CS739 P4: Solving@Home

Wiley Corning
Spring 2022

1. Overview

To solve a program synthesis problem, one must search a given language of programs to find a term that satisfies certain behavioral constraints. This task is NP-hard: the number of terms to be searched grows exponentially with the size of the grammar or the minimal size of a solution. However, some synthesis techniques are well-suited to parallelization. This is especially true of enumerative synthesis, in which each term in the language is checked in some deterministic order. Different sections of the search space can be checked independently and concurrently; furthermore, these subproblems are small enough to be evaluated on devices that are relatively weak or only briefly available.

In this project, we propose and implement a distributed system for solving synthesis problems in parallel. Inspired by Folding@Home [1], we envision that this software could run on a large network of volunteer computers. Such devices would be heterogeneous, untrusted, and potentially unreliable. To maintain reliability in the face of Byzantine failures, we implement a consensus mechanism adapted from the method of Castro and Liskov, 1999 [2] that is robust to a limited number of slow, unreliable, or maliciously coordinating volunteers.

This report describes the design and implementation of the system, *Solving@Home*, and presents experimental evidence to support our claims of robustness. We discuss the synthesis task as-implemented, and describe how our system uses the structural characteristics of enumerative synthesis to effectively divide work. Some aspects of our intended design could not be implemented due to time constraints; we will explain these design elements and our plans for future implementation.

2. Synthesis task

We implement a naïve form of enumerative synthesis for a fixed language of integer expressions. This technique constructs a dense map from sequence numbers to syntax trees, and then checks whether each sequence number satisfies the problem's constraints. More specifically, we consider a grammar containing m productions, each of which is assigned a number from 0 to $m-1$; the flattened syntax tree of any term in the grammar can then be specified as a number written in base-m.

For example, suppose we have the productions $E ::= x \mid y \mid (-E) \mid (E + E)$. The program $-x + y$, when flattened, consists of the productions $[+, -, x, y] \to [3, 2, 0, 1]$, so we can encode it as $3m^0 + 2m^1 + 0m^2 + 1m^3 = 3 + 8 + 64 = 75$. It will be the 75th program in the enumeration; we can recover the original syntax tree by repeatedly taking the quotient and remainder with respect to m.

Note that this encoding is inefficient: it will contain many syntactically invalid programs, e.g. xx or $+1-$, and many redundant programs, such as $(x + y)$ and $(y + x)$. We find this to be sufficient to demonstrate the characteristics of our distributed system. A proposal for a more sophisticated "production-ready" approach is described in Section 6.1.

Synthesis problems are sent to the leader as a specification consisting of a set of input-output examples that the program must satisfy; for example, the program `2x+y` might be specified by the examples `[(3,5) -> (11), (7,2) -> (16)]`. Our goal is then to find a minimal program with this behavior.

This needle-in-a-haystack problem bears some similarities to the tasks performed in proof-of-work blockchains. Once a solution is proposed, it can be reliably be checked in $O(1)$ time on any node, which might seem to undermine the need for consensus. However, we must also be able to reliably determine that a given range of terms does *not* contain a solution. A false negative could cause us to miss the solution, extending the synthesis task significantly (and perhaps infinitely).

3. Architecture

The *Solving@Home* system is centered around a single trusted leader node, which exposes RPC interfaces for volunteers and frontend clients. The leader's primary responsibilities are to manage the state of work on each problem, to delegate subproblems to volunteers, and to determine consensus for each subproblem from the volunteers' results. The leader does not perform any compute-heavy tasks by itself; we offload as much work as possible to the volunteers to improve the scalability of the system.

Delegation

Every problem is associated with a *task block generator*, which partitions the problem into indexed blocks of work. This generator is stateful, and may construct new task blocks based on the results of prior computation (see Section 6.1). We maintain the invariant that, after the generator yields a task block at index k, it will yield the exact same block at k from that point on; this ensures that different volunteers will be replicating the same work.

Volunteers communicate with the server over a bidirectional streaming RPC channel. The leader sends task blocks to the volunteer, which asynchronously responds with its evaluation results. To avoid unnecessarily waiting for round-trips, we will attempt to send multiple task blocks in immediate sequence even as the volunteer continues work on prior blocks. Each volunteer stub may have up to M task blocks *pending* at a time. Volunteer stubs track both their pending tasks and a list of all uncommitted tasks that they have attempted.

Each volunteer stub runs a work loop on its own thread. While a volunteer stub has less than M pending tasks, it will attempt to acquire new task. For each active problem, the leader will begin searching for new task blocks starting at the problem's commit head. It will skip any blocks that are already committed or have already been dispatched to the maximum permitted number of volunteers. Once a fresh block index is found, it will be requested from the generator; if the generator can't yet produce this block, we will repeat the process with other problems until a block is found (or else sleep).

The first time each volunteer is delegated a block from a particular problem, it is sent that problem's specification in a preliminary message. The volunteer's state incorporates the specifications of all active problems that it has worked on; this allows us to minimize the amount of information sent on the wire in later messages.

Consensus mechanism

As task block results are received from a remote volunteer, they are passed to the server's aggregation channel. The leader runs an aggregation loop in parallel with the volunteer stub loops. This loop repeatedly pulls results from the aggregation channel and adds them to a central state object.

The leader is initialized with a fixed Byzantine failure tolerance F, meaning that at most F volunteers may behave incorrectly without harming the overall computation. A minimum of $F + 1$ results must agree for us to have consensus on a given task block.

As in [2], we anticipate that faulty volunteers might stall indefinitely. Given that N volunteers are dispatched, we must be able to proceed with $N - F$ results; otherwise, F faulty volunteers could block progress. However, it may be the case that at least F honest volunteers are slower to respond than F faulty ones. Therefore, $N - F$ must be large enough to reach a correct consensus given F incorrect results, so we must have $N - F > 2F$. With this in mind, we set $N = 3F + 1$ to be the maximum number of volunteers assigned to each task.

The division of our problem into task blocks also serves as an effective form of load balancing. Every volunteer is allotted the same number of pending blocks, regardless of speed; a fast volunteer will quickly complete its tasks and receive new assignments, while a slow volunteer will still make occasional contributions to the work without accumulating a backlog that will bottleneck the entire system.

When consensus is reached for a task block, the server commits the consensus result to its overall record of the problem state. At this point it also drops all temporary data about the block, including its per-volunteer results and the record of which volunteers have attempted it; this reduces the rate at which the problem's memory use grows (although it remains $O(n)$). Any subsequent results it receives for this block are discarded.

If a block reaches consensus on a proposed solution to the problem, the server returns that solution to the client and clears all data associated with the problem.

4. Implementation details

Solving@Home is written in Rust. For our gRPC framework we use the Tonic library, which exposes an interface similar to that of the first-party gRPC library for C++. Tonic is built on the Tokio asynchronous runtime; we make use of Tokio's multithreading capabilities, as well as a few synchronization primitives such as MPSC channels.

5. Experimental observations

To demonstrate the reliability of our system, we instrumented the volunteer client with *slow* and *malicious* modes. When in slow mode, the client injects `sleep` calls between its responses; in malicious mode, the client instantly responds to each task block with a negative result, meaning that it will acquire as many tasks as possible and skip over any solutions.

In a configuration with $F = 1$ and four volunteers, we observed that the system makes progress at an equal rate given at most one malfunctioning volunteer and at most one slow volunteer. As expected, our aggregation loop finds the correct consensus value for each task block given $\leq F$ Byzantine failures. The system will continue to make progress given at least $F + 1$ reliable volunteers, but will potentially be bottlenecked by its slowest volunteers given $< 3F + 1$ total volunteers.

We observed that the system *does* accumulate meaningful information given fewer than $F + 1$ reliable volunteers. Each task block will remain uncommitted until enough new volunteers connect. When the $(F + 1)$th honest volunteer connects, each of its results will immediately result in a commit. This behavior is beneficial in that it allows some work to continue in the absence of a quorum. However, it is also potentially dangerous: the system will continue to generate new task blocks and store incremental results indefinitely, even after passing the true solution. This can cause unnecessary load on the volunteers, and will eventually crash the leader once it runs out of memory.

For small synthesis tasks such as $2x + y$ and $(x + y) * (x + y)$, Solving@Home produced results in only a few seconds. It struggled to find solutions to problems beyond an AST size of about 10 syntax nodes; given the inefficient nature of our enumeration scheme, the exponentially large search space quickly becomes prohibitive to traverse.

Deliberately crashing and restarting a volunteer in the middle of work did not disrupt the problem-solving process. However, we note a possible vulnerability: because the server treats each new volunteer channel as a fresh node, it is possible for a volunteer to crash, restart, and then report duplicate results that will be accepted by the server. This could result in a false consensus if the worker is malfunctioning consistently.

6. Planned enhancements

Our full design for Solving@Home included some elements that we have not yet been able to implement due to time constraints. This section will cover those missing pieces, both to complete the picture of our design and to lay out the immediate next steps in future work.

Generally speaking, we have prioritized implementing the parts of our design involving the volunteer pool, delegation scheme, and consensus mechanism, as these seemed to be the aspects most relevant to Distributed Systems thinking. We deferred work most heavily in the parts of our project relating to the program synthesis domain, and in standard DS challenges (replication, persistence, failover) that we have covered in earlier projects.

6.1 Term banking with reductions

In practice, the key to an effective enumerative synthesis technique lies in finding effective ways to reduce the size of the search space without omitting possible solutions. For example, if we know that the terms `(x+y)` and `(y+x)` are equivalent, then there is no need to examine both `(x+y)+1` and `(y+x)+1`. A common technique in enumerative synthesis is to maintain an *expression bank* of semantically distinct terms, and to construct new terms only from the contents of the bank [3]. Any new term that is enumerated must be tested to see whether it is part of an existing semantic equivalence class.

The method of *observational equivalence reduction* holds two terms to be equivalent if their behavior is identical with respect to a set of examples; for instance, $(x + y)$ and $(x + 1)$ are observationally equivalent on the inputs $[(2, 1), (0, 1)]$ [4]. Each element in the term bank is characterized by a unique signature of output values for some relevant set of inputs. In addition to its role as a pruning strategy, observational equivalence reduction also admits the use of dynamic programming to optimize the search: the precomputed output signature of each term can be used to memoize our interpreter.

Adding observational equivalence reduction to Solving@Home would require a few alterations to the current implementation. Instead of a range of term sequence numbers, our task blocks would consist of subsets of the expression bank; volunteers would be asked to check every term constructible from a given subset, and report any terms that are observationally distinct. For example, given a term bank $[x, y, 1]$, we might ask one volunteer to work on $[x, y]$, another to work on $[x, 1]$, and a third to work on $[y, 1]$; these might yield $(x + y)$, $(x + 1)$, and $(y + 1)$ as new distinct terms, respectively.

Future task blocks would then be constructed from the results of past task blocks. This could cause the system to temporarily run out of new tasks while waiting for prior results; we might also want to receive incremental progress reports from each volunteer when new distinct terms are discovered. The leader would need to maintain a complete view of the expression bank, and perform a final uniqueness check on any distinct term reported by a volunteer.

While our current code does not implement these features, it is structured with them in mind.

6.2 Connect to SemGuS solver codebase

In our prior research work, we have constructed solvers and other tooling for the *Semantics Guided Synthesis* (SemGuS) project [5]. SemGuS provides a framework for users to specify synthesis problems over languages with arbitrary semantics, and is intended to act as a lowest-common-denominator input format for a broad ecosystem of solvers.

Moving forward, we intend to remove the fixed-language synthesis code from the volunteers and instead pass problems directly to an existing SemGuS solver. Our frontend will then accept specifications written in the SemGuS format. Because this format is somewhat verbose, we will see an increased benefit to our choice to store specifications in the volunteer state.

6.3 Continue work given client disconnection

Currently, each problem-solving session takes place during the lifespan of a single long-running RPC call between the frontend client and the server. If the client disconnects during this period, any progress on its request will be lost; this is an unnecessary limitation, given that the client is only involved at the very start and end of the protocol.

We will improve on this by decoupling the protocol for requesting synthesis from the protocol for receiving results. Each client will generate a UUID for itself and send it with their synthesis requests; the server will hash the UUID with a problem sequence number, and return that key immediately while starting work asynchronously. The client will persist its UUID and keys to local disk. It could then receive progress updates from the server, over a streaming connection or via polling, until a final conclusion is reached. The server will persist results until they have been delivered to a client, or until a fixed lease period expires.

If the client is allowed to go offline, we must anticipate that a client may permanently disconnect after initiating a long-running synthesis task. Without further intervention, this could result in a situation where

6.4 Eject malfunctioning volunteers

If a volunteer returns results that differ from the eventual consensus, we must assume that it is experiencing some sort of Byzantine fault. If a volunteer consistently returns faulty results, we should permanently remove it from the pool.

6.5 Server replication

It is possible that the leader may crash or become inaccessible while work is ongoing. If this were to happen, we would like to seamlessly failover to a backup. A simple approach to this would be to use Raft to pick the leader from a set of replicas, and an external routing service to point clients and volunteers to the current leader; we could also use a consensus service such as a Chubby cell to provide mutual exclusion.

It's worth noting that every operation in the leader-volunteer protocol is idempotent. If a volunteer sends a duplicate result for some task block, the duplicate will simply be dropped by the leader. The leader can always "rewind" to an earlier point in the search without compromising the system's reliability. This suggests that we could back up the leader's data infrequently and asynchronously, and acknowledge every volunteer result immediately. If the leader were to crash while holding non-replicated progress data, the lost progress would be recovered as the relevant task blocks are reenumerated. Performing infrequent batched backups would reduce the traffic between servers, and allow us to more efficiently compress the content of each backup message.

Overview

Program synthesis is a challenging problem domain, but one that has great potential for massive parallelization. A volunteer computing network for synthesis would allow researchers to quickly solve complex problems without requiring heavy-duty computing equipment. Solving@Home demonstrates the potential for such a network to be fast, flexible, and robust to the disruptions associated with volunteer resources. Our implementation serves as a full proof-of-concept, and we have clearly identified the next steps to bring it to a production-ready state.

8. References

[1] Larson, Stefan M., et al. "Folding@ Home and Genome@ Home: Using distributed computing to tackle previously intractable problems in computational biology." *arXiv preprint arXiv:0901.0866* (2009).

[2] Castro, Miguel, and Barbara Liskov. "Practical byzantine fault tolerance." *OsDI*. Vol. 99. No. 1999. 1999.

[3] Gulwani, Sumit, Oleksandr Polozov, and Rishabh Singh. "Program synthesis." *Foundations and Trends® in Programming Languages* 4.1-2 (2017): 1-119.

[4] Albarghouthi, Aws, Sumit Gulwani, and Zachary Kincaid. "Recursive program synthesis." *International conference on computer aided verification*. Springer, Berlin, Heidelberg, 2013.

[5] Kim, Jinwoo, et al. "Semantics-guided synthesis." *Proceedings of the ACM on Programming Languages* 5.POPL (2021): 1-32.