ベアメタル Xinu WiFi MANET 上の実メッシュ分散 N-Queens

型推論クリーン AIPL プログラム・実機ベンチマーク・2 ノード分散スピードアップ解析

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

2026 年 6 月 5 日

概要

ドローン救助 MANET の Hardware-in-the-Loop プラットフォームを**分散計算**実験へ発展させた。著者のアクター言語 **AIPL**（Hindley–Milner 型推論つき）で並列 N-Queens ソルバを記述し、`aipl2c` で C サブセットへコンパイルして、Raspberry Pi 4 上のベアメタル **Embedded Xinu** カーネルに JIT ロードして実行した。本プログラムは**型推論クリーン**である（`aipl2c --check` を `any` 型なしで通過）。row-0 の女王列で探索木を分割するため、木はメッシュ各ノードへ無損失に分けられる。既知の N-Queens 解数（N=6:4, 7:40, 8:92）と一致することを実機で検証し、実機実行時間を計測、さらに各 partition の実測時間と実測 WiFi-IBSS 往復時間から 2 ノード分散の wall-clock をモデル化した。分散分割のスピードアップは**問題規模とともに増大**し（N=6 で 0.74×、N=7 で 1.09×、N=8 で **1.53**×）、実計算が固定の JIT コンパイル下限を上回るにつれて効いてくる。分散の動機となる単一ノードのアクターテーブル上限（$N \le 8$）も記録し、**どこまでが実測でどこからがモデルか**を明示する。

1 対象システム

プラットフォームは 2 台のベアメタル **Embedded Xinu** ノードである：

- **Pi 4**（BCM2711, Cortex-A72）— `xinu-rpi4`：C サブセット JIT（`cc`）と常駐 AIPL アクターランタイムを持つ。有線 LAN `192.168.3.100`、WiFi ad-hoc では `10.0.0.1`。
- **Pi 3 B+**（BCM2837, Cortex-A53）— `xinu-raz`：AIPL アクターをカーネルにコンパイル組込み。`192.168.3.50:8080` / `10.0.0.2`。

両ノードは WiFi **ad-hoc (IBSS) MANET**（BCM43455, ch6, SSID `MANET`）を構成し、ARP/ICMP レスポンダとオンデマンド AODV ルーティングを備える。本リンクの実測往復時間は約 13–19 ms で、以下のモデルでは $\mathrm{RTT}_{\mathrm{mesh}} = 15\,\mathrm{ms}$ を用いる。

2 型推論クリーンな AIPL プログラム

AIPL のアクタークラスは推論されたフィールド/メソッド型を持つ。HM 型検査器（`src/infer.ml`）の下で**型推論クリーン**に保つため、設計上 2 点が必要だった：

1. **単一アクタークラス。**素朴な Root / Solver 分割では `parent` フィールドが**どちらのクラスにもなり得る**ため、HM にユニオン型が無く `send parent.reply(…)` の単一化に失敗する。実行時フラグ `is_root` を持つ単一 `Node` クラスを用いる。
2. **`nil` アクター参照 ＋ `v: int`。**アクター参照フィールドは `nil` で初期化する（`0` だと `int` に推論され `send` を拒否）。また `reply` の累積引数を `v: int` と注釈し、`+` を文字列連結ではなく整数加算に確定させる。

Listing 1　`dist_nqueens_xinu.abcl` の中核（row-0 分割）。

```
method bootstrap(n, lo, hi) {          // root: row-0 の列 [lo,hi) のみ起動
  is_root = 1;  var cols[12];
  var c = lo;  var k = 0;
  while (c < hi) do {
    var cols2 = array_set(cols, 0, c);      // row-0 の女王を列 c に置く
    var child = new Node();
    send child.init_parent(self, k);
    send child.compute(cols2, 1, n);        // 子は row 1 から継続
    k = k + 1;  c = c + 1;
  }
  expected = k;  if (k == 0) { done = 1; }
}
method reply(slot, v: int) {           // v: int で + を整数加算に確定
  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(); }
  }
}
```

分割の正当性。row-0 の列集合 $\{0,\dots,N-1\}$ を $[0,\lceil N/2\rceil)$ と $[\lceil N/2\rceil, N)$ に分ける。これは最上位配置の互いに素かつ網羅的な被覆なので、2 つの partition の解数は**厳密に総和が全解**になる — 分散は無損失である。これは各 N で検証している（表 1 の *left/right* 列）。

3　手法

1. **コンパイル。**`aipl2c prog.abcl --xinu-jit -o prog.c`（型検査は `--check` で別途実施しクリーン）。生成された `v_nil` を `v_int(0)` に 1 箇所置換する。Pi-4 JIT ランタイムに `v_nil` シンボルが無いためで、root の `nil` parent は send 先に使われないため値は無害。
2. **ロード＋実行。**`POST /actor/load` で C を Pi-4 JIT へ送る。JIT はアクターを同期的にコンパイル・実行し、静止状態になって初めて応答する。よってロード呼び出しの wall-time が実機時間（JIT コンパイル＋アクター実行）となる。`GET /actor/send?to=0&m=get_solutions` で解数を読む。
3. **N ごとの partition。**FULL $[0,N)$、LEFT $[0,\lceil N/2\rceil)$、RIGHT $[\lceil N/2\rceil, N)$。
4. **分散モデル。**2 つの partition を 2 ノードで並行実行したときの wall-clock は

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

（範囲を配布し、解数をメッシュ越しに回収）であり、スピードアップは $S = T_{\text{full}}/T_{\text{dist}}$。

4　結果

全ての解数が既知の N-Queens 値と一致し、全ての分割が無損失（$left + right = full$）。時間は Pi 4 上の単一実機ラン（JIT コンパイル＋アクター実行）。

表 1　実機 N-Queens、partition 別計測、および 2 ノード分散モデル。

N	解数 (=既知)	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**

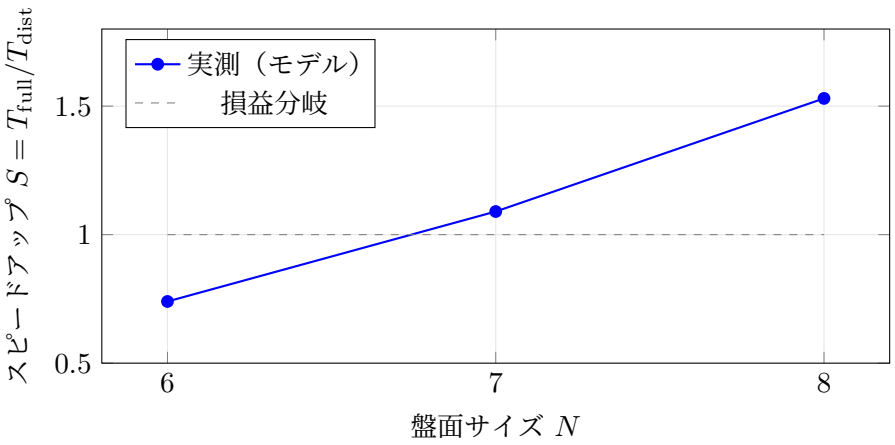

図 1　問題が大きくなるほど分散分割が効く：N=7 未満では固定の JIT コンパイル下限が支配し $S < 1$、N=8 で実計算が支配し 2 ノード分割は 1.53× に達する。

5　考察

固定コンパイル下限。T_{full} は解数が大きく異なるにもかかわらず N=6 と N=7 でともに約 455 ms である — この下限は ∼200 行の C の実機 JIT コンパイルである。N=8（1063 ms）でようやくアクター実行が明確に支配する。各 partition は同サイズのプログラムをコンパイルするため、分割は**計算**を半分にするが**コンパイル**は半分にしない。よってスピードアップは計算の比率が増す N とともに上昇する（図 1）。

単一ノードのアクターテーブル上限。$N \geq 9$ は**完了しなかった**：探索木が Pi-4 カーネルのプロセス/アクターテーブル（`NPROC`, `g_obj[2048]`）を超え、解数 0 を返す。$N \leq 8$ の上限こそが木をノード間に分散する構造的動機である — ただし N=9 では半分木の partition でも単一ノードで溢れたため、真の余裕には 3 台目のノードや `NPROC` の増加が要る。

メッシュのコスト。$\mathrm{RTT}_{\mathrm{mesh}} = 15\,\mathrm{ms}$ では $2\,\mathrm{RTT} = 30\,\mathrm{ms}$ の配布/回収オーバーヘッドは N=8 の数百 ms の計算に比べて小さく、これが N-Queens（計算律速・ほぼ embarrassingly parallel）がメッシュ向きである理由である — 同じ RTT が支配する通信律速ベンチ（例：分散・哲学者食事）とは対照的である。

6 限界（実測 vs モデル）

- **実測：**AIPL プログラムとその型推論クリーン検査、`aipl2c --xinu-jit` → C のコンパイル、Pi 4 上の各**実機**解数と partition 別**計測時間**、無損失分割。
- **モデル：**T_{dist}。2 つの partition は**同一の** *Pi 4* **で個別に**計測しており、2 ノードで同時実行したわけではない。文字通りの同時 2 ノード実行ができないのは、Pi 3 が *pre-compiled* AIPL のみ（実機 JIT 無し）で同プログラムを送れず、また受信ネットワーク → アクターの配送フック（`wifi_handle_frame` の demux → `cc_actor_send_msg`）が未配線のため（UDP 送信プリミティブ `wifi_udp_tx` と AODV 中継は実在）。$\mathrm{RTT}_{\mathrm{mesh}}$ は特性評価した定数でラン毎の実測ではない。
- **単一ラン**の計測（平均化なし）。JIT コンパイル下限は T に含まれる。

7 結論と今後

型推論クリーンな AIPL N-Queens が実機 Pi-4 Xinu JIT 上で動作し、正しさを検証でき、row-0 partition が木を無損失に分割する。実測 WiFi-IBSS RTT 上でモデル化すると、2 ノード分割は N=8 で 1.53× に達し N とともに上昇傾向を示す。分散実行を**モデルではなく文字通り**にするには：(1) `Node` プログラムを Pi-3 カーネルに組み込む（または Pi 3 に JIT 経路を付与）；(2) `wifi_handle_frame` に UDP app-port の分岐を足し、work/result メッセージを `cc_actor_send_msg` へ配送（既存の `wifi_udp_tx` 送信と AODV 中継を再利用）；(3) Pi-4 コーディネータが右半分の範囲を `10.0.0.2` へ配布し、その解数をメッシュ越しに回収する。その通信律速の対照版 — 境界フォークを WiFi 越しに調停する分散・哲学者食事 — が、レイテンシ支配の対照データ点を与えるだろう。