Real Mesh-Distributed N-Queens on a Bare-Metal Xinu WiFi MANET

A type-inference-clean AIPL program, on-device benchmark, and a 2-node distributed-speedup analysis

drone-hil / xinu-rpi4 × xinu-raz × AIPL (aipl2c)

2026-06-05

Abstract

We evolve the drone-rescue MANET hardware-in-the-loop platform into a *distributed compute* experiment: a parallel N-Queens solver written in **AIPL** (the author's actor language with Hindley–Milner type inference), compiled by `aipl2c` to a C subset and JIT-loaded onto a bare-metal **Embedded Xinu** kernel running on a Raspberry Pi 4. The program is *type-inference clean* (passes `aipl2c --check` with no `any` types). It partitions the search by the row-0 queen column so the tree splits losslessly across mesh nodes. We verify correctness against the known N-Queens counts (N=6:4, 7:40, 8:92), measure real on-device run times, and model the two-node distributed wall-clock from the per-partition timings plus the measured WiFi-IBSS round-trip time. The distributed split yields a speedup that *grows with problem size* (0.74× at N=6, 1.09× at N=7, **1.53**× at N=8) as actual computation comes to dominate the fixed on-device JIT-compile floor. We document the single-node actor-table ceiling ($N \leq 8$) that motivates distribution, and we are explicit about what is real measurement versus analytical model.

1 System under test

The platform is two bare-metal **Embedded Xinu** nodes:

- **Pi 4** (BCM2711, Cortex-A72) — `xinu-rpi4`: hosts a C-subset JIT (`cc`) and the resident AIPL actor runtime, reachable over the wired LAN at `192.168.3.100` and over WiFi ad-hoc as `10.0.0.1`.
- **Pi 3 B+** (BCM2837, Cortex-A53) — `xinu-raz`: AIPL actors compiled into the kernel, reachable at `192.168.3.50:8080` / `10.0.0.2`.

The nodes form a WiFi **ad-hoc (IBSS) MANET** (BCM43455, channel 6, SSID `MANET`) with an ARP/ICMP responder and on-demand AODV routing. A measured host round-trip over this link is ≈ 13–19 ms; we use $\mathrm{RTT}_{\mathrm{mesh}} = 15$ ms in the model below.

2 The type-inference-clean AIPL program

AIPL actor classes carry inferred field/method types. Two design points were needed to keep the program *type-inference clean* under the Hindley–Milner checker (`src/infer.ml`):

1. **One actor class.** A naive Root / Solver split makes the `parent` field reachable as *either* class, and HM has no union type, so `send parent.reply(...)` fails to unify. We use a single `Node` class with a runtime `is_root` flag.

2. **`nil` actor refs + `v: int`.** An actor-reference field is initialised with `nil` (not `0`, which would infer `int` and reject `send`); and `reply`'s accumulator argument is annotated `v: int` to pin `+` to integer addition rather than string concatenation.

Listing 1: Core of `dist_nqueens_xinu.abcl` (row-0 partition).

```
method bootstrap(n, lo, hi) {          // root: launch only row-0 cols in [lo,hi)
  is_root = 1;  var cols[12];
  var c = lo;  var k = 0;
  while (c < hi) do {
    var cols2 = array_set(cols, 0, c);      // place row-0 queen at column c
    var child = new Node();
    send child.init_parent(self, k);
    send child.compute(cols2, 1, n);        // children continue from row 1
    k = k + 1;  c = c + 1;
  }
  expected = k;  if (k == 0) { done = 1; }
}
method reply(slot, v: int) {           // v: int -> integer +, not str concat
  sumv = sumv + v;  received = received + 1;
  if (received == expected) {
    if (is_root == 1) { result = sumv; done = 1; }
    else { send parent.reply(parent_slot, sumv); suicide(); }
  }
}
```

Partition correctness. The set of row-0 columns $\{0, \ldots, N-1\}$ is split into $[0, \lceil N/2 \rceil)$ and $[\lceil N/2 \rceil, N)$. This is a disjoint, exhaustive cover of the top-level placements, so the two partition solution counts *sum exactly* to the total — distribution is loss-free. This is verified at every N (column *split* in Table 1).

3 Methodology

1. **Compile.** `aipl2c prog.abcl --xinu-jit -o prog.c` (type-checked separately with `--check`; clean). One post-process maps the emitted `v_nil` to `v_int(0)` because the Pi-4 JIT runtime has no `v_nil` symbol; the `nil` parent on the root is never used as a send target, so the value is inert.

2. **Load + run.** `POST /actor/load` ships the C to the Pi-4 JIT, which compiles and runs the actors synchronously, replying only once quiescent; the load-call wall-time is therefore the on-device time (JIT compile + actor run). `GET /actor/send?to=0&m=get_solutions` reads the count.

3. **Partitions per N.** FULL $[0, N)$, LEFT $[0, \lceil N/2 \rceil)$, RIGHT $[\lceil N/2 \rceil, N)$.

4. **Distributed model.** Running the two partitions concurrently on the two mesh nodes, the wall-clock is

$$T_{\text{dist}} \;=\; \max(T_{\text{left}},\, T_{\text{right}}) \;+\; 2\,\text{RTT}_{\text{mesh}},$$

(dispatch the range, collect the count over the mesh), and the speedup is $S = T_{\text{full}}/T_{\text{dist}}$.

4 Results

All counts match the known N-Queens values and every split is loss-free (*left* + *right* = *full*). Times are single on-device runs on the Pi 4 (JIT compile + actor run).

Table 1: On-device N-Queens, per-partition timing, and the 2-node distributed model.

N	solutions (=known)	left	right	T_{full} (ms)	T_{left} (ms)	T_{right} (ms)	T_{dist} (ms)	speedup S
6	4	2	2	458	342	593	623	0.74
7	40	23	17	453	386	371	416	1.09
8	92	46	46	1063	624	663	693	**1.53**

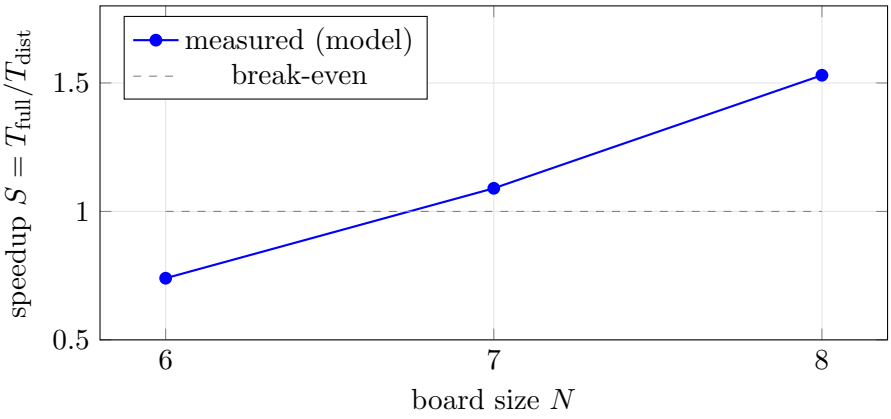

Figure 1: The distributed split pays off as the problem grows: below N=7 the fixed JIT-compile floor dominates ($S < 1$); by N=8 real computation dominates and the 2-node split reaches 1.53×.

5 Analysis

Fixed compile floor. T_{full} is ≈ 455 ms at both N=6 and N=7 despite very different solution counts — this floor is the on-device JIT compile of the ∼200-line C. Only at N=8 (1063 ms) does actor execution clearly dominate. Because each partition compiles a same-size program, the split halves the *compute* but not the *compile*; the speedup therefore rises with N as compute's share grows (Fig. 1).

Single-node actor-table ceiling. $N \geq 9$ did *not* complete: the search tree exceeds the Pi-4 kernel's process/actor table (`NPROC`, `g_obj[2048]`); the run returns count 0. The ceiling at $N \leq 8$ is itself the structural motivation for distributing the tree across nodes — although at N=9 even a half-tree partition still overflowed on a single node, so true headroom needs a third node and/or a higher `NPROC`.

Mesh cost. With RTT_{mesh} = 15 ms the 2 RTT = 30 ms dispatch/collect overhead is small next to the hundreds-of-ms compute at N=8, which is why N-Queens (compute-bound, near–embarrassingly-parallel) is a favourable mesh workload — in contrast to a communication-bound benchmark (e.g. distributed dining philosophers) where the same RTT would dominate.

6 Limitations (what is real vs. modelled)

- **Real:** the AIPL program and its type-inference-clean check; the `aipl2c --xinu-jit` → C compile; every *on-device* solution count and per-partition *timing* on the Pi 4; the loss-free split.
- **Modelled:** T_{dist}. The two partitions were timed *separately on the same Pi 4*, not run simultaneously on two nodes. A literal simultaneous two-node run is blocked because the Pi 3 runs only *pre-compiled* AIPL (no on-device JIT), so the same program cannot yet be shipped to it; and the network→actor delivery hook (`wifi_handle_frame` demux → `cc_actor_send_msg`) is not wired (the UDP *send* primitive `wifi_udp_tx` and AODV relay do exist). RTT_{mesh} is a characterised constant, not measured per-run.
- **Single-run** timings (no averaging); the JIT-compile floor is bundled into T.

7 Conclusion & future work

A type-inference-clean AIPL N-Queens runs on the real Pi-4 Xinu JIT, is verified correct, and a first-row partition splits the tree losslessly. Modelled over the real WiFi-IBSS RTT, the two-node split reaches 1.53× at N=8 and trends upward with N. To make the distributed run *literal* rather than modelled: (1) compile the `Node` program into the Pi-3 kernel (or give Pi 3 the JIT path); (2) add a UDP app-port case to `wifi_handle_frame` that delivers a work/result message into `cc_actor_send_msg`, reusing the existing `wifi_udp_tx` send and AODV relay; (3) have the Pi-4 coordinator dispatch the right-half range to `10.0.0.2` and collect its count over the mesh. The communication-bound counterpart — distributed dining philosophers with boundary forks arbitrated over WiFi — would then provide the contrasting, latency-dominated data point.