Both *Nature* and *SB* use Simulated Annealing (*SA*) [5] as a baseline for comparison. Table 2 of [2] gives a concise comparison of hyperparameters used by the two works. As in [3], we implement and run *SA* based on the description given in the *SB* manuscript. Our implementation differs from that described in *Nature* in its use of *move* and *shuffle* in addition to *swap*, *shift* and *mirror* actions. We also use two initial macro placement schemes, i.e., "spiral macro placement" whereby macros are sequentially placed around the boundary of the chip canvas in a counterclockwise spiral manner, and "greedy packer" whereby macros are packed in sequence from the lower-left corner to the top-right corner of the chip canvas [2]. Force-directed (FD) placement is used to update the locations of standard-cell clusters every $\{2n, 3n, 4n, 5n\}$ macro actions, where n is the number of hard macros; FD is not itself an action. The *SA* cost function is the proxy cost, which consists of wirelength, density and congestion.

As described in [3], Google's implementations of FD and proxy cost calculation are not open-sourced, but are only available via the *plc_client* in [7]. For speed and transparency, our *SA* experiments use our own C++ reimplementations of FD and proxy cost calculation; however, the Google *plc_client* is used for final FD soft macro placement and proxy cost evaluation at the end of the *SA* run. (Section 3.2 in [3] provides details of force-directed placement and proxy cost congestion.) Our *SA* codes are open-sourced in *MacroPlacement* [8].

A Go-With-the-Winners Enhancement. Similar to *Nature* and *SB*, our previous work in [3] runs 320 *SA* workers in parallel for 12.5 hours. Each worker, with its own hyperparameter setting, operates independently and does not communicate with other workers. Then, the macro placement solution with minimum proxy cost (as calculated by our C++ code) is used as the final *SA* solution. However, running 320 *SA* workers in parallel requires multiple servers, which may be impractical for users with limited computing resources.

To obtain a stronger *SA* baseline, we adopt the "go-with-the-winners" (GWTW) scheme [1] in a multi-threading implementation. In essence, GWTW allows a set of solution threads to proceed independently, but periodically executes a 'sync-up' whereby (i) the best threads are identified, (ii) their solutions are cloned to fill up the entire set of threads, and (iii) the threads then independently continue the solution process until the next 'sync-up'. This approach has seen previous adoption in physical design, e.g., for gate sizing [4].

The detailed algorithm is shown in Algorithm 1. The algorithm can be divided into following steps:

- **Lines 1-10:** We initialize all the *SA* workers in parallel. Each *SA* worker uses a unique random seed to shuffle same-size macros. Placement is initialized via spiral initialization for workers with odd IDs and greedy packing for those with even IDs.
- **Lines 14-17:** Each *SA* worker is run for *sync_iter* iterations in parallel. *sync_iter* is set based on $sync\_freq$. We use $sync\_freq = 0.1$, meaning there are 9 synchronizations among the workers.
- **Lines 18-21:** The algorithm stops when $Iter$ iterations are performed; otherwise, $syncWorkers$ selects the top k workers based on proxy cost and replicates their macro locations and orientations to the remaining workers evenly.

- **Line 22:** writes out the best macro placement solution in terms of proxy cost for each worker.

Algorithm 1: Simulated Annealing

Input: Random seeds: $seed = 1$,
Number of iterations: $Iters$,
$N \times \#macro$ moves per iteration ($N = 20$),
Initial temperature: $T_0 = 0.005$,
Minimum temperature: $T_{min} = 1 \times 10^{-8}$,
Cooling rate: $\alpha = \exp\left(\frac{\ln(T_{min}/T_0)}{Iters}\right)$,
Number of workers: $W = 80$,
Replicated top $k = 8$ workers,
Synchronization frequency: $sync\_freq = 0.1$

Output: Macro placement solutions.

```
// SA wrapped Go-With-the-Winners
workers ← create W workers;
for i ← 0 to W − 1 in parallel do
    workers[i].seed ← seed + i;
    workers[i].N ← N;
    workers[i].T ← T_0;
    workers[i].α ← α;
    if (i mod 2) = 0 then
        workers[i].macro_placement ← "spiral macro placement";
    else
        workers[i].macro_placement ← "greedy packer";
iter_count ← 0;
sync_iter ← Iters × sync_freq;
while true do
    end_iter ← min(Iters, iter_count + sync_iter);
    for i ← 0 to W − 1 in parallel do
        Each worker performs (end_iter − iter_count) SA iterations;
          applying N × #macro moves per iteration and updating
          temperature;
    iter_count ← end_iter;
    if iter_count = Iters then
        break;
    candidate_solutions ← extractTopK(workers, k);
    Evenly distribute these top-k solutions across all the workers;
Write out the best solution of each worker.
```

As noted above, after placing soft macros (standard-cell clusters) with GWTW *SA*, we use *CT*'s *plc_client* to evaluate the proxy cost of the best macro place-

ment solutions for each worker, and then return the best solution in terms of proxy cost for P&R evaluation. In our runs, the probabilities for five solution move operators (swap, shift, move, shuffle and flip) are respectively set to 0.24, 0.24, 0.24, 0.24 and 0.04. The number of iterations is set to ensure that overall runtime for each testcase is less than 12 hours on our slowest CPU server.[1]

Relative to the *SA* implementation in [3], our present *SA* implementation achieves similar or better results while using only one-fourth of the CPU resources: 80 threads instead of 320 threads, enabling execution on a single CPU server. Further, to ensure exact reproducibility across different platforms, (i) we use lookup tables for exponent computation and provide a binarized version of the lookup table in our repository; and (ii) we provide scripts to generate Docker and Singularity images that reproduce the same environment. Our testing across a range of Intel Xeon Gold and AMD EPYC CPUs confirms exact matching of *SA* solutions obtained by all 80 workers using the same Docker or Singularity image.

References

[1] D. Aldous and U. Vazirani, "Go With the Winners Algorithms", *Proc. FOCS*, 1994, pp. 492-501.

[2] S. Bae, A. Yazdanbakhsh, M.-C. Kim, S. Chatterjee, M. Woo and I. L. Markov, "Stronger Baselines for Evaluating Deep Reinforcement Learning in Chip Placement", August 2022. `https://statmodeling.stat.columbia.edu/wp-content/uploads/2022/05/MLcontra.pdf` (Author listing obtained from [6].)

[3] C.-K. Cheng, A. B. Kahng, S. Kundu, Y. Wang and Z. Wang, "Assessment of Reinforcement Learning for Macro Placement", *Proc. ISPD*, 2023, pp. 158-166.

[4] J. Hu, A. B. Kahng, S. Kang et al., "Sensitivity-guided Metaheuristics for Accurate Discrete Gate Sizing", *Proc. ICCAD*, 2012, pp. 233-239.

[5] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing", *Science* 220(4598) (1983), pp. 671-680.

[6] I. L. Markov, "Reevaluating Google's Reinforcement Learning for IC Macro Placement", *Comm. of the ACM* 67(11) (2024), pp.60-71.

[7] Circuit Training: An Open-source Framework for Generating Chip Floorplans with Distributed Deep Reinforcement Learning. `https://github.com/google-research/circuit_training`, *commit hash: 4c6fd98.*

[8] Implementation of Simulated Annealing. `https://bit.ly/4iz5o95`

[1] For Ariane, BlackParrot, MemPoolGroup, Ariane-X2 and Ariane-X4, we set the number of iterations to 18K, 9K, 4.5K, 9K and 4K, respectively. This corresponds, e.g., to ∼11 hours on an Intel Xeon Gold 6148 CPU, or ∼3 hours on an AMD EPYC 9684X CPU.