks to coordinate their simultaneous execution effectively. This limits the ability by both speculative and lock-based synchronization to exhaust parallelism. We have developed a system that features the needed coordination facilities, allowing tasks to look into the future of other tasks and reduce conflict accordingly. We evaluate our system on a suite of 12 challenging algorithmic benchmarks of wide applicability, as well as a large-scale commercial application. The results are encouraging.

1. Introduction

Nondeterminism arises in the sequential specification of many algorithms and software applications. Examples include graph algorithms, such as obtaining some minimum spanning tree (MST) from an input graph or finding some shortest path between two nodes; search techniques, whose objective is to arrive at some admissible goal state (as in the n-queens and combinatorial-assignment problems [22]); learning strategies, such as converging on some classifier that labels all data instances correctly [58]; and more generally, any algorithm for finding a minimal or maximal subset of a collection of values which satisfies some property, or finding a representative value satisfying the property.

Recent research has explored the connection between nondeterminism and parallelism at the specification level [14, 15], reducing the correctness of a parallel program to checking whether every result it produces is also possible under a nondeterministic sequential version of the program. There is, however, latent nondeterminism even in purely deterministic programs with only one admissible result.

The following simple example, in Java syntax, illustrates this observation:

```
/* global */ Set elems = ...;

task 1: @atomic {                    ||  task 2: @atomic {
  Iterator it=elems.iterator();      ||    Object o2 = ...;
  Object o1=it().next();             ||    elems.remove(o2);
  // assumes: o1 ∈ elems             ||    // assumes: o2 ∉ elems
  ... }                              ||    ... }
```

Task 1 above has latent nondeterminism: The Java specification for iterating over a `Set` object places no guarantees on iteration order. This permits task 1 to choose an object that differs from (the object pointed-to by) `o2`, thereby avoiding conflict with task 2.

More generally, when there are multiple valid executions of a given task, arising from nondeterminism, one might coerce that task to choose the executions that have the least amount of conflict with concurrent tasks (*e.g.* executions that commute with those tasks). This provides an opportunity for increased parallelism.

Unfortunately, current implementations of the `atomic` construct are unable to fully exhaust this source of parallelism. Existing techniques cannot let the two tasks proceed concurrently while guaranteeing absence of conflicts.

Optimistic synchronization approaches—most notably, transactional memory (TM) [34]—handle potential conflicts through the *history* of the execution—what has occurred thus far—where utilizing nondeterminism requires foresight. In the above example, a standard optimistic protocol would let task 1 make a free choice which object to read from `elems`, aborting due to a read/write conflict if that object is `o2`. The conflict is semantic, and thus abstract-level synchronization [33, 41, 48], which was recently proposed as a means of reducing conflict, would still force an abort.

Meanwhile, techniques for pessimistic synchronization, including recent proposals featuring user interaction and client-driven lock inference [16, 43, 46], make conservative forecasts about the future behavior of concurrent tasks for lack of appropriate communication mechanisms, which disables the available parallelism due to nondeterminism. Because `task 1` *may* read any object stored in `elems` (and in particular `o2`), such synchronization would enforce ownership of the entire `elems` set by `task 1`. This would effectively serialize the execution of the two tasks.

Existing solutions do not utilize the observation that the tasks can coordinate access to different objects, thereby avoiding conflict. Acting upon this observation requires a communication mechanism that lets tasks broadcast their intentions (*i.e.* may-modify information), such that other tasks can specialize their nondeterministic choices. For example, if `task 1` is aware of `task 2`'s intention to remove `o2` from `elems`, then it can read another object.

Scope We investigate the problem of exploiting parallelism resulting from nondeterminism in the presence of mutation operations, where synchronization is required. We have found that the key is a concurrency paradigm in which conflict is based on the *future*, rather than the *past*. In our model, tasks share information about the behaviors they may exhibit in the future, and then *specialize* themselves (*i.e.* choose from available nondeterminsitic options) to reduce conflict.

In practice, this is achieved by letting each task view only the portion of the shared state that is invariant over the effects of concurrent tasks, and make its choices based on this partial view. In the example above, for instance, the available view for `task 1` does not contain (the object pointed-to by) `o2`. Later, in Sections 3–4, we discuss the assumptions governing this synchronization approach, as well as the conditions under which progress is guaranteed.

We have realized our approach in the TANGO synchronization protocol, which we implemented as a software tool for loop parallelization. TANGO is a pessimistic protocol, ensuring deadlock-free, serializable parallel execution. TANGO enables efficient runtime task specialization thanks to compile-time analysis of task intentions (*e.g.* the intention of `task 2` to remove `o2`). As is the case with other approaches [9], TANGO is best suited for the broad class of applications where task intentions are "predictable" and constrained to a small portion of the shared state [26]. This lets concurrent operations view and act on most of the shared state already at an early point, enabling a high level of parallelism.

We have evaluated the performance characteristics of TANGO in two experiments. In the first, we applied TANGO to a suite of 12 loop-based algorithms that are considered challenging to parallelize. The results are encouraging, demonstrating that TANGO is able to leverage latent nondeterminism, and is in fact comparable in performance to parallelization schemes that were tailored specifically for some of our benchmarks. In the second study, we applied TANGO to a commercial security testing tool. This required few, relatively simple code transformations, yielding speedups of up to 3.2x on challenging loops in the tool's code.

Contributions We make the following contributions:

Nondeterminism and Specialization We take a first step in developing a concurrent programming paradigm that turns latent nondeterminism into a source of parallelism. In our approach, conflict is based on potential future behaviors, and threads make nondeterminsitic choices that avoid conflict (Section 4). In particular, threads may specialize both their view of the shared state (*e.g.* reads) and their modifications to the shared state (*e.g.* writes).

Synchronization Protocol We have developed a general synchronization protocol that enforces the available parallelism due to nondeterminism in a pessimistic manner (Section 5). Our protocol generalizes several recent parallelization schemes that were tailored for specific applications, and achieves comparable performance on these applications.

Greedy Algorithms We describe experiments on a suite of 12 benchmarks, 8 of which are greedy algorithms (Section 6.1). These experiments, beyond measuring the performance of our protocol, enable understanding of the available parallelism in greedy algorithms according to the way in which a greedy step depends on the previous steps.

Implementation and Evaluation We have implemented our protocol as a tool for loop parallelization, available both in Java and in .NET. We present experiments over the tool (Section 6) that consists of (i) comparing it with several other synchronization techniques on the 12 algorithmic benchmarks (Section 6.1), and (ii) applying it to a major component in a commercial product (Section 6.2). The results of both experiments are encouraging.

2. Running Example

We illustrate our approach with the REVERSE DELETE algorithm, whose description—using LINQ syntax [4]—appears in Figure 1. (For now, ignore the `Monotonic` attribute.) REVERSE DELETE computes a minimum spanning tree (MST) from a connected, edge-weighted graph. It does so in a greedy style. At each step, the algorithm considers the next heaviest edge, attempting to find a path between its incident nodes assuming the edge is discarded. If the search fails, then the speculative edge removal is reverted seeing that the edge is essential to preserve connectivity.

This program is difficult to parallelize for at least two reasons. First, it imposes ordering constraints: The input edges are traversed according to a total order (from the heaviest edge to the lightest one). Second, side effects on the edge set cause loop-carried dependencies that must be

```
var sortedEdges = edges.OrderBy(x => -x.Weight);
[Monotonic(FindPath)]
foreach (var edge in sortedEdges) {
  edges.Remove(edge);
  if (!FindPath(edge.U, edge.V)) {
    edges.Insert(edge); } }
```

Figure 1. The REVERSE DELETE algorithm in LINQ syntax, annotated for parallelization under TANGO

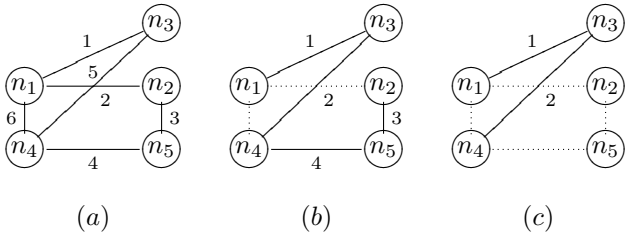

Figure 2. Illustrative (weighted, undirected) input graph for the REVERSE DELETE program listed in Figure 1 (a); a view of this graph excluding edges (n_1, n_2) and (n_1, n_4) (which is also its MST) (b); and another view where only the two lightest edges are kept (c)

preserved. If the i-th iteration deletes its designated edge, then iteration $i + 1$ must account for this deletion when searching for a path between its corresponding nodes.

However, closer inspection of the semantics of this program suggests that there is a lot of available parallelism on certain classes of input graphs. The `FindPath` call is *nondeterministic*: Any path returned by this call can serve as a witness for the deletion of the corresponding edge, provided that the edges comprising this path are not deleted by earlier iterations. Thus, the existence of multiple (nontrivial) paths between a pair of connected nodes is a source of parallelism, allowing several loop iterations to run in parallel if each has available a witness path that is invariant over edge deletions by other iterations.

To illustrate this, we refer to graph (a) in Figure 2. In this graph, the deletion of edge (n_1, n_2) can be executed in parallel to deleting (n_1, n_4) if the supporting path for the deletion of (n_1, n_2) is $[n_1, n_3, n_4, n_5, n_2]$. Path $[n_1, n_4, n_5, n_2]$ cannot, however, support the deletion of (n_1, n_2), because deleting edge (n_1, n_4) invalidates this path.

3. Our Approach

In this section, we describe and illustrate the main features of our approach, and discuss its scope and limitations.

3.1 Exploiting Parallelism due to Nondeterminism

We take a first step in developing synchronization methods that are aware of latent nondeterminism and exploit it to increase parallelism. We have identified a prevalent loop coding pattern, appearing *e.g.* in greedy algorithms, whereby each iteration tests the program state for some property, and then—based on the result—decides which mutation operation to apply (if at all). In REVERSE DELETE, for example, the iteration checks for connectivity between a pair of previously connected nodes. If the answer is negative, then the nodes' incident edge is restored into the graph. In certain cases, such as the `FindPath` query, the tested property has available nondeterminism.

We have designed a general protocol, TANGO, that realizes parallelism due to nondeterminism through *specialization*. Intuitively, specialization is the idea of restricting the choices of a nondeterministic command, so that it becomes more commutative with concurrent tasks, thereby enabling more parallelism.

Specifically, TANGO implements the notion of specialization over nondeterministic commmands that are also *monotonic*.[1] Indeed, nondeterministic property tests are often also monotonic, in that successful evaluation of the test on a substate implies that the test holds over the entire state. This is the case with REVERSE DELETE, for instance, where if the `FindPath` test succeeds over graph G, and graph G' contains G, then `FindPath` is guaranteed to succeed also over G'.

Roughly speaking, monotonicity permits serializable evaluation of a command on a substate. Focusing the command on a substate, rather than the entire state, restricts its (nondeterministic) behavior, thereby achieving specialization. TANGO's choice of which substate to evaluate the task on is guided by the effects of concurrent tasks. That substate is the portion of the shared state that is disjoint from the effects of other tasks. In REVERSE DELETE, for example, restricting the path test to a subgraph that does not include an edge e does not interfere with the deletion of e.

Projecting this discussion onto the example of graph (a) in Figure 2, we obtain correct parallel execution of the first two iterations, with respective edges (n_1, n_4) and (n_1, n_2), if the second iteration performs connectivity testing over a view of the graph that is *disjoint* from the possible effects of the first iteration; namely, without edge (n_1, n_4). The resulting view is shown as graph (b) in Figure 2 (which coincides with the MST of the graph in (a)). Thanks to *monotonicity*, the success of the test, with witness path $[n_1, n_3, n_4, n_5, n_2]$, is serializable.

3.2 Preliminaries: Guards, Methods and Relations

The coding idiom motivating our concrete protocol, where each loop iteration performs a test and then acts on its result, encourages language-level abstractions that separate tests from actions. Dijkstra has formulated this distinction in his language of Guarded Commands in terms of guards

[1] We refer to monotonicity in the mathematical sense, whereby a function f between a pair of ordered sets is monotonic if it preserves the given order; *i.e.* $\forall x, y.\ x < y \Rightarrow f(x) < f(y)$.

(being boolean queries) and methods (being state transformers) [21], which we adopt also for our setting. As an illustration, in Figure 1 we denote the guard with dotted underline, and the methods with solid underline. We assume that guards are pure (*i.e.* free of side effects).

In our technical description, as well as our implementation, we instantiate our conceptual approach into a relational setting, whereby the shared state between loop iterations is represented and manipulated as relations. This style of data organization is familiar from databases, and has been utilized by several recent works [30, 31, 55]. The key advantage of the relational form is in abstracting data from its representation, which permits a clear, high-level description of what guards and methods do. This is also the rationale behind recent language features, such as the LINQ queries in C# [4]. In fact, our prototype implementation is based on the LINQ language constructs, which offer a convenient means of expressing guards.

The relational abstraction consists of tuples and relations. A tuple $t = \langle c_1 \colon v_1, c_2 \colon v_2, \ldots \rangle$ maps a set of columns, $\{c_1, c_2, \ldots\}$, to values. A relation is a set of tuples over identical columns. We support the standard relational operations, which we describe below in ML-like syntax (where $!r$ fetches the current value of r, and $r \leftarrow v$ denotes assignment):

$$\begin{aligned} \mathsf{empty}\ r &= r \leftarrow \mathsf{ref}\ \emptyset \\ \mathsf{insert}\ r\ t &= r \leftarrow\ !r \cup \{t\} \\ \mathsf{remove}\ r\ t &= r \leftarrow\ !r \setminus \{t\} \\ \mathsf{query}\ r\ \phi\ b &= b \leftarrow \phi(r) \end{aligned}$$

Informally, $\mathsf{empty}\ r$ creates a fresh relation r; $\mathsf{insert}\ r\ t$ inserts tuple t into relation r; $\mathsf{remove}\ r\ t$ removes tuple t from r; and $\mathsf{query}\ r\ \phi\ b$ tests relation r for property ϕ and stores the (boolean) result in b. For simplicity, we leave the syntax for ϕ unspecified. This level of detail is not required for our technical description, and our prototype tool features the built-in LINQ querying constructs.

Revisiting our running example in Figure 1, we can express the shared state as the edges binary relation. `Remove` and `Insert` are modeled directly by their relational counterparts, remove and insert, and `FindPath(`u`,`v`)` is translated into the query $(u, v) \in \mathsf{tc}((u, \_), \mathsf{edges})$, where tc denotes transitive closure, and the entire query checks whether tuple (u, v) is in the transitive closure of the tuples t such that $\mathsf{fst}(t) = u$ over the edges relation.

3.3 The TANGO Protocol: Informal Sketch

The concrete TANGO protocol leverages the observations discussed in Section 3.1 as follows:[2]

User Specification The distinction between guards and methods is made explicit thanks to the relational setting, as explained in Section 3.2. For our current prototype, we ask the user to decorate monotonic guards with the `Monotonic` attribute, as exemplified in Figure 1. This attribute is likely amenable to automatic inference, but we leave this enhancement for future versions of our tool.

Compile-time Analysis At compile time, a lightweight static analysis computes a safe approximation of the effects of each iteration. Specifically, for loops iterating over sequences, the analysis tries to link these effects to a symbolic representation of the element pointed-to by the loop variable (the `edge` variable in the loop in Figure 1).

Runtime Synchronization At runtime, during parallel loop execution, the TANGO synchronization protocol automatically builds "safe" views for monotonic guards, as illustrated with graphs (a) and (b) in Figure 2. The view is constructed according to the possible effects of higher-importance concurrent tasks (*e.g.* earlier iterations, as in REVERSE DELETE). If a monotonic guard evaluates positively based on a limited view of the entire state, then parallel progress has been achieved. Otherwise, there are two options: If the guard fails, but all higher-importance tasks have completed already, then the guard result is serializable. Otherwise, view construction is retried per the observation that the possible effects of parallel tasks only decrease with time (becoming empty when a task completes), permitting a more complete view on retry.

A remaining challenge, which we have yet to discuss, is support for parallel method execution (like the `edges.Remove` and `edges.Insert` calls in Figure 1). If, for instance, the second iteration deletes edge (n_1, n_2) while the first iteration is still running, then parallel execution is not necessarily serializable, since there is no sequential run where the first iteration observes a state where edge (n_1, n_2) is absent.

To deal with this challenge, TANGO follows an approach that is similar in spirit to *flat combining* [32]. Instead of directly applying a method to the shared state, TANGO records the method application to a task-private log. Logged methods are accounted for in views constructed for the task (*e.g.* the deletion of (n_1, n_2) in graph (b) in Figure 1). This permits a task to proceed through methods without blocking.

Moreover, if task t is about to complete, but concurrent tasks that must not observe its effects are still running, then instead of either committing t's private log (which is wrong) or blocking t until these tasks finish (which is inefficient), TANGO appends t's log of outstanding methods to that of a running task, t', where t' is the live task that immediately succeeds t in importance. t can then terminate, and t' becomes responsible for executing t's methods, ensuring that these are applied to the shared state in a serializable order.

3.4 Discussion: Scope and Limitations

Our approach generalizes several recent efforts to parallelize specific algorithms, including

[2] In our discussion here, and in general, we refer to tasks and loop iterations interchangeably, where what we mean by a task is the atomic parallel execution of a single iteration.

- the *helper-threads* (HT) scheme [38, 49], where auxiliary threads offload work from the main thread by performing computations ahead of time, in parallel, either pessimistically [38] or speculatively (HT+TM) [49]; and
- the *dynamic-independence* (PMS) approach [12], which exploits the observation that in certain greedy algorithms, such as the algorithm for finding a maximal independent set (MIS), each iteration only depends on a subset of the previous iterations to run a task ahead of its designated time, as soon as the tasks it depends on have completed.

Both HT and PMS enforce pessimistic parallelization, letting background threads execute computations ahead of time when doing so is provably serializable. TANGO generalizes these two approaches in enforcing weaker preconditions for correct parallelization. From a practical standpoint, TANGO asks only for monotonicity annotations, and obtains comparable speedups to HT and PMS on the benchmarks motivating these approaches.

Runtime Overhead The main challenge faced by TANGO is to constrain the cost of constructing task-specific views. The practical limitation this places is that the effects of a task must have a compact description. This is true of REVERSE DELETE, where the substate manipulated by a given loop iteration is a single edge. TANGO incurs tolerable overheads over the 12 algorithmic benchmarks from our suite, the worst case being a factor of 1.35x compared to a sequential version of the benchmark (*cf.* Table 2).

Applicability As our analysis so far suggests, TANGO is geared toward computations where guards are nondeterministic with multiple admissible witnesses, and the intentions of a task refer to a small portion of the shared state. Two broad categories of applications satisfying this profile are

- *pruning algorithms*, such as REVERSE DELETE, the alpha-beta search algorithm [51] and decision-tree minimization [45], where a witness contained in a modification-free substate secures early deletion of an element; and dually,
- *incremental algorithms*, such as KRUSKAL MSF [19], task scheduling [19] and quick elimination [6], where a witness contained in a past version of the evolving solution secures early exclusion of an element.

Our experiments (described in Section 6), where we consider benchmarks of varying characteristics, indicate that TANGO is highly effective in parallelizing applications with nondeterministic reads and granular intentions, like KRUSKAL MSF and REVERSE DELETE, and is also able to utilize some of the available parallelism in codes with deterministic reads, such as GREEDY COLORING and the DP algorithms. However, algorithms where the intentions of a task are initially broad, and undergo refinement only at a late point, like DIJKSTRA SP and JARVIS MARCH, are not good candidates for TANGO.

4. Concurrent Tasks, Specialization, Priority

In this section, we provide a complete technical description of our approach. We begin by stating a restrictive requirement for serializable execution of an operation in parallel with concurrent tasks, whereby that operation must commute with all possible future behaviors of the tasks. We then introduce specialization as a means of turning this restriction into an advantage: Instead of requiring commutativity over *all* behaviors, concurrent tasks can select specific behaviors that are mutually compatible and commit to these behaviors. Finally, we introduce a notion of priority, and show that this leads to a progress guarantee.

4.1 Execution Model

We assume a generic programming model with standard control structures (`if`, `while`, sequential composition, *etc*). There is a single shared state from an unspecified state space Σ. There are a finite number of threads, each with a unique identifier $\tau : \mathcal{T}$, that access the shared state in only two ways:

- *Guard* $g : \Sigma \to \{\mathsf{true}, \mathsf{false}\}$ is an observation that a thread makes on the shared state.
- *Method* $m : \Sigma \to \mathcal{P}(\Sigma)$ is a modification that a thread makes on the shared state.

Guards and methods can be realized in many ways, depending on the particular language context. The most natural example is Dijkstra's language of Guarded Commands [21]. Another example is a language that performs low-level memory operations, where `read` is a guard and `write` is a method. As discussed in Section 3.2, for this paper we have chosen a relational program representation, whereby the shared state is expressed as one or more relations, guards are boolean query commands, and the methods are insert and remove.

When tasks are executed concurrently without synchronization, they do not always behave correctly. There may be executions in which one task interferes with another task.

Example 4.1. *Assume an unsychronized parallel run of the code in Figure 1 on graph (a) in Figure 2, where task* (n_2, n_5) *is the first to execute. That task would then delete edge* (n_2, n_5) *based on witness* $[n_2, n_1, n_4, n_5]$*, which is invalid, because in sequential execution of the code, both edge* (n_1, n_4) *and edge* (n_1, n_2) *would be deleted, rendering edge* (n_2, n_5) *part of the MST.*

For safe concurrency, tasks must be aware of the potential effects of concurrent tasks, as well as their assumptions on the shared state. As is standard, we use the notion of *commutativity* to quantify interference [39, 40]:

$$m \bowtie_\sigma m' \iff [\![m\,;\,m']\!]\sigma \;=\; [\![m'\,;\,m]\!]\sigma$$

If the commands of two threads (pairwise) commute, then it is safe for them to proceed in parallel. However, our work

takes a new tack: Our notion of conflict is based on what may happen in the future rather than what has happened in the past. To this end, we add two criteria:

Condition 4.1. *A method m performed by a task τ must not change the boolean value of every guard g that may ever be observed by a concurrently executing task τ'.*

Condition 4.2. *A method m performed by a task τ must* commute *with every method m' that may ever be performed by a concurrently executing task τ'.*

If Conditions 4.1 and 4.2 hold, then every execution of the concurrent tasks is serializable.

These two conditions are strong and thus somewhat surprising. We are requiring that a method commutes with every method/guard that any other task *may* ever perform. This is stronger than the pairwise synchronization conditions that hold over histories in traditional systems such as TM or lock-based mutual exclusion.

Example 4.2. *For the code in Figure 1 and graph (a) from Figure 2, the* `FindPath` *guard over (n_1, n_2) does not commute with the* `edges.Remove` *method over (n_1, n_4) because of witness path $[n_1, n_4, n_5, n_2]$. Parallel execution of these two commands is possible, however, assuming the (n_1, n_2) guard can only select witness $[n_1, n_3, n_4, n_5, n_2]$.*

Our important insight, which we formulate in the following, is that the strict synchronization model due to Conditions 4.1 and 4.2 makes potential conflicts explicit. This lets tasks coordinate their present behaviors in order to *avoid* as much conflict as possible in the future. For this reason, as we show in Section 6, our approach is in fact competitive with many leading-edge concurrent programming algorithms.

4.2 Specialization and Preservation

Nondeterminism complicates commutativity judgments, because there are multiple possible behaviors for a task. Concurrent tasks can only take a next step if that step commutes with all of these behaviors. This is seen in Conditions 4.1 and 4.2, where m must commute with *all* possible commands that *may* be executed by concurrent tasks.

We turn this limitation of synchronizing nondeterministic tasks into an advantage through specialization: A task specializes by restricting its possible behaviors, which increases concurrency by relaxing the commutativity checks. We define specialization for both methods and guards as follows:

Definition 4.1 (Specialization)**.** *Specialization of methods $\preceq_M$ and of guards $\preceq_G$ is as follows:*

$$m_1 \preceq_M m_2 \quad \equiv \quad \forall\sigma.\, [\![m_1]\!]\sigma \subseteq [\![m_2]\!]\sigma$$
$$g_1 \preceq_G g_2 \quad \equiv \quad \forall\sigma.\, [\![g_1]\!]\sigma \Rightarrow [\![g_2]\!]\sigma.$$

We will often drop the subscript on $\preceq$ as it is clear from context. For the definition of dynamic specialization over a specific state σ, we simply remove the universal quantification.

Above a *specialized command* is a state transformer that may have less nondeterminism than the original command. If a *specialized guard* holds, then the original guard must hold, but the specialization may have a particular implementation that, for example, is focused on a portion of the state space.

Example 4.3. *For the setting described in Example 4.2, if we specialize the* `FindPath` *guard over (n_1, n_2) by restricting its evaluation to edges that are invariant over higher-priority tasks (namely, by excluding (n_1, n_4), as illustrated in graph (b) in Figure 2), then we obtain witness path $[n_1, n_3, n_4, n_5, n_2]$. This witness implies commutativity between the guard and the* `edges.Remove` *command over edge (n_1, n_4).*

The advantage with specialization is that a task can make an informed choice of which behaviors to eliminate and which behaivors to retain based on the behaviors of concurrent tasks. This permits inter-task coordination, where nondeterminism allows tasks to choose behaviors that are mutually compatible, thereby enabling safe parallel progress.

Theorem 4.3 (Preservation)**.** *Method specialization, as well as specialization of monotonic guards, preserve the behaviors and serializability of the original system.*

Proof Sketch. For a method, specialization yields less post-states than the original method, but these are all admissible due to containment. For monotonic guards, successful evaluation on a substate implies successful evaluation on the entire state thanks to monotonicity. Local preservation of behaviors and serializability implies a system that is globally serializable. □

Of course we have to be careful when we specialize that we have not performed trivial specialization, simply eliminating all behaviors. We address this concern in the remainder of this section, where we show that if tasks are ordered by priority, then at each point, at least one of the tasks (specifically, the highest-importance one) is free of specialization obligations, obviating the threat that all tasks specialize trivially, thereby blocking execution.

4.3 Priority and Progress

While specialization—in conjunction with commutativity—guarantees preservation, progress is still not ensured. The difficulty is in verifying that at least one task can step through a guard, which is not necessarily the case unless the tasks are assigned priorities.

Example 4.4. *Assume a parallel run of* REVERSE DELETE *on graph (a) in Figure 2, where the first two iterations have completed, and now the third and fourth iterations—with respective edges (n_4, n_5) and (n_2, n_5)—simultaneously specialize, resulting in view (c). This is a stuck state, because for both iterations, (i) the* `FindPath` *guard fails, and (ii) the view is partial, disabling progress through the guard when its evaluation is negative.*

In response to this problem of nonterminating executions, we introduce task priorities. We show that with this strengthening, tasks can execute in parallel while ensuring both progress and preservation. Intuitively, the power of prioritization is in breaking the symmetry between tasks, such that a higher-priority task need not account for the behavior of lower-priority tasks. This yields a progress guarantee.

The rules we have presented thus far provide a serializable concurrent model in which tasks specialize their nondeterministic choices in order to proceed in concert. We now revise Conditions 4.1 and 4.2, obtaining new conditions that guarantee progress (provided the tasks terminate sequentially).

We first define priority:

Definition 4.2 (Priority)**.** *Let* $\ll : (\mathcal{T} \times \mathcal{T})$ *be a total order on task identifiers.*

We can now state the two conditions.

Condition 4.4. *A method* m *performed by task* τ *must not change the boolean value of a guard* g *that was observed by another task* τ' *such that* $\tau' \gg \tau$.

Condition 4.5. *A method* m *performed by task* τ *must commute with every method* m' *that may ever be performed by another task* τ' *such that* $\tau' \gg \tau$.

The formal detail of these conditions involves quantifying over all possible executions, and for each execution, performing a pairwise check of the moverness of m against methods and guards of other concurrent tasks of higher priority Intuitively, priority manifests as a discount: It lets a task ignore both the assumptions (*i.e.* guards; Condition 4.4) and the effects (*i.e.* methods; Condition 4.5) of lesser-priority tasks, which means that at each point, at least one task can progress without constraints.

Example 4.5. *We return to Example 4.4 and fix it per Conditions 4.4 and 4.5. The third iteration (with respective edge* (n_4, n_5)*) is more important than the fourth one (with edge* (n_2, n_5)*), which means that it can ignore the effects of the fourth iteration. Thus, the graph view by the third iteration after the first two iterations have terminated is complete, coinciding with the actual graph state. This permits progress through the* `FindPath` *guard in spite of its negative evaluation. The same situation arises for the fourth iteration after the third iteration completes.*

Theorem 4.6 (Progress)**.** *For tasks* $\tau_1 \ldots \tau_n$ *that each terminate in a sequential setting and priority relation* $\ll$*, either (i) all tasks have completed, or else (ii) there exists a task* τ_i *that can perform a method or a guard.*

Proof Sketch. The highest-priority task (denoted τ_i) can proceed without waiting for other tasks. Condition 4.4 ensures that the values of all guards of the highest-priority task are not altered by concurrent tasks. Condition 4.5 ensures that if another task $\tau_j \ll \tau_i$ executes a method m concurrently, then m commutes with all the operations of τ_i. Thus, there is necessarily an equivalent serial history in which τ_i happens before τ_j. □

5. The TANGO Protocol

We now describe the TANGO synchronization protocol, which instantiates the formal framework developed in Section 4.

5.1 Intention Analysis

In the concrete TANGO protocol, we model the effects of a task as the substates it may modify. This assumes that the state is decomposable into substates: $\sigma = \bigcup_i s_i$ for atomic substates s_i, with the assumption that the state space Σ is closed under substates. For a concrete state, the standard decomposition is into memory locations (*e.g.* variables and object fields), the (low-level) state description being a mapping from memory locations to their associated value [56].

In TANGO, the shared state is expressed as relations, and so state decomposition is in terms of qualified tuples. The possible effects of the two simple methods are straightforward, as follows:

$$\mathsf{mod}(\mathsf{insert}\ r\ t) = \quad \langle !r, t\rangle$$
$$\mathsf{mod}(\mathsf{remove}\ r\ t) = \quad \langle !r, t\rangle$$

To induce correct task-centric state views, containing only portions of the shared state that are invariant over the effects of concurrent tasks, the runtime system must be aware of what these tasks may do. Computing the effects of a task on the fly, during a parallel run, is expensive. Instead, TANGO features a lightweight static data-flow analysis, tailored for loop structures.

The analysis is performed at compile time, and is a forward analysis starting at the loop head. When the analysis encounters a method, it stores its affected substate as a symbolic access path (*e.g.* $\langle$`edges`, `edge`$\rangle$ for the `Remove` and `Insert` statements in Figure 1, modeled as remove and insert respectively).

After the analysis reaches a fixpoint over the control structure of the loop body, each collected access path $\langle r, t\rangle$ is processed as follows:

1. If $\langle r, t\rangle$ is a constant access path (*i.e.* both the relation and the tuple are amenable to static resolution), then it flows into the may-modify set as is.
2. Otherwise, if the loop iterates over a sequence—and therefore the loop variable points to a tuple—and $\langle r, t\rangle$ is identical to the (symbolic) tuple pointed-to by the loop variable, then again $\langle r, t\rangle$ flows into the may-modify set directly. This is the case *e.g.* in Figure 1 with $\langle$`edges`, `edge`$\rangle$. The runtime system can resolve symbolic tuples of this sort before the loop starts executing based on the elements in the argument sequence. Hence the relevance of this case.

3. A third situation is where the tuple as a whole is unavailable, but the analysis can statically resolve the relation r. In this case, the abstract tuple $\langle r, * \rangle$ flows into the may-modify set, indicating that a given iteration may mutate any tuple in relation r.
4. The final case is where neither the relation nor the tuple are amenable to static resolution. This translates into $\top$, meaning that an iteration may modify any tuple in any relation.

The results of the compile-time intention analysis are made available to the runtime system as a may-modify oracle, mod, mapping tasks to symbolic tuple descriptions.

5.2 Runtime Protocol

The runtime TANGO protocol is described in Figure 3 according to the operations it synchronizes: method execution (EXECUTEMETHOD), task termination (EXIT) and guard evaluation (TESTGUARD). To avoid clutter, we separate the description into two context. The first context is where the argument task τ ranks first in priority. In this case, methods and guards are evaluated directly over the shared state. On exit, the task applies the outstanding methods in its log (due to other tasks) to the shared state.

In the complementary context where τ is not the highest-priority task, methods are recorded into the task-private log, rather than performed. When the task is about to terminate, it delegates its outstanding log to the task succeeding it in priority. Evaluation of a general guard must block until the task becomes first in priority. A monotonic guard, however, is amenable to premature evaluation over the portion of the shared state that is invariant over concurrent tasks ($\sigma \setminus \bigcup_{\tau' \gg \tau} \mathsf{mod}(\tau')$), where we account for logged methods ($[\![\mathsf{log}(\tau)]\!]\sigma'$).

Theorem 5.1 (Correctness)**.** *The* TANGO *protocol ensures both progress and preservation per Theorems 4.3 and 4.6.*

Proof Sketch. For guard evaluation, the only situation where a guard is tested outside its sequential order is if that guard is monotonic. In that case, the guard is evaluated over an invariant substate, and progress is conditioned on its successful evaluation. This means that the guard result is serializable.

For method execution, the highest-priority task performs its methods in their sequential order. Other tasks log their operations, rather than performing them directly, and the log is manipulated according to sequential constraints, combining logs on task exit with the task succeeding it in importance. Finally, when a task becomes first in priority ($\tau \neq \max \mathcal{T} \dashrightarrow \tau = \max \mathcal{T}$), it commits its own methods, $\mathsf{log}(\tau)|_\tau$ (but not those combined into its log from other tasks), to enable direct execution of its future methods in a serializable fashion, as specified in Figure 3.

Progress is ensured thanks to task priorities. (See Theorem 4.6.) The most important task is oblivious to other tasks, and so if all tasks terminate sequentially, then all tasks are also guaranteed to complete when running in parallel. □

5.3 Prototype Implementation

TANGO is implemented both in Java and in C# (.NET). The TANGO library provides a parameterized implementation of the `Relation` abstract data type (ADT), featuring efficient linearizable implementations of the primitive relational operations. TANGO also provides common specializations of the `Relation` type (`UnaryRelation`, `BinaryRelation`, *etc*), as well as a collection of useful guards (including transitive reachability, as in `FindPath`, and existential tests). Other ADTs, such as `Map`, `Set`, `List` and `Graph`, are implemented atop the `Relation` ADT. As stated earlier, the C# version of TANGO is built over the LINQ syntax features, enabling concise coding of loops with side effects, as exemplified in Figure 1. Finally, TANGO's algorithm for statically approximating task intentions makes use of the WALA framework [5].

6. Experimental Evaluation

In this section, we describe two sets of experiments that we conducted over TANGO. The first experiment compares TANGO with several specialized parallelization schemes on 12 algorithmic benchmarks. We then present a case study, where we investigate the applicability of TANGO to a real-world, commercial software application.

6.1 Algorithmic Benchmarks

Our benchmark suite for the first experiment consists of 12 benchmarks: 8 greedy algorithms and 4 dynamic programming (DP) algorithms. The main characteristics of the benchmarks are summarized in Table 1. Our choice of benchmarks was driven by two considerations: First, the chosen algorithms exhibit varying levels of available parallelism, which enables proper investigation of when and why our approach would work. This is also the reason why for all the greedy benchmarks we used two different input profiles, one permitting more parallelism than the other. Second, all the benchmarks in our suite are popular, being of practical interest and applicability. Some of the benchmarks have also been parallelized in the past, which enables comparing our approach with other techniques.

Experimental Setting The experiments were conducted on a 16-core IBM X3550 M2 machine with 100GB of RAM running Windows Server 2008 R2 64-bit with Microsoft .NET Framework v4.0.30319. Inputs were generated randomly (random graphs, random sets of 2D points, *etc*). Each configuration of benchmark, input profile and parallelization technique was run 11 times. The reported numbers are the average of the 10 last executions (excluding the first, cold run).

In addition to TANGO and the HT and PMS approaches, presented in Section 3, we included the following parallelization techniques in our evaluation:

Top-down Parallelization (TD) [54] This scheme parallelizes DP algorithms in a top-down manner, where so-

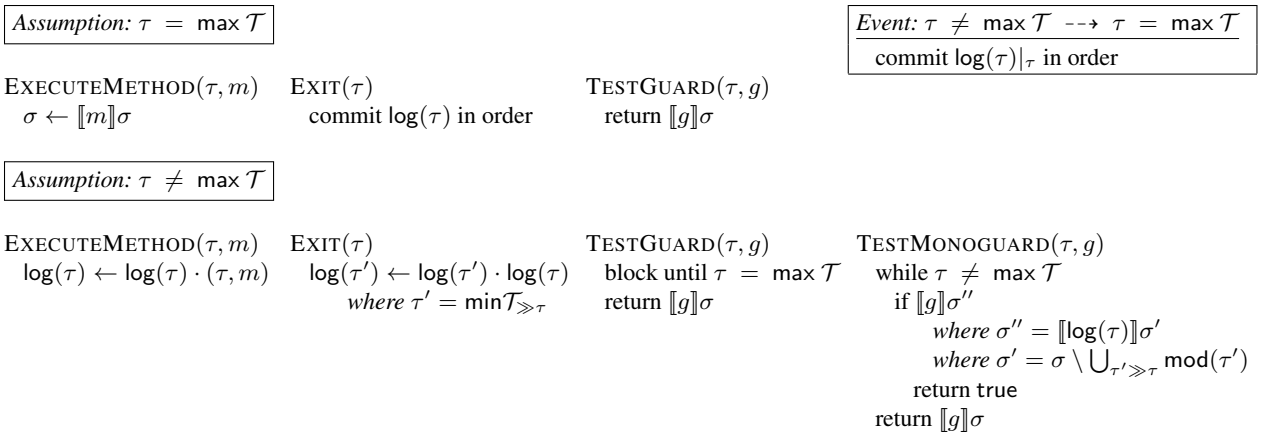

Figure 3. The TANGO protocol, described with the following supporting notation: σ is the shared state; $\mathcal{T}$ is the set of live tasks; and log maps every task τ to its log of outstanding methods (qualified by their original task)

Name	Domain	Description
DIJKSTRA SP	Graph Theory	Produces a shorest path tree
EDIT DISTANCE	Information Theory	Measures the difference between two sequences
ϵ-NET	Machine Learning	Computes an ϵ-net from a set of points
GREEDY COLORING	Graph Theory	Computes a vertex-coloring solution for a graph
JARVIS MARCH	Comp. Geometry	Computes the convex hull of a set of points
KNAPSACK	Combinatorial Opt.	Solves the $0/1$ knapsack optimization problem
KRUSKAL MSF	Graph Theory	Computes an MST incrementally
MIS	Graph Theory	Finds a max. independent set of graph vertices
PARTITION	Combinatorics	Decides if multiset has two equal partitions
REVERSE DELETE	Graph Theory	Computes an MST decrementally
RULE PRUNING	Data Mining	Finds a min. set of assoc. rules incrementally
SEQ. ALIGNMENT	Bioinformatics	Performs global alignment of two sequences

Table 1. Benchmark characteristics

Speedup	Overhead
0.95 – 0.9	1.1 – 1.11
2.62	1.25
0.76 – 4.4	1.21 – 1.26
1.5 – 3.5	1.35 – 1.16
0.88 – 0.75	1.1 – 1.14
2.55	1.19
0.825 – 5.8	1.2 – 1.19
3.9 – 4.2	1.09 – 1.12
2.36	1.23
0.87 – 6.25	1.16 – 1.17
0.77 – 5.2	1.17 – 1.21
2.4	1.26

Table 2. Parallelization stat.s

lutions to subproblems are cached in a lock-free hash table, and the order in which subproblems are computed is random.

Work Distribution (BSP) [17] This framework works for associative operators, such as min and max. Each thread applies the operator to a portion of the candidate values, and broadcasts its local candidate to all other threads.

Lazy Speculation (OPT) [55] This form of optimistic synchronization tests for conflicts only at transaction commit time. The transaction operates on a privatized version of the shared state.

For 8 of the benchmarks, we compared TANGO with the technique that was originally proposed to parallelize the benchmark. Our choice of parallelization technique for the remaining 4 benchmarks (GREEDY COLORING, RULE PRUNING, REVERSE DELETE and ϵ-NET) was guided by their similarity to the other 8 benchmarks. The available parallelism in GREEDY COLORING has the same structure as MIS, being due to the sparse dependencies between the iterations. RULE PRUNING and ϵ-NET are similar to KRUSKAL MSF in that a helper thread can determine, ahead of time, the redundancy of an element. Finally, for REVERSE DELETE, speculation is the only technique out of the above that yields a sound parallelization solution.

Performance Results The speedup and overhead numbers appear in Table 2. Speedup is calculated as the ratio of sequential running time to parallel running time using 16 threads. Overhead is the ratio of execution time using 1 thread to sequential execution time. For benchmarks with two inputs profiles, the left number corresponds to the low-parallelism input. The speedup trends going from 1 to 16 threads are visualized in Figures 4–15.

TANGO incurs a slowdown of up to 1.35x compared to the sequential implementation (on GREEDY COLORING 10Kx500K). TANGO achieves a speedup of over 3x on 6 of the benchmarks, being comparable to its competing specialized technique (if not faster) on 10 out of the 12 applications (DIJKSTRA SP and JARVIS MARCH being the exceptions). TANGO's speedup on DIJKSTRA SP and JARVIS MARCH is negative. On DIJKSTRA SP, even the specialized

technique of [49] achieves negligible speedup. On JARVIS MARCH, however, TANGO is significantly worse than the technique of [17], which obtains a speedup of up to 7x. In contrast, TANGO outperforms lazy speculation on REVERSE DELETE, being more than twice as fast.

Discussion The speedup statistics reported above are consistent with our expectations. TANGO favors nondeterminism in read operations, as well as granular and predictable write operations. The classic example is REVERSE DELETE (*cf.* Section 3). This explains the poor results on JARVIS MARCH and DIJKSTRA SP, where a task can only communicate conservative intentions until a late point in its execution, and read operations are deterministic.

The reason why the BSP approach works well for JARVIS MARCH is that it exploits intra-operator, rather than inter-operator, parallelism. Instead of parallelizing the entire task of extending the convex hull, BSP parallelizes only the inner loop searching for the next point in the hull. We could do the same, applying TANGO to the (inner) search task, but preferred to remain consistent with the other benchmarks.

GREEDY COLORING and the DP algorithms are more compatible with TANGO. While read operations are deterministic, the task has narrow intentions as it commences. The remaining 5 benchmarks feature both nondeterministic reads and granular intentions, and thus behave well under TANGO.

A final note—concerning the overhead data in Table 2—is that the main penalty of TANGO lies in inducing task-centric views, where a view is computed by factoring in the effects of concurrent tasks. In all our benchmarks, the effects of a task can be summarized succinctly. In REVERSE DELETE, for instance, task modifications are restricted to a single graph edge. At the other extreme, in the "pathological" case of DIJKSTRA SP, task modifications span the entire state, which can again be stated concisely. This explains the tolerable synchronization overhead of TANGO.

6.2 Case Study: Commercial Security Testing Tool

For the second study, we considered a testing module contained in a commercial product for dynamic detection of web security vulnerabilities [3]. We report on our experience in applying TANGO systematically to "heavy" (sequential) loops in the code. The methodology and hardware configuration for this experiment are the same as in the previous one.

Tool Description The module's architecture is outlined in Figure 17. The input to the testing engine is an HTML page. Outgoing links are extracted, and "injection points", such as parameter, cookie and session values, are identified. Next, for each candidate injection point, the engine checks for security vulnerabilities by sending further requests for the respective link, this time substituting the original input value with different test payloads (*e.g.* a script block, when checking for a cross-site scripting [2] vulnerability). The response received from the subject application is then validated to confirm whether the attack had succeeded, in which case the discovered vulnerability is reported to the user.

Parallelization Process To parallelize the subject testing tool, we first reviewed the tool's profiling data to identify "heavy" sequential loops. Manual inspection of the candidate loops disclosed three candidates where TANGO is immediately applicable. These loops perform greedy pruning, which renders the guard governing the observable effect of an iteration nondeterministic and monotonic.

The first loop (Reflection) iterates over input points, mapping each point to its respective reflection contexts, and eliminating redundant targets based on context overlap. The other two loops (Testing-{1,2}) fire test payloads, checking for each payload whether it is compatible with the tool's online model of the site's server-side defenses, and if so, sending it, as explained in [3]. The first loop builds a coarse model of the site's defenses, which the second loop refines if the first loop fails.

Having settled on the parallelization targets, we next applied two transformations: First, we decorated the loops with the `Monotonic` attribute, and second, we replaced ADT implementations accessed within the loops with their counterparts from the TANGO library (*e.g.* the TANGO `Set` implementation). These transformations proved straightforward, requiring little time and few code changes.

Performance Results To measure the impact of our parallelization transformations, we compared between the original version of the tool, where the candidate loops are executed sequentially, and a version where the loops are parallelized using TANGO. The data inputs were sampled randomly from the tool's test suite, and the experiments were conducted in the same way as before. The performance results are provided in Table 3, which shows the fraction of time TANGO required to execute each of the loops compared to its sequential counterpart (*e.g.* 32% for the Testing-1 loop), as well as the overall execution time with TANGO, which improved the tool's performance by more than 40%.

Discussion TANGO proved effective for the Testing-1 and Testing-2 loops, yielding respective speedups of 3.12x and 2.43x. The main reason is that for most data inputs, parallel guard evaluation succeeds: The partial model due to early payloads suffices for parallel rejection of a large number of future payloads, which translates into noticeable performance improvement. In the Reflection loop, however, there is less parallelism to exploit. The majority of discovered reflection contexts are independent, obviating most attempts for early, concurrent pruning of redundant attack points.

7. Related Work

In this section, we survey other parallelization approaches that have features in common with TANGO. We refer the reader to [14] and references therein for a discussion of spec-

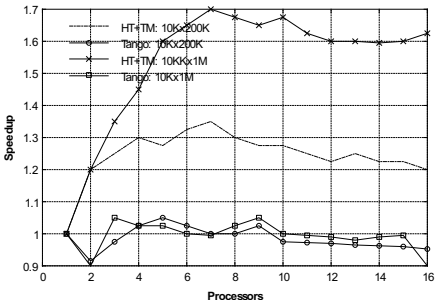

Figure 4. Speedup on DIJKSTRA SP

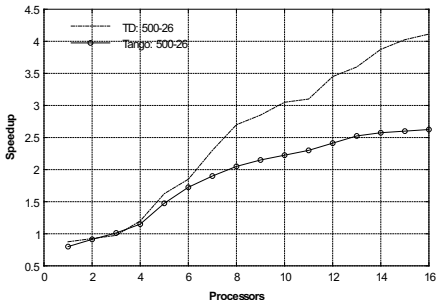

Figure 5. Speedup on EDIT DISTANCE

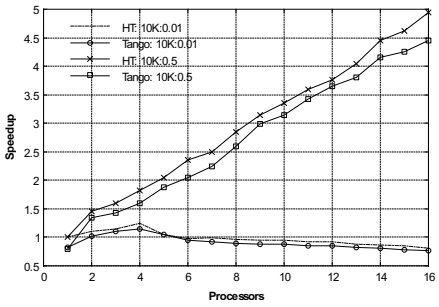

Figure 6. Speedup on ϵ-NET

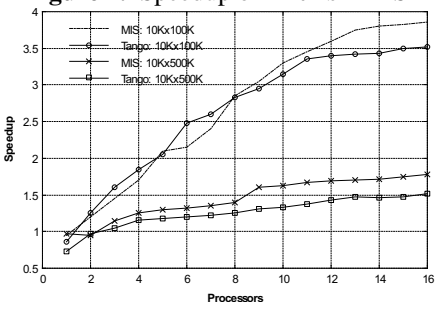

Figure 7. GREEDY COLORING

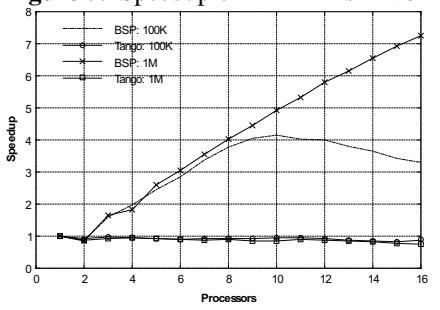

Figure 8. JARVIS MARCH

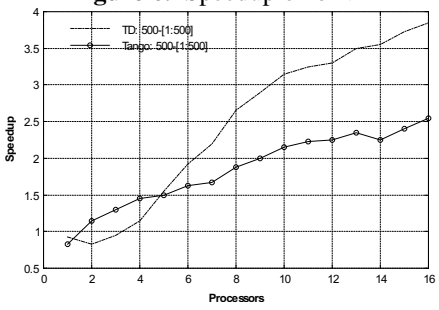

Figure 9. KNAPSACK

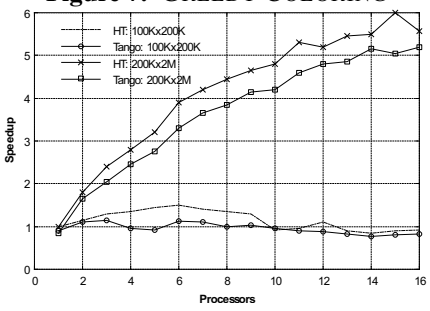

Figure 10. KRUSKAL MSF

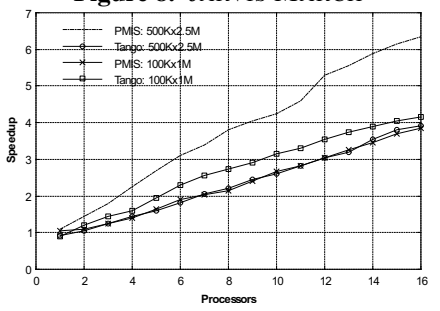

Figure 11. MIS

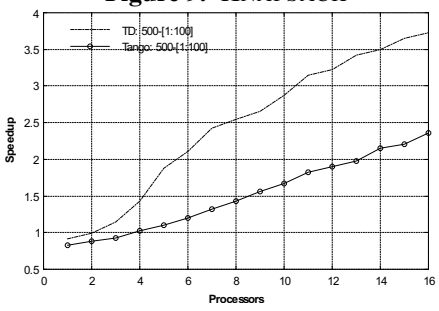

Figure 12. PARTITION

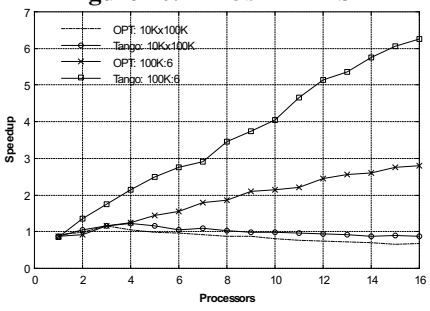

Figure 13. REVERSE DELETE

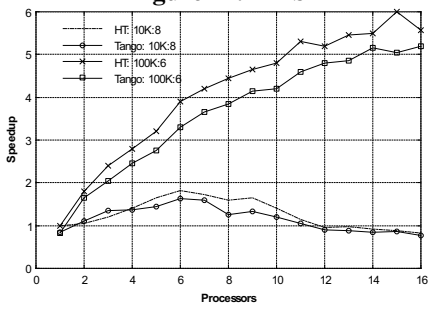

Figure 14. RULE PRUNING

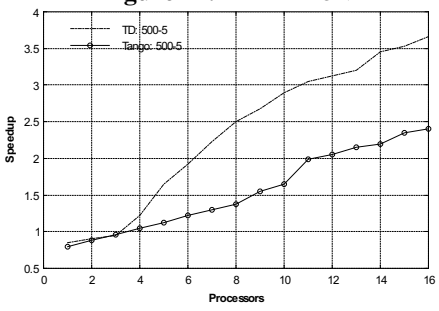

Figure 15. SEQ. ALIGNMENT

Figure 16. Speedup results

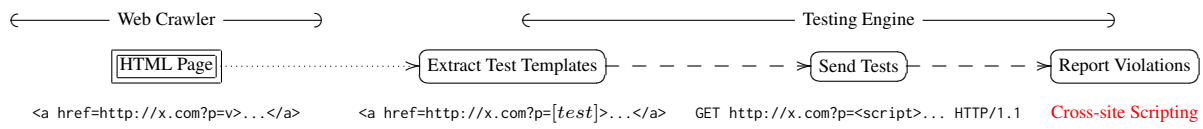

Figure 17. Architecture of the commercial engine, including illustration of its flow

	Reflection	Testing-1	Testing-2	Overall
Fraction	68%	32%	41%	58%
Speedup	1.47x	3.12x	2.43x	1.73x

Table 3. Overall as well as loopwise performance data, relativizing TANGO's running time to the sequential version

ification, verification and testing tools designed for nondeterministic codes.

Language Features NESL [10, 11] is a data-parallel programming language based on ML. Similarly to TANGO, NESL enables concise coding of a wide range of algorithms from different fields, including sequences and strings, graphs and computational geometry. Pregel [44] is a distributed programming framework, providing a simple API for programming graph algorithms while managing the details of distribution invisibly, including messaging and fault tolerance. Green-Marl [35] also specializes in graphs, focusing on graph analysis algorithms. The Green-Marl language captures the high-level semantics of the algorithm, allowing the Green-Marl compiler to apply nontrivial parallelization transformations.

TANGO, in contrast, is implemented as a library for loop parallelization, which can be integrated into real-world applications written in general-purpose languages (currently: Java and C#). As illustrated in Figure 1, TANGO's LINQ frontend allows use of declarative, SQL-like syntactic constructs for concise parallelization of challenging loops. This is close in spirit to Dryad [36] and DryadLINQ [37], which provide a general-purpose distributed execution engine for data-parallel applications. These systems, however, are constrained to side-effect-free loops, where TANGO is able to parallelize loops with irregular dependencies.

Harris and Fraser [28] use conditional critical regions (CCRs) for concurrency control. The user guards an atomic region by a boolean condition, with calling threads blocking until the guard is satisfied, which is reminiscent of the TANGO flow. The key difference is that [28] ensures the atomicity of a CCR via STM, whereas TANGO applies pessimistic synchronization, leveraging guard monotonicity. The `retry` construct in STM Haskell [29] allows a task to backtrack from its current behavior and select an alternate path that is hoped to have less conflict. However, `retry` does not involve coordination between threads, so one is still at the mercy of nondeterministic luck and what has been done in the past.

Communication and Visibility The TIC programming model [53] extends standard STM by allowing threads to observe the effects of other threads at selected points, thereby supporting operations that wait or perform irreversible actions inside transactional code. Lesani and Palsberg [42] describe a semantics that combines STM with message passing through tentative message passing, which keeps track of dependencies between transactions to enable undo of message passing in case a transaction aborts. Certain TM contention managers, such as the greedy contention manager [8, 25], let a transaction wait for a conflicting transaction or abort it depending on their status and priority.

Gueta *et al.* [27] describe a methodology for utilizing foresight information for effective synchronization in general-purpose systems. They apply their approach to safe composition of ADT operations (*i.e.* without deadlocks and atomicity violations). Their implementation consists of static analysis of the client code to compute which ADT operations it may use at different points in the execution, combined with lazy runtime synchronization over a "smart" implementation of the ADT, which leverages the semantics of the ADT's operations to increase parallelism. TANGO also bases synchronization on foresight, but with the different focus of parallelism due to latent nondeterminism, where we introduce task specialization as a synchronization primitive.

Dataflow programming [7, 20] makes data dependencies between tasks explicit by representing a program as a directed graph, where nodes represent units of work and arcs denote dependencies. Recently, there have been attempts to integrate dataflow abstractions into STM to enable controlled access to mutable shared state while constraining the number of false conflicts [23, 52]. TANGO takes a pessimistic approach, while enforcing fine-grained parallelization of applications with irregular dependencies, such as graph algorithms. Also, the purpose of communication in TANGO is to publish task intentions, rather than propagate data values.

Botincan *et al.* [13] present a technique for synchronization synthesis, which allows fine-grained parallelization of resource usage by using separation logic, rather than points-to analysis. The parallelization transformation inserts `grant` and `wait` barriers to transfer resources between concurrent tasks, which is reminiscent of the TANGO communication mechanism. Our focus, however, is on nondeterministic choice. TANGO focuses a task on a particular way of evaluating a nondeterministic action, which permits fine-grained synchronization, reducing contention, similarly to the idea of multiple granularity locking [24, 47].

Beyond Data Independence TANGO is able to enforce parallelism in the presence of data dependencies, as we illustrated in Section 3 for REVERSE DELETE. Another way of transcending data dependencies is *value speculation* [50], where a task is run speculatively based on a guess of the data values that would flow into its input parameters at its designated point of execution. The Alter framework [57] identifies and enforces optimistic parallelism that violates certain dependencies while preserving overall program functionality. This is governed by user annotations allowing reordering of loop iterations or stale reads.

The Bloom language [1] leverages theoretical results in monotonic logic to achieve eventual consistency in distributed programs. Bloom programs describe relationships

between distributed data collections that are updated monotonically (*i.e.* in an incremental fashion). Conway *et al.* [18] generalize Bloom to a lattice-parameterized system, Bloom^L. TANGO is similar to Bloom in leveraging monotonicity, but has wider applicability, supporting general-purpose languages and arbitrary state manipulations, beyond incremental updates (as in REVERSE DELETE).

8. Conclusion and Future Work

This paper takes a first step in exploiting parallelism due to nondeterminism. Existing approaches lack the necessary coordination mechanisms to guide concurrent tasks toward nonconflicting nondeterministic choices. We develop a novel concurrency paradigm that reduces conflict by letting nondeterministic operations *specialize* based on the potential *future* behavior of concurrent tasks.

We have implemented our approach in the TANGO synchronization protocol. TANGO generalizes several existing parallelization schemes. Our empirical evaluation of TANGO, including both algorithmic benchmarks and a commercial security testing system, demonstrates TANGO's ability to utilize available parallelism in challenging nondeterministic codes.

In the future, we plan to augment TANGO with the ability to automatically infer monotonicity annotations. This is facilitated by the fact that the shared state is encoded in relational form. We also intend to address other forms of nondeterminism, *e.g.* where a guard is not necessarily monotonic. We recently started work in this direction, the idea being to perform semantic compile-time checks, as well as compile-time synchronization synthesis, to compensate for lack of monotonicity.

References

[1] The bloom language. http://www.bloom-lang.net/.

[2] Cross-site scripting (xss). http://en.wikipedia.org/wiki/Cross-site_scripting.

[3] Ibm security appscan standard. http://www.ibm.com/software/awdtools/appscan/standard/.

[4] Linq: .net language-integrated query. http://en.wikipedia.org/wiki/Language_Integrated_Query.

[5] Watson libraries for analysis (wala). wala.sourceforge.net.

[6] S. G. Akl and G. T. Toussaint. A fast convex hull algorithm. *Inf. Process. Lett.*, 7:219–222, 1978.

[7] Arvind and D. E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow architectures, pages 225–253. Annual Reviews Inc., 1986.

[8] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 308–315, 2006.

[9] H. Attiya and A. Milani. Transactional scheduling for read-dominated workloads. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 3–17, 2009.

[10] G. E. Belloch. Nesl: A nested data-parallel language. regular tech report CMU-CS-92-103, Carnegie Mellon, 1995.

[11] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39:85–97, 1996.

[12] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, pages 308–317, 2012.

[13] M. Botincan, M. Dodds, and S. Jagannathan. Resource-sensitive synchronization inference by abduction. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 309–322, 2012.

[14] J. Burnim, T. Elmas, G. Necula, and K. Sen. Ndseq: runtime checking for nondeterministic sequential specifications of parallel correctness. PLDI '11, pages 401–414, 2011.

[15] J. Burnim, T. Elmas, G. C. Necula, and K. Sen. Ndetermin: inferring nondeterministic sequential specifications for parallelism correctness. In *PPOPP*, pages 329–330, 2012.

[16] S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 304–315, 2008.

[17] L. Cinque and C. diMaggio. A bsp realization of jarvis' algorithm. In *Proceedings of the 10th International Conference on Image Analysis and Processing*, pages 247–, 1999.

[18] N. Conway, W. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. Technical Report UCB/EECS-2012-167, EECS Department, University of California, Berkeley, http://db.cs.berkeley.edu/papers/UCB-lattice-tr.pdf, june 2012.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

[20] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, 1975.

[21] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, 1975.

[22] R. W. Floyd. Nondeterministic algorithms. *J. ACM*, 14:636–644, 1967.

[23] V. Gajinov, M. Milovanovic, O. Unsal, A. Cristal, E. Ayguade, and M. Valero. Integrating dataflow abstractions into transactional memory. *First Workshop on Systems for Future Multi-Core Architectures*, 2011.

[24] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Readings in database systems. chapter Granularity of locks and degrees of consistency in a shared data base, pages 94–121. Morgan Kaufmann Publishers Inc., 1988.

[25] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 258–264, 2005.

[26] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 315–324, 2007.

[27] G. G. Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Concurrent libraries with foresight. PLDI '13, 2013.

[28] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, 2003.

[29] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.

[30] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Concurrent data representation synthesis. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 417–428, 2012.

[31] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, pages 38–49, 2011.

[32] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. SPAA '10, pages 355–364, 2010.

[33] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*. ACM, 2008.

[34] M. Herlihy and E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[35] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 349–362, 2012.

[36] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[37] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 987–994, 2009.

[38] A. Katsigiannis, N. Anastopoulos, K. Nikas, and N. Koziris. An approach to parallelize kruskals algorithm using helper threads. Technical report, National Technical University of Athens, 2012.

[39] E. Koskinen, M. J. Parkinson, and M. Herlihy. Coarse-grained transactions. In *POPL*, pages 19–30, 2010.

[40] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *PLDI*, 2011.

[41] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[42] M. Lesani and J. Palsberg. Communicating memory transactions. PPOPP'11, pages 157–168, 2011.

[43] J. S. Foster M. Hicks and P. Prattikakis. Lock inference for atomic sections. *ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[44] G. Malewicz, M. H. Austern, A. J.C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[45] Y. Mansour. Pessimistic decision tree pruning based on tree size. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 195–201, 1997.

[46] B. McCloskey, F. Zhou, D. Gay, and E. A. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, pages 346–358, 2006.

[47] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17:94–162, 1992.

[48] Y. Ni, V. S. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. PPOPP'07, pages 68–78, 2007.

[49] K. Nikas, N. Anastopoulos, G. I. Goumas, and N. Koziris. Employing transactional memory and helper threads to speedup dijkstra's algorithm. In *ICPP*, pages 388–395, 2009.

[50] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *PLDI*, 2010.

[51] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[52] C. Seaton, D. Goodman, M. Lujan, and I. Watson. Applying dataflow and transactions to lee routing. In *Proceedings of Fifth Workshop on Programmability Issues for Heterogeneous Multicores*, 2012.

[53] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 191–210, 2007.

[54] A. Stivala, P. J. Stuckey, M. Garcia de la Banda, M. Hermenegildo, and A. Wirth. Lock-free parallel dynamic programming. *J. Parallel Distrib. Comput.*, 70:839–848, 2010.

[55] O. Tripp, R. Manevich, J. Field, and M. Sagiv. Janus: exploiting parallelism via hindsight. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 145–156, 2012.

[56] O. Tripp, G. Yorsh, J. Field, and M. Sagiv. Hawkeye: effective discovery of dataflow impediments to parallelization. In *OOPSLA*, pages 207–224, 2011.

[57] A. Udupa, K. Rajan, and W. Thies. Alter: exploiting breakable dependences for parallelization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 480–491, 2011.

[58] V. N. Vapnik. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., 1995.