VerifyThis 2026 - Challenge 2

Particle simulation

Based on a contribution from Arthur Charguéraud*

1 Context

This challenge is inspired by Pic-Vert [1], a parallel particle-in-cell (PIC) code for plasma simulations on multi-core processors. In PIC simulations, the physical space is divided into a grid of cells, and each cell contains a set of particles. At each time step, each particle moves to a new cell determined by its velocity and position. In the challenge, cells are represented as bags of integers, particles as integers, and a function f abstracts the particle motion.

1.1 Bags of integers

The algorithm we consider in this challenge manipulates bags (multisets) of integers. We specify their interface using the following functional-style type signatures:

```
type bag
create : unit -> bag
atomic_push : bag -> int -> unit
nonatomic_push : bag -> int -> unit
transfer : bag -> bag -> unit
iter : bag -> (int -> unit) -> unit
clear : bag -> unit
```

In the context of this challenge, we keep these operations abstract.[1]

The operation `create` creates an empty bag (which is allocated on the heap). The operations `atomic_push` and `nonatomic_push` add an element to the bag. Note that the operation `atomic_push`, which is slower than the `nonatomic_push` one, is the only operation that can be used concurrently by multiple threads. All other operations assume a single thread operating on the bag.

The operation `transfer` transfers all items from the first bag into the second bag, and leaves the first bag empty. The operation `iter` applies a function to each item in a bag; the bag being iterated must not be modified during the iteration. The operation `clear` empties a bag.

*Inria & ICube lab, CNRS, Université de Strasbourg, France

[1] In the real algorithm, bags are implemented as linked lists of fixed-size arrays.

```
void compute(bag cur[N], int K) {
  // Initially, each cur[i] contains a bag of integers.

  bag next_private[N]; // subject to nonatomic_push operations
  bag next_shared[N]; // subject to atomic_push operations
  foreach i from 0 to N-1 {
    next_private[i] = create();
    next_shared[i] = create();
  }

  // foreach timestep
  foreach k from 0 to K-1 {
    parallel foreach i from 0 to N-1 { // foreach current bag, in parallel
      iter(cur[i], fun x -> // foreach item in the bag
        int j = f(i, x); // compute destination
        if j = i then
          nonatomic_push(next_private[i], x);
        else
          atomic_push(next_shared[j], x);
      );
    }

    // prepare for next iteration:
    // merge 'next_private' and 'next_shared' into 'cur'
    parallel foreach i from 0 to N-1 { // foreach current bag, in parallel
      clear(cur[i]);
      transfer(next_private[i], cur[i]);
      transfer(next_shared[i], cur[i]);
    }
  }
}
```

Figure 1: Simplified real numerical particle simulation code.

1.2 The algorithm

Consider the function `compute` shown in Figure 1. This function takes as input N (where $N \geq 1$ is a constant) bags of integers and an integer $K \geq 0$, and moves items between bags according to a function `f : int -> int -> int`. More precisely, $f(i, x)$ expects i in $[0, N-1]$ and returns a result in $[0, N-1]$; the values of x are arbitrary (64-bit) integers. Each iteration moves an item x from bag `cur[i]` to bag `cur[f(i, x)]`. In the real code, f encodes the physics of particle motion (velocity and position updates) followed by a cell index computation; the condition $j = i$ abstracts a tile-based routing criterion that determines whether a particle can be handled without synchronization.

2 Tasks

The goal of this challenge is to verify that after K iterations, each item is stored in the bag determined by iterated application of f. Let $f_x = (\lambda i.\, f(i, x))$ be the mapping induced by item x. If an item x is initially stored in `cur[i]`, then after `compute` terminates it should be stored in `cur[j]`, where:

$$j = f_x^K(i) = (\underbrace{f_x \circ f_x \circ \cdots \circ f_x}_{K \text{ times}})(i)$$

For example, if $K = 4$, then $j = f(f(f(f(i, x), x), x), x)$.

Safety and termination. Verify that the function `compute`

1.1 is memory-safe (including that no arrays are accessed out of bounds).

1.2 terminates.

1.3 is data-race free.

As a reminder, `atomic_push` is the only operation safe to call concurrently on the same bag; all other operations (`nonatomic_push`, `transfer`, `clear`, `iter`) must not be called concurrently with any other operation on the same bag.

Correctness. To simplify the specification, we assume all the integers in all bags to be distinct.

2.1 Verify that any element initially stored in `cur[i]` is stored in `cur[j]` when the function terminates, where $j = f_x^K(i)$.

2.2 Verify that if an element x is stored in `cur[j]` when the function terminates, then there exists i such that x was initially stored in `cur[i]` and $j = f_x^K(i)$.

Possible simplifications. If the general problem is too complicated, feel free to make one or more of the following simplifying assumptions (from mildest to most restrictive):

(S1) $K = 1$

(S2) $N = 3$

(S3) $N = 2$

(S4) Sequential `foreach`-loop (instead of the parallel `foreach`-loop)

References

[1] Yann Barsamian, Arthur Charguéraud, Sever A. Hirstoaga, and Michel Mehrenberger. Efficient strict-binning particle-in-cell algorithm for multi-core simd processors. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 749–763, Cham, 2018. Springer International Publishing.