Harvard John A. Paulson
School of Engineering
and Applied Sciences

Parallelizing Hyperparameter Tuning in Artificial Neural Networks

Roy Han, Evan Jiang

Outline

1. Introduction
2. Architecture and Parallel Design
3. Findings
4. Analysis
5. Future Work

Introduction

- **Problem:** hyperparameters key to model performance – however, optimizing has become increasingly challenging b/c more **complex datasets** + **high-dimensional search spaces**
- **Current Solutions:** grid search, random search, adaptive selection (manual tuning), and SH
    - Sequential Halving (SH): sequentially iterate over all models on the base rung and promote the top half to the next rung, then double the epochs and iterate over all models in next rung and promote, etc.
- **Parallel Solution:** Asynchronous Sequential Halving (ASH) is an algorithm for efficiently tuning ML hyperparameters by combing over the hyperparameter search space with **multiple cores simultaneously**

Parallel Code Architecture

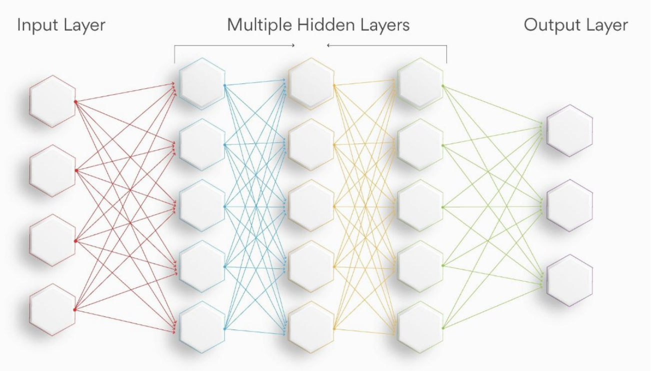
m5_ASH.cpp

Implementation of our parallelized rendition of **m5_SH.cpp** using openMP

Defines classes for solution space discretization and rung climbing

m5_neural_network.cpp

a cpp neural network file we adapted to suit our needs for the ANN model infrastructure



Auxiliary Programs

Job Scripts

sh_check.sh

ash_check.sh

ANN

test_features.txt

test_outputs.txt

train_classes.txt

train_features.txt

train_output.txt

predictions.txt

Harvard John A. Paulson School of Engineering and Applied Sciences

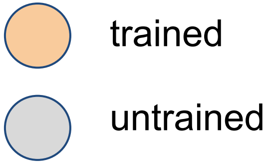
Parallel Design

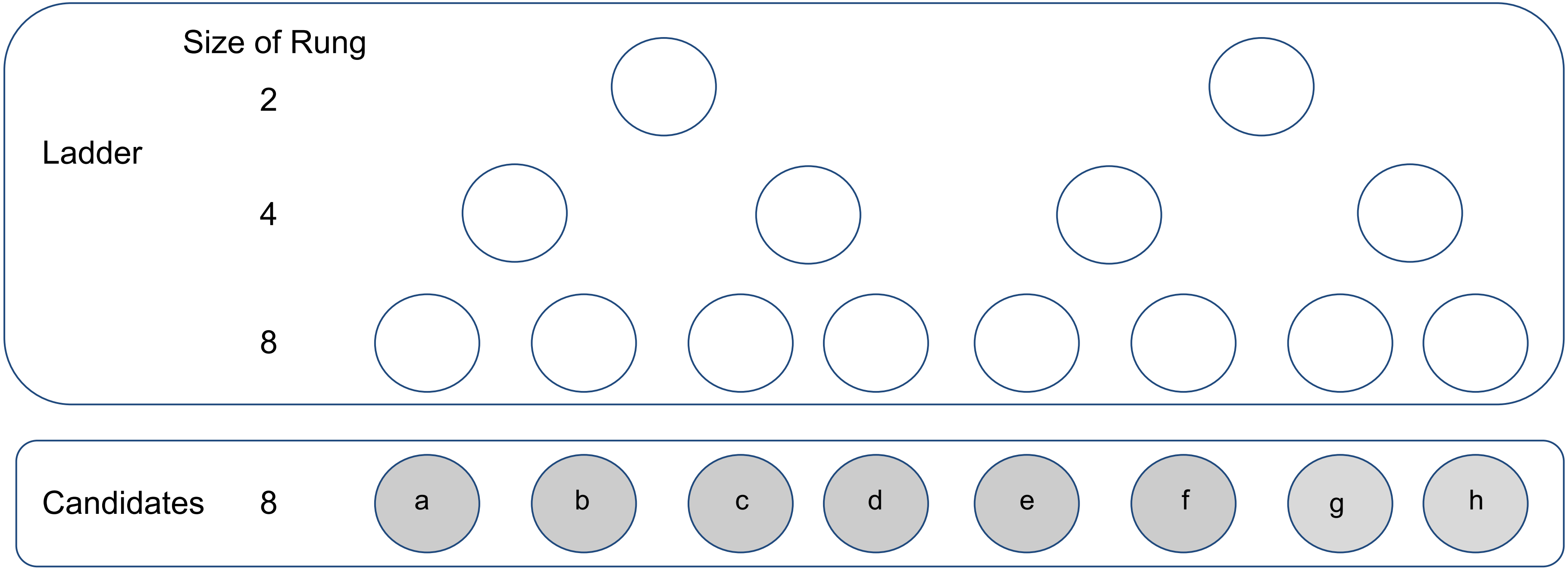
1. **Initialization**: We discretize the hyperparameter solution space into the vector **candidates**.
2. **Evaluation**: We distribute untrained candidate models to our **openMP** threads, evaluating each model with 1 epoch, keeping track of their scores.
3. **Advancement**: Holistically, the top half of each rung is promoted to the next rung. **Ladder** is a vector of **maxHeaps**: when a model is evaluated, its parameters and score pushed into the **maxHeap** corresponding to its rung. We double epochs for each rung.
4. **Asynchronous Execution**: Granularly, each thread checks **ladder** starting from the highest rung for an available top-half candidate. If there are no available candidates in **ladder**, it pops and trains a model from **candidates** to push into the bottom rung of **ladder** or waits until a candidate is available to train. Operations with ladder and candidates are **critically protected**, as they represent shared memory.

Parallel Design Justification

1. Data Structures
    a. After discretizing the solution space into **candidates** (a vector of Candidate models), we define our **ladder**, a vector of **maxHeap** and serves as the basis of candidate rung climbing
    b. every **maxHeap** corresponds to a rung on the ladder, managing the candidates on that rung in way that we can easily keep track of the highest scoring candidates
2. High Rung Prioritization
    a. we always **prioritize giving workers jobs that are highest up on the ladder**, as those *strictly dominate* the jobs on lower rungs in terms of runtime and computation intensity
    b. The **maxHeap** on every rung is how we maintain dynamic promotions and maximize our efficiency

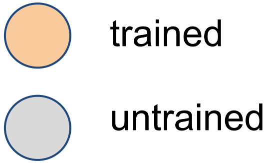
Example: 2 Threads

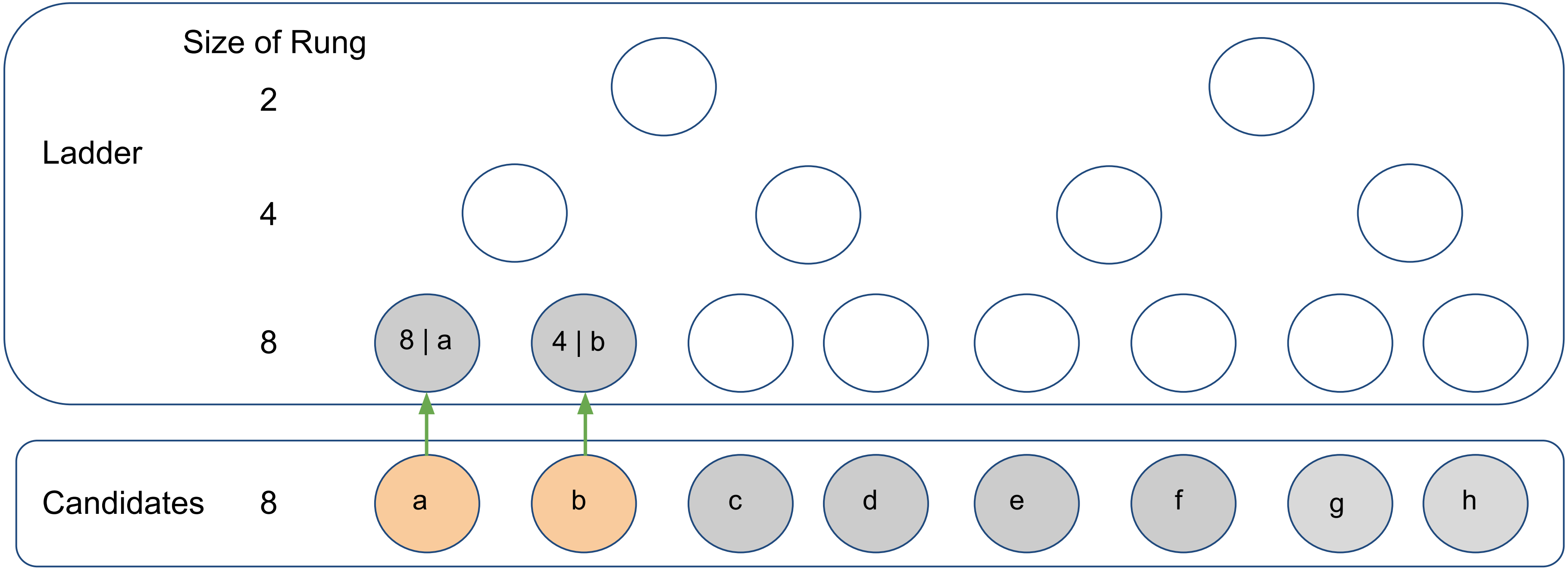




Ladder: vector of maxHeaps || Candidates: vector

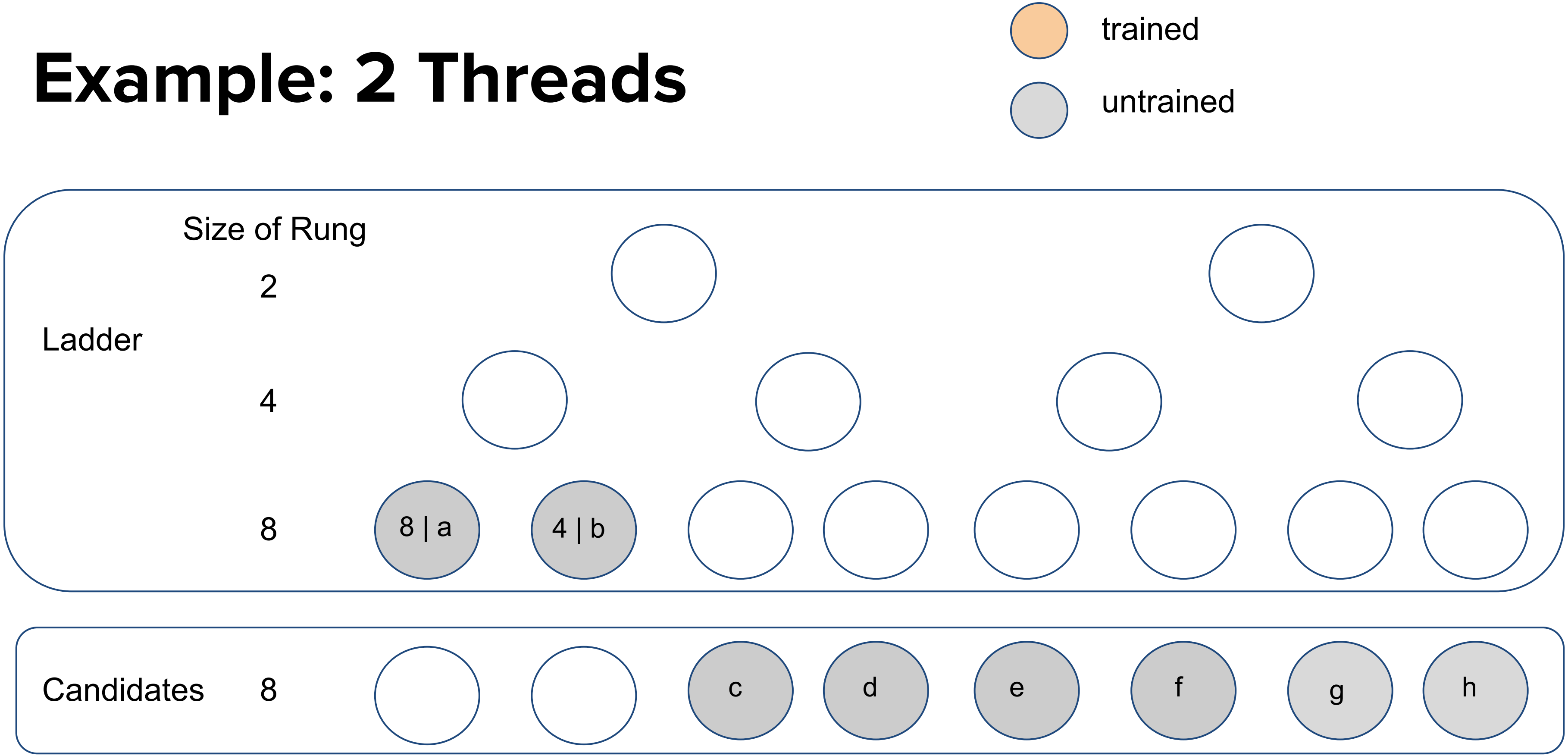
Example: 2 Threads





Ladder: vector of maxHeaps || Candidates: vector

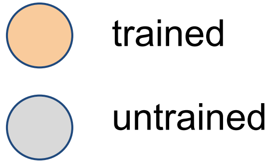
Example: 2 Threads

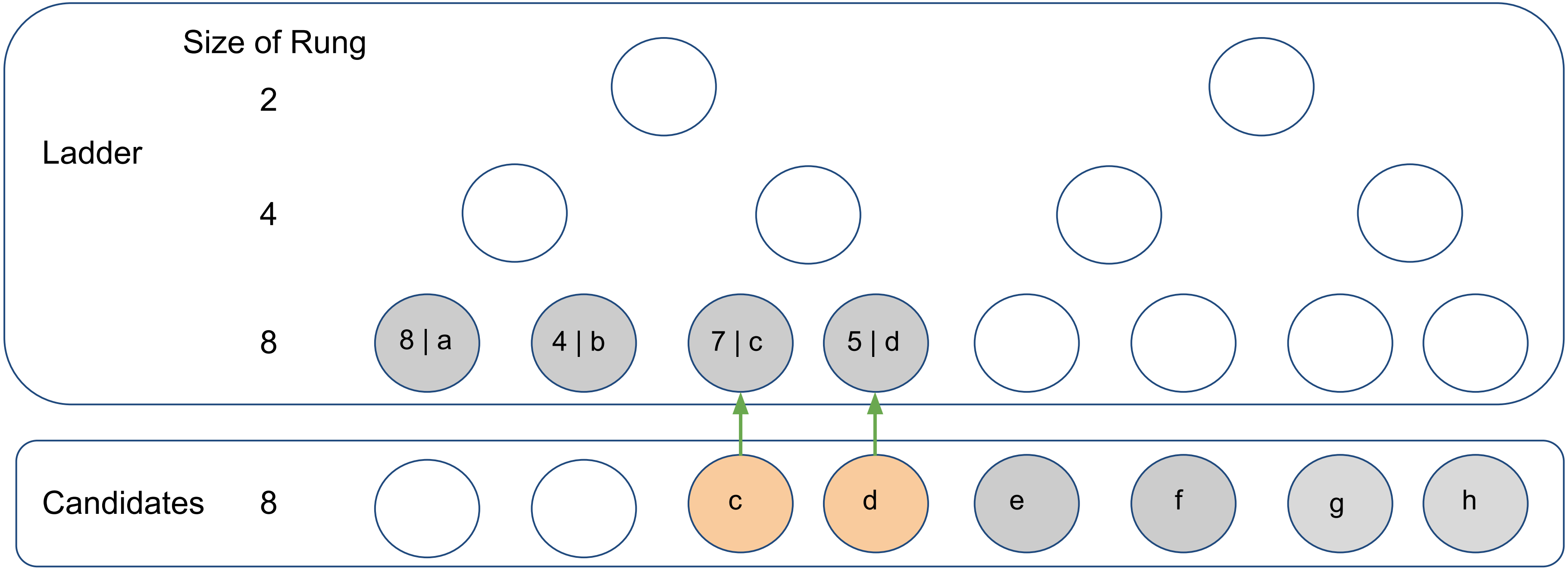


Ladder: vector of maxHeaps || Candidates: vector

Harvard John A. Paulson School of Engineering and Applied Sciences

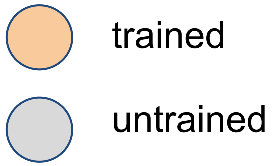
Example: 2 Threads

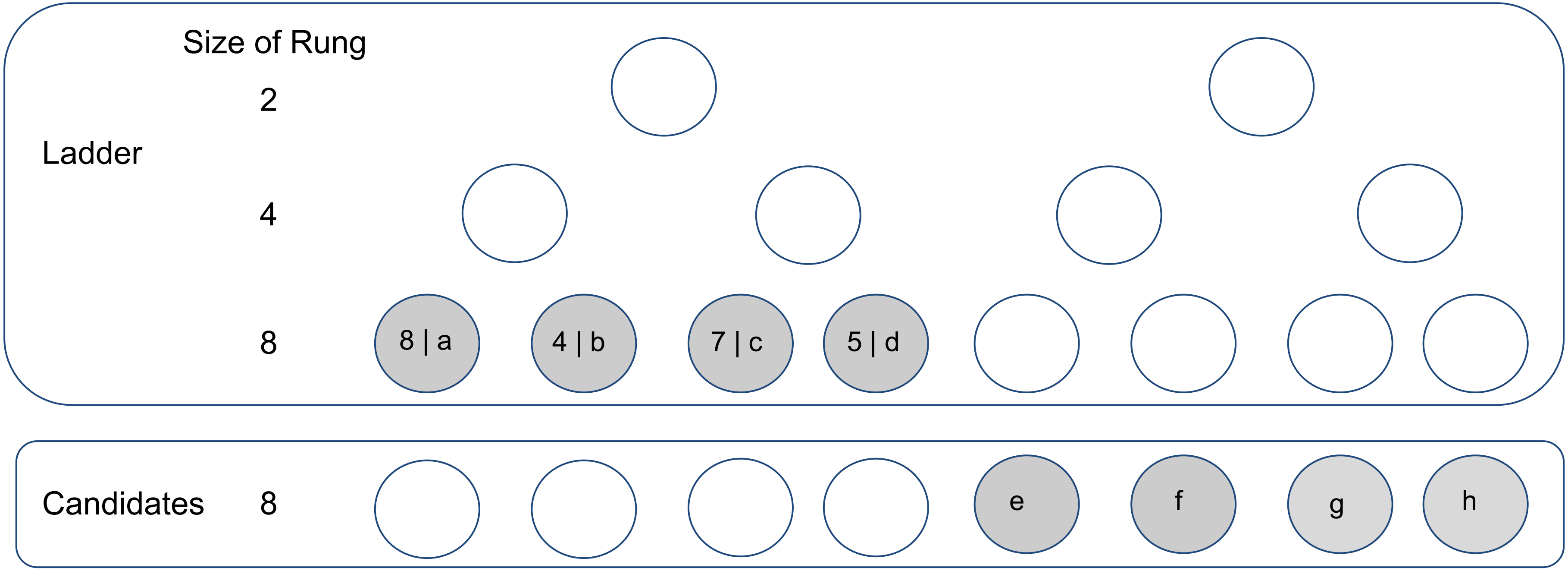




Ladder: vector of maxHeaps || Candidates: vector

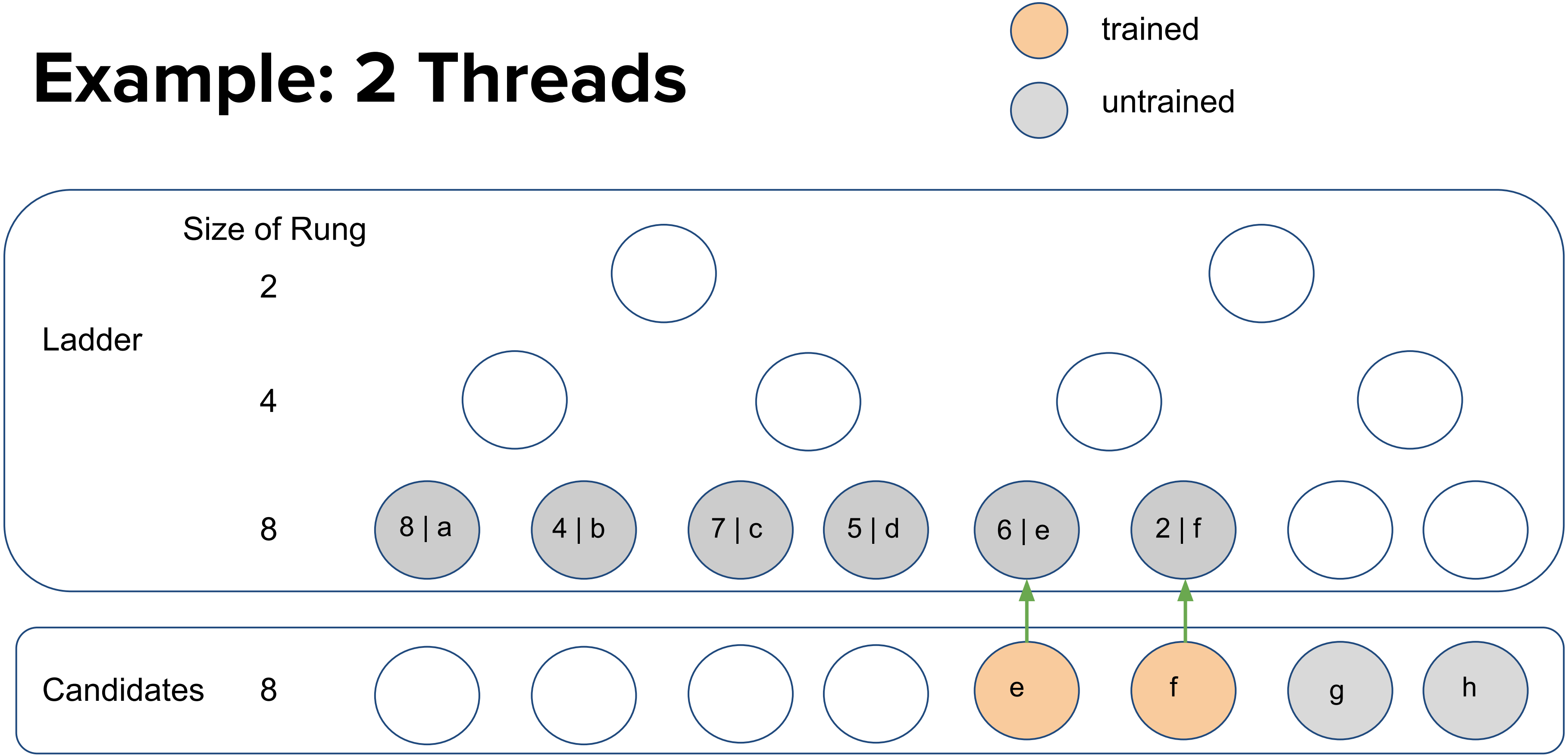
Example: 2 Threads





Ladder: vector of maxHeaps || Candidates: vector

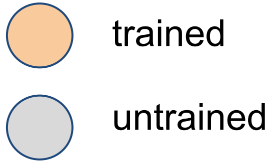
Example: 2 Threads

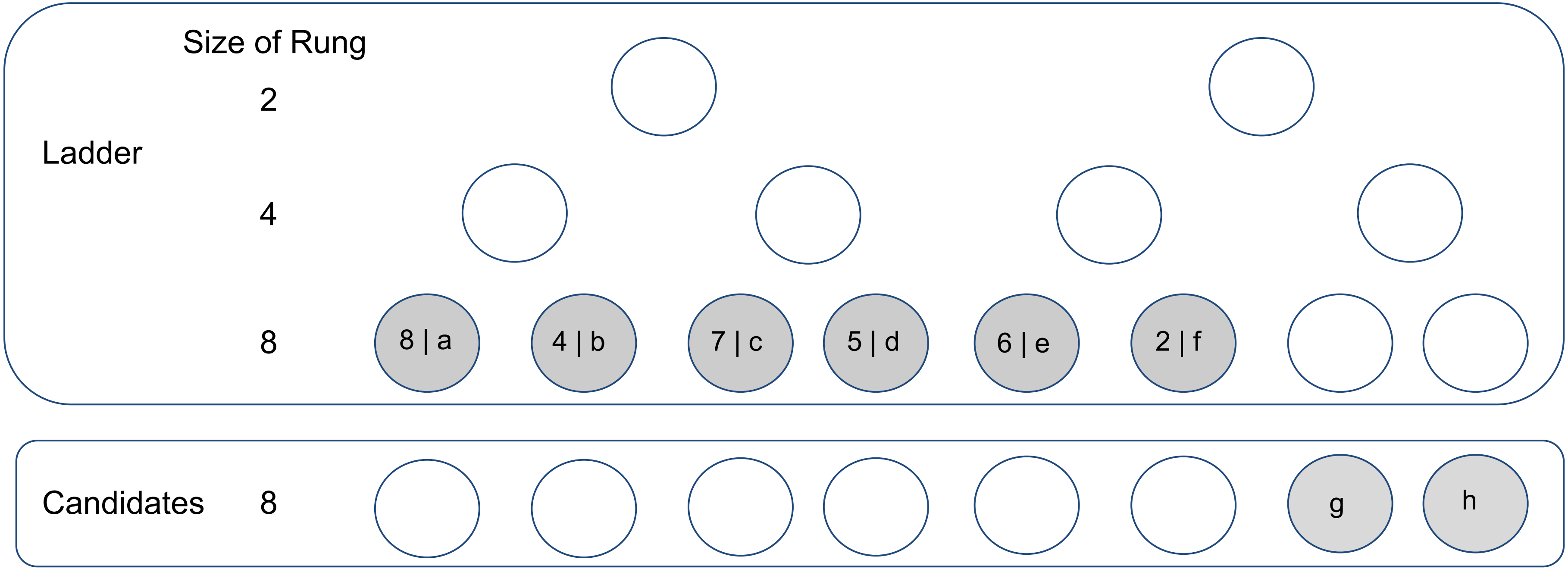


Ladder: vector of maxHeaps || Candidates: vector

Harvard John A. Paulson School of Engineering and Applied Sciences

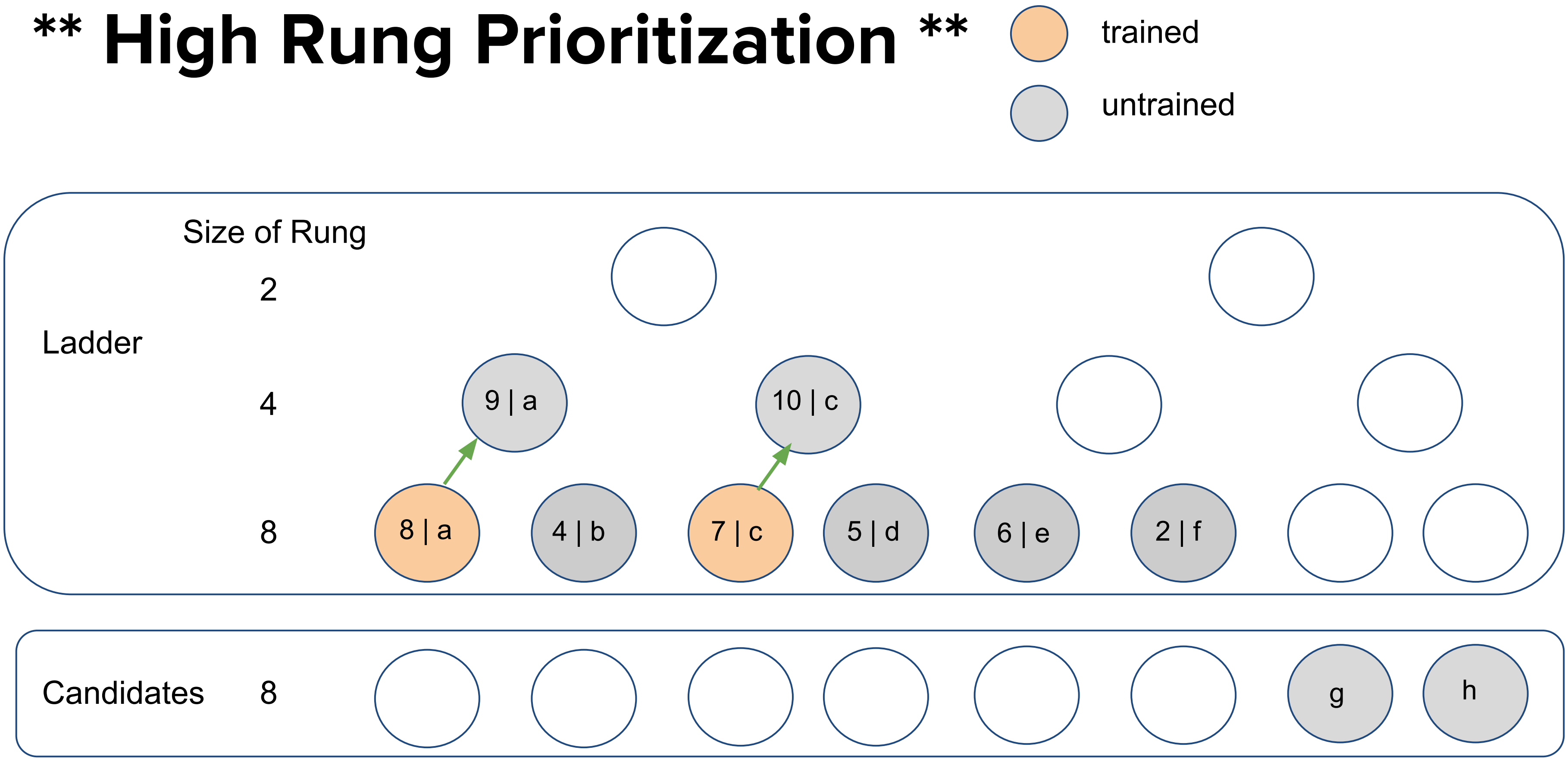
Example: 2 Threads





Ladder: vector of maxHeaps || Candidates: vector

** High Rung Prioritization **

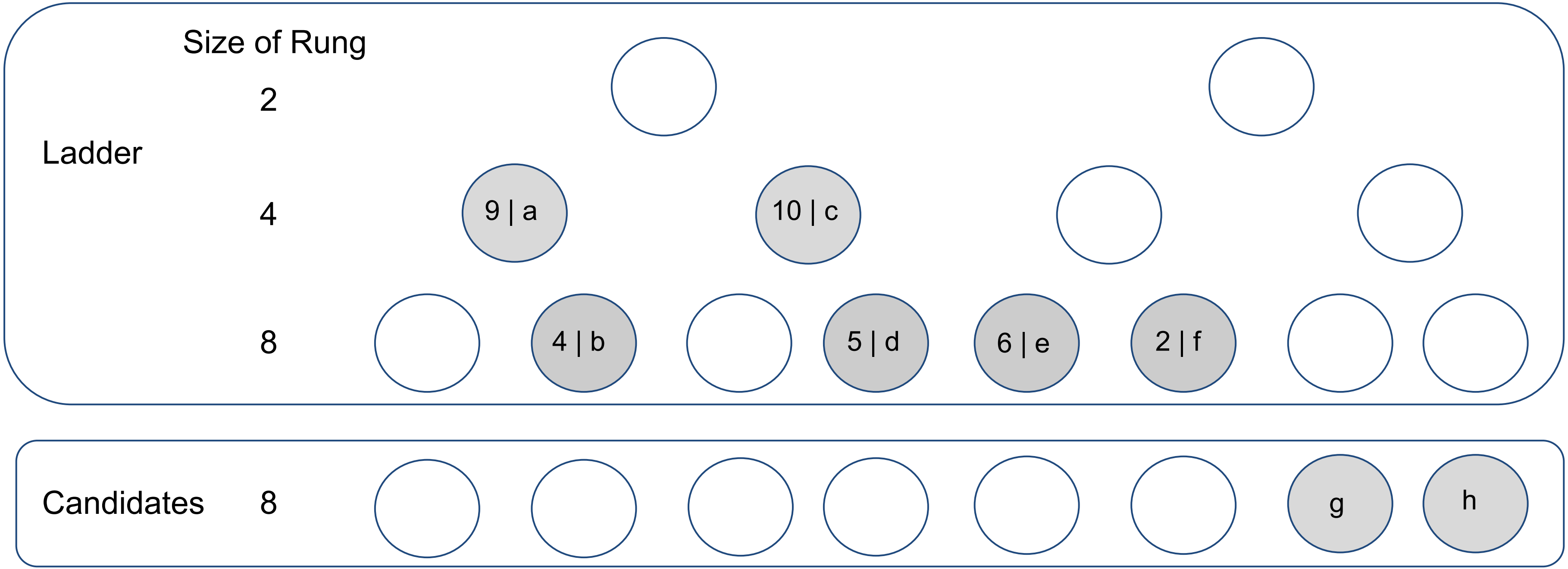


Ladder: vector of maxHeaps || Candidates: vector

Example: 2 Threads

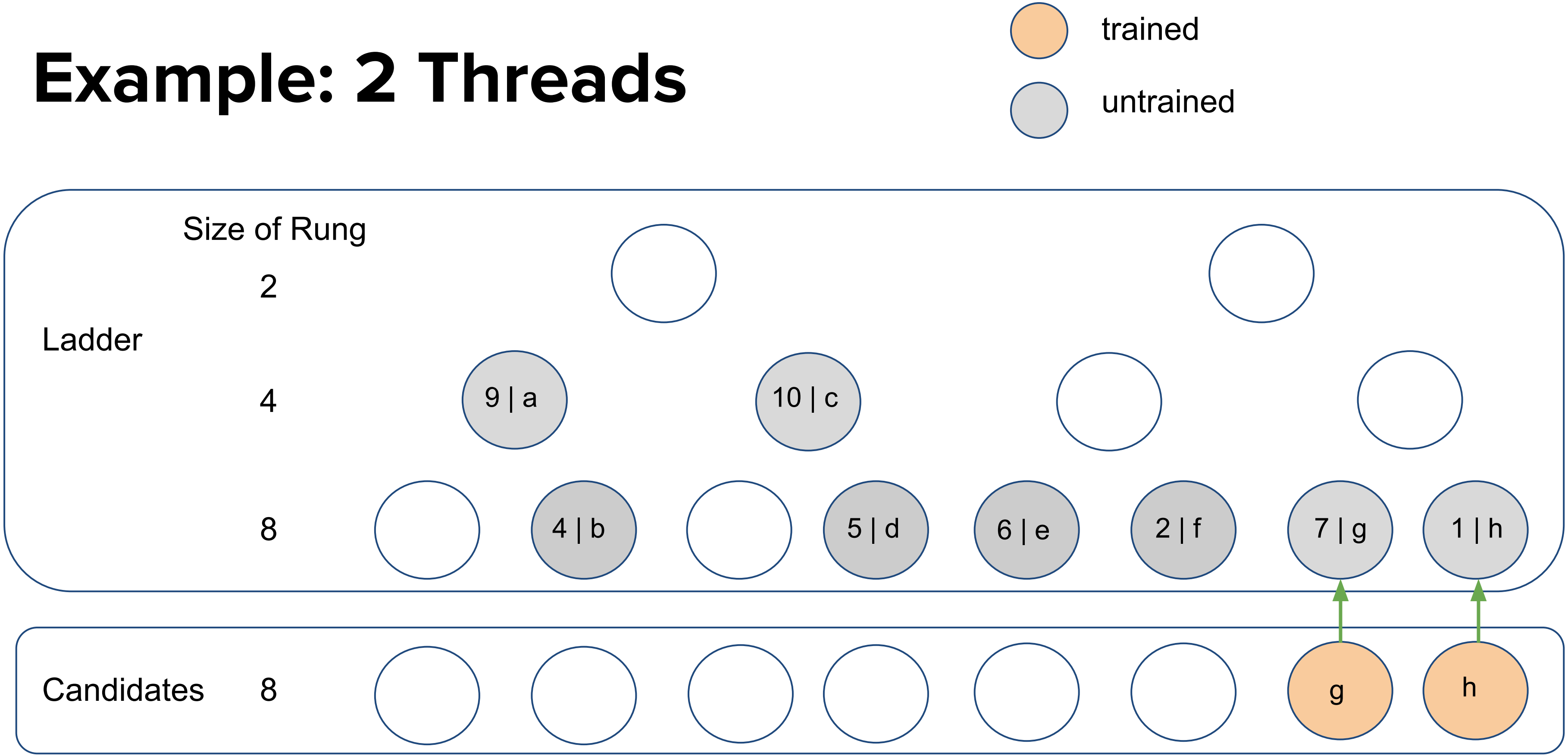
trained

untrained



Ladder: vector of maxHeaps || Candidates: vector

Example: 2 Threads

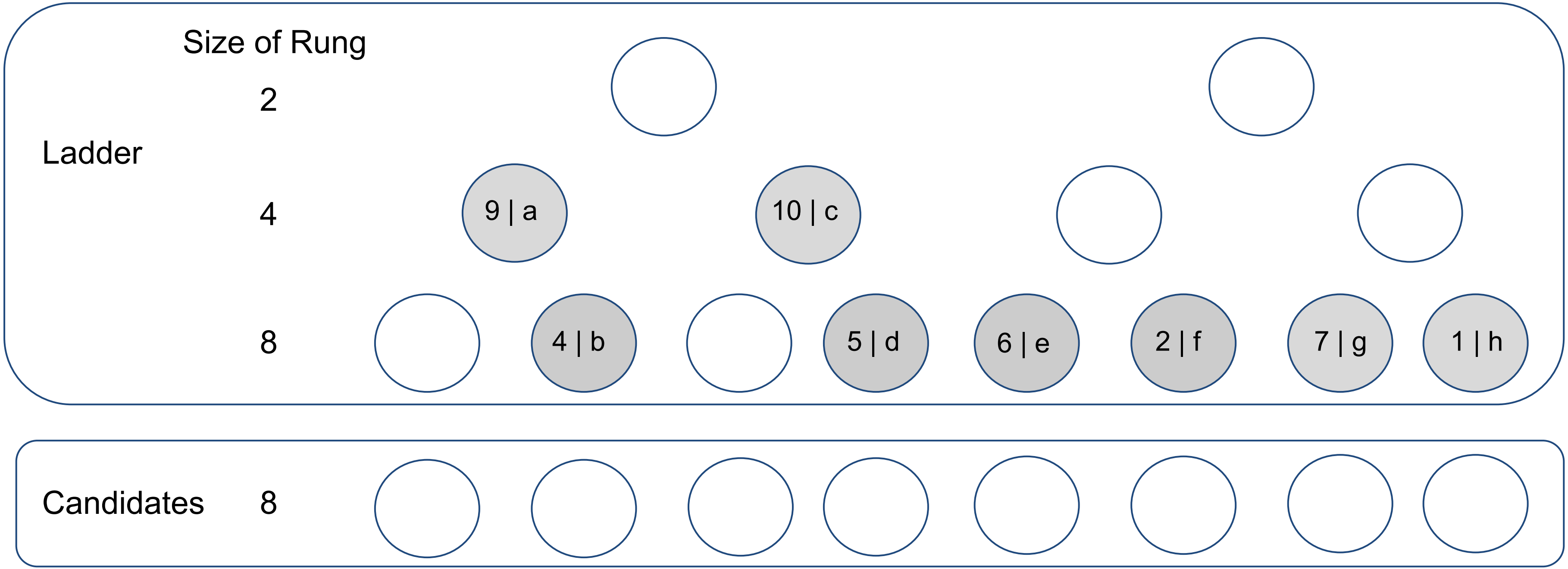


Ladder: vector of maxHeaps || Candidates: vector

Example: 2 Threads

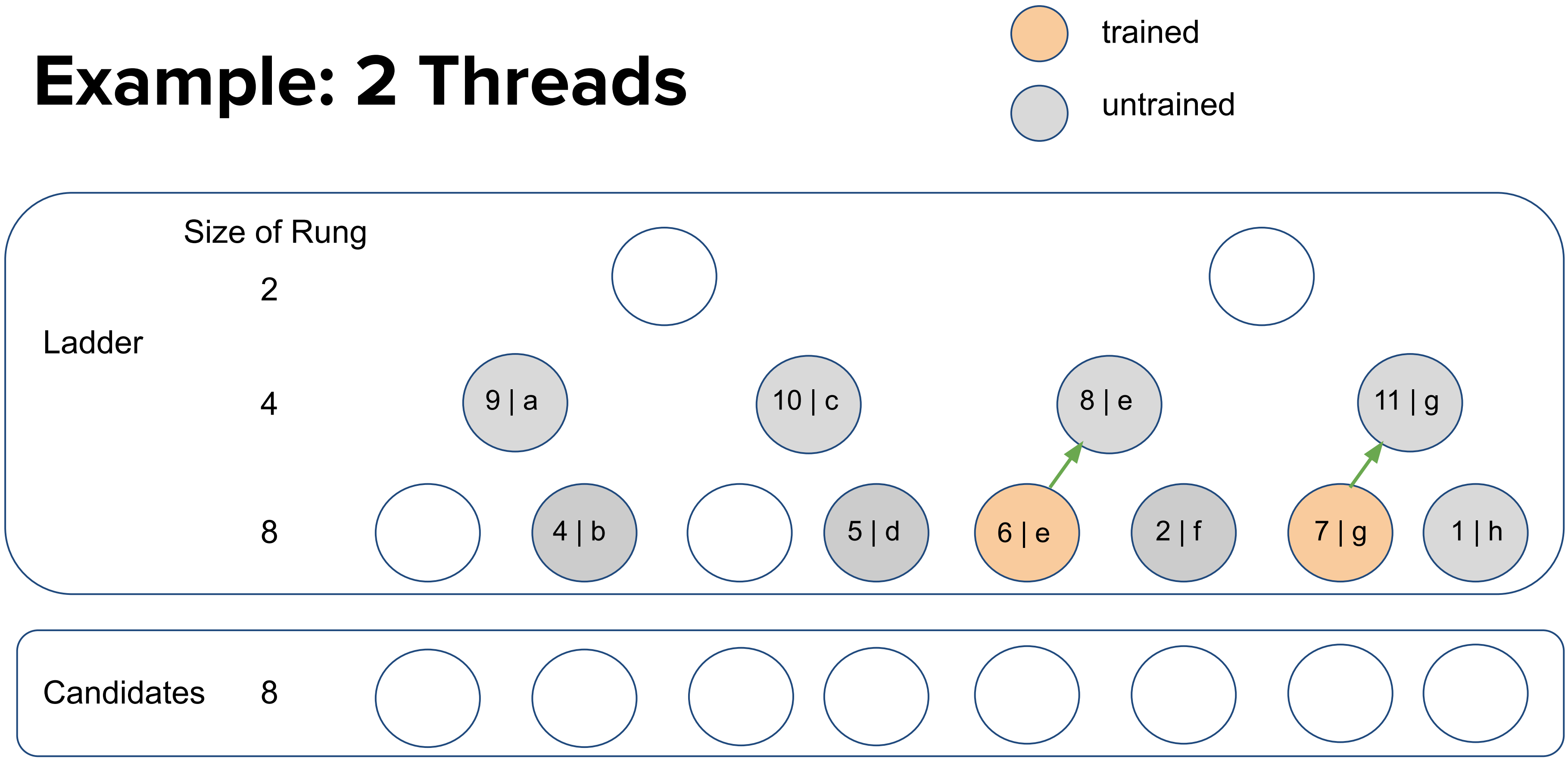
trained

untrained



Ladder: vector of maxHeaps || Candidates: vector

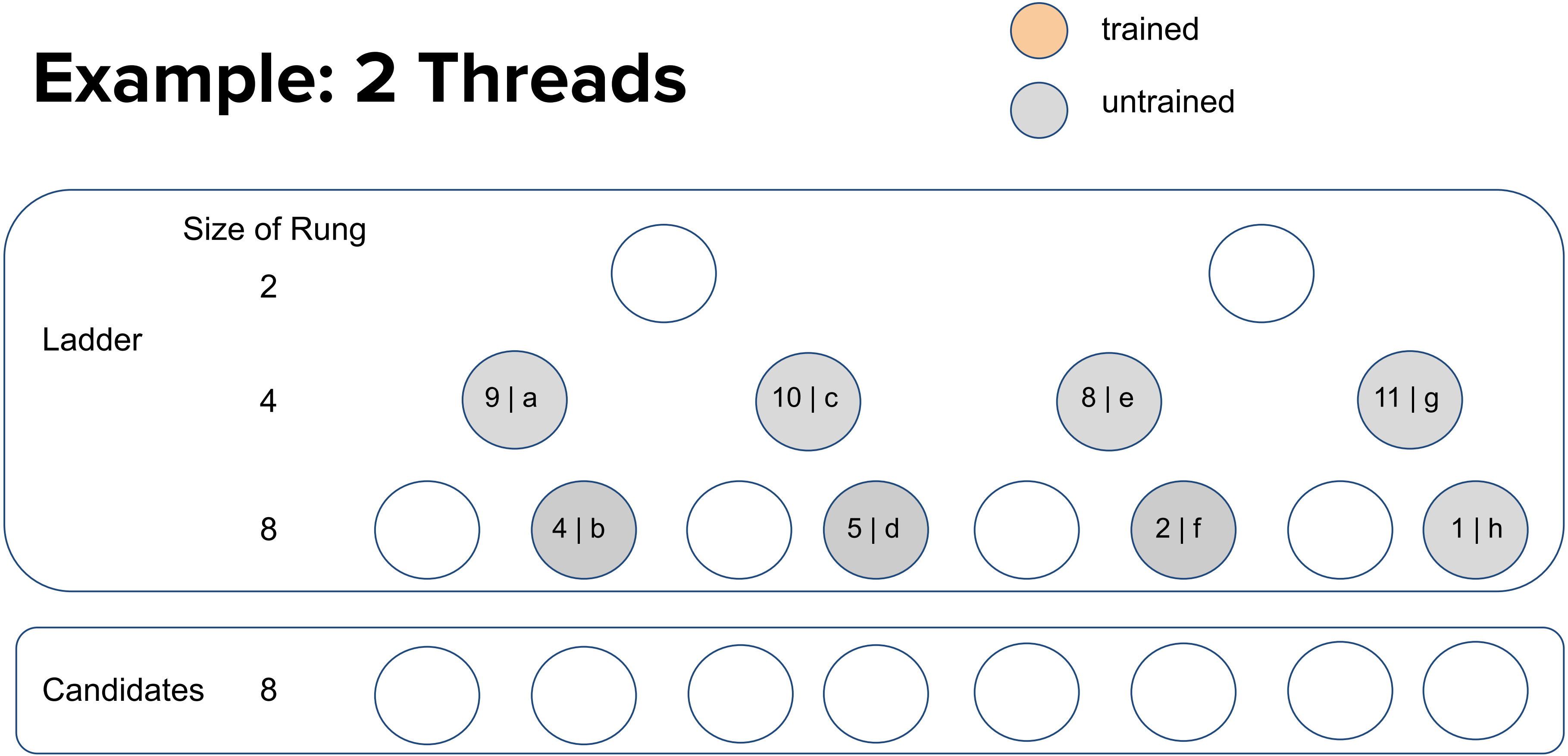
Example: 2 Threads



Ladder: vector of maxHeaps || Candidates: vector

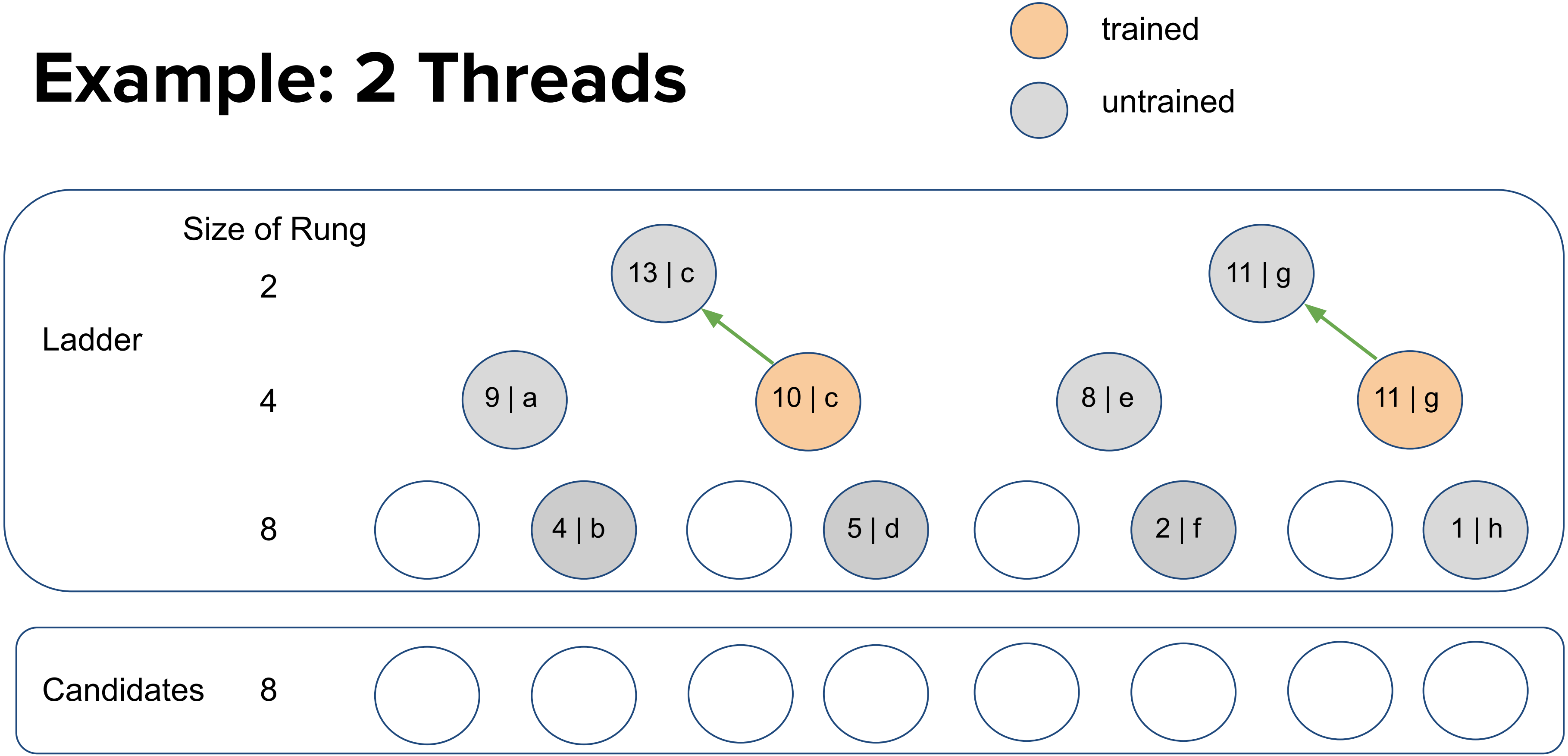
Harvard John A. Paulson School of Engineering and Applied Sciences

Example: 2 Threads



Ladder: vector of maxHeaps || Candidates: vector

Example: 2 Threads

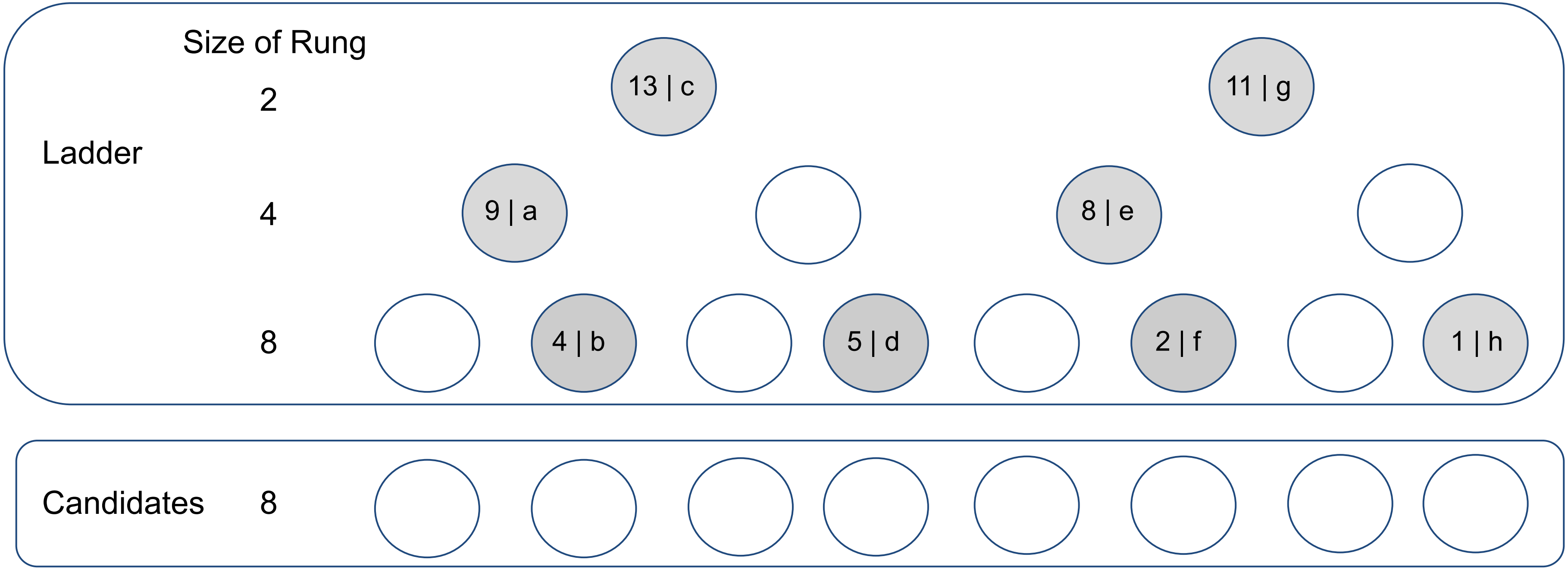


Ladder: vector of maxHeaps || Candidates: vector

Example: 2 Threads

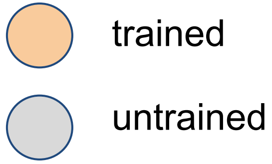
trained

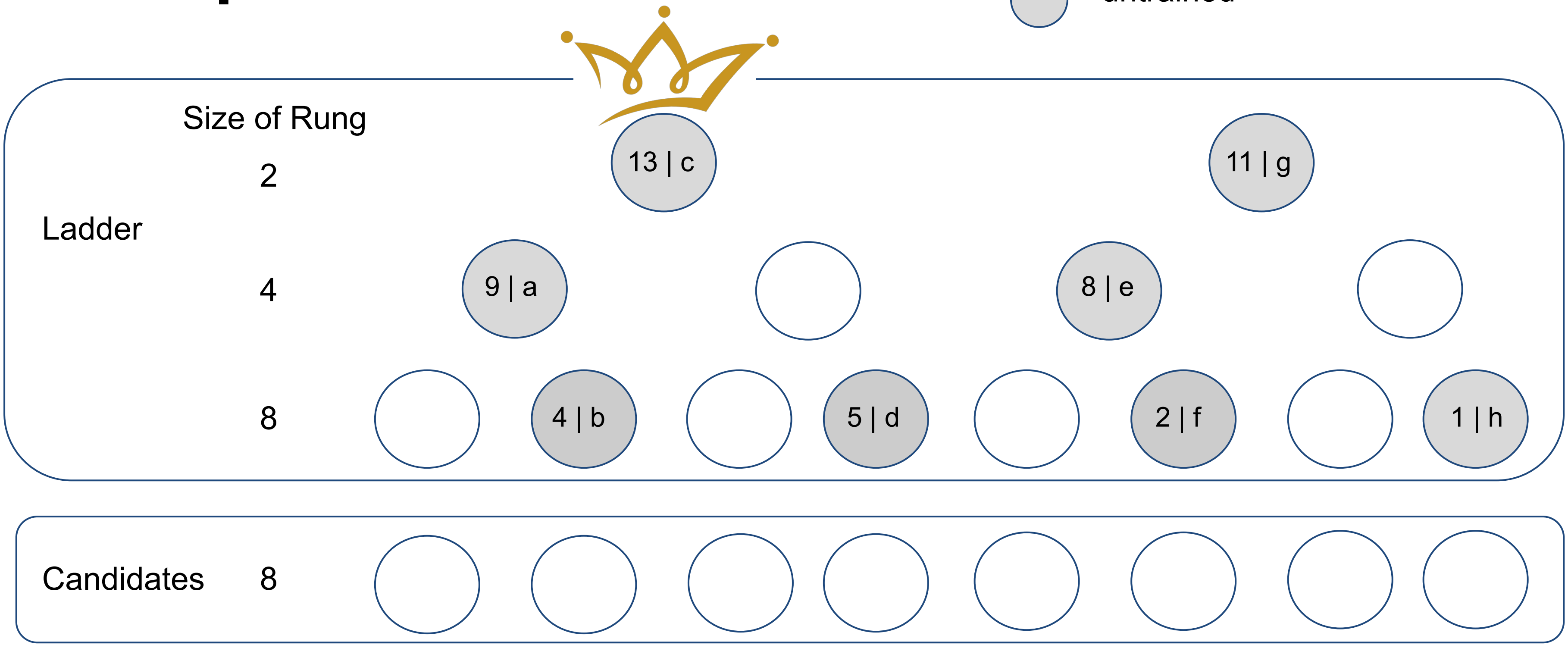
untrained



Ladder: vector of maxHeaps || Candidates: vector

Example: 2 Threads





Ladder: vector of maxHeaps || Candidates: vector

Speedup Findings

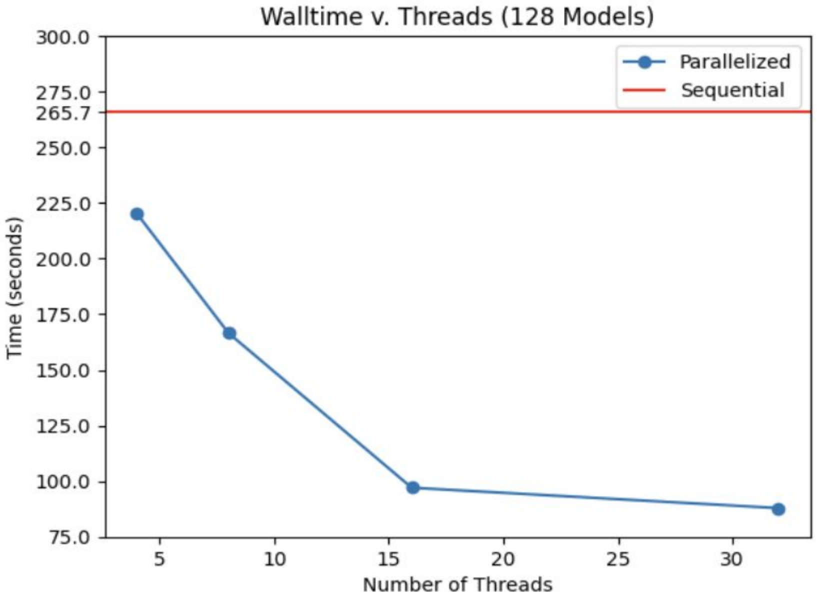
- We kept the memory per core constant (2 GB) to standardize memory access of each thread and observed speedups with varying thread counts:

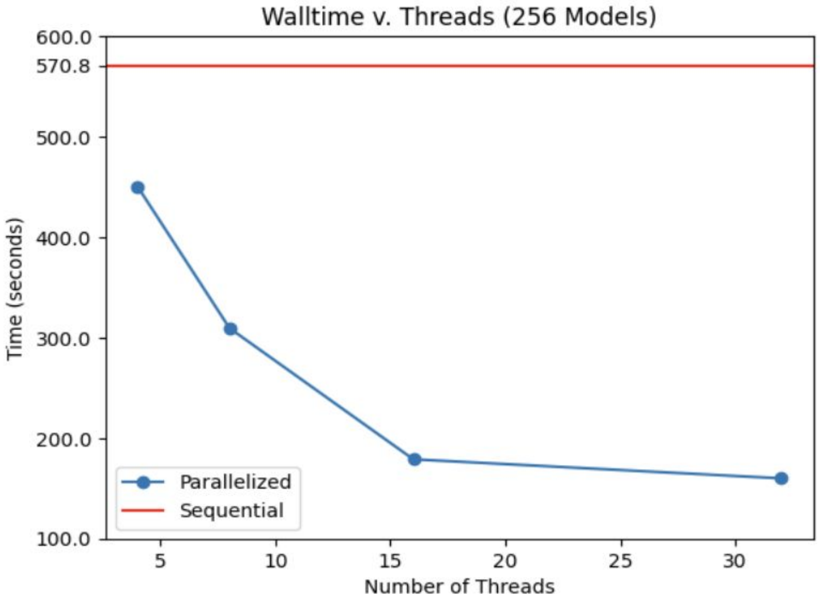
Models		SH	ASH (4T)	ASH (8T)	ASH (16T)	ASH (32T)
128	Walltime (min:sec)	4:25.73	3:40.23	2:46.56	1:37.06	1:27.91
	Observed Speedup	1x	1.21x	1.60x	2.74x	3.02x
256	Walltime (min:sec)	9:30.85	7:30.43	5:09.47	2:58.89	2:39.95
	Observed Speedup	1x	1.27x	1.84x	3.19x	3.57x
512	Walltime (min:sec)	-	16:41.96	11:16.47	8:07.31	5:46.35

Table 2: Runtimes and Speedups for SH and ASH with N threads (T), by number of models

Sequential v. Parallel

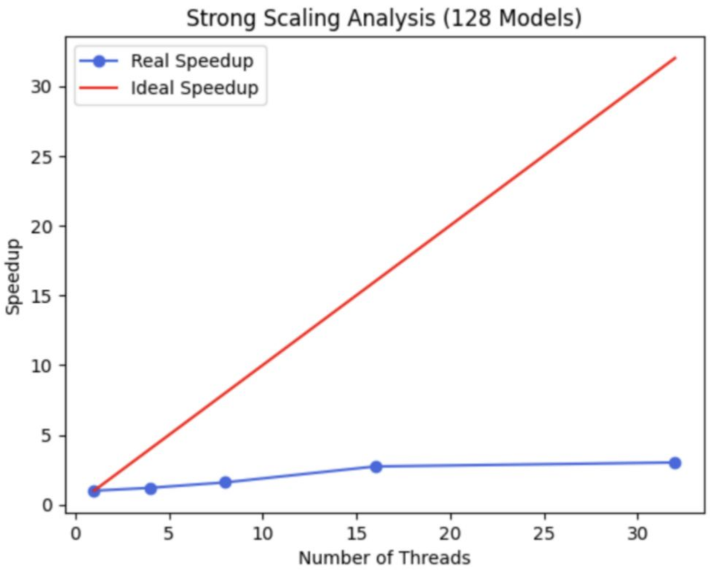
- ASH outperforms SH for threadcount > 1, where the runtime separation grows with more threads

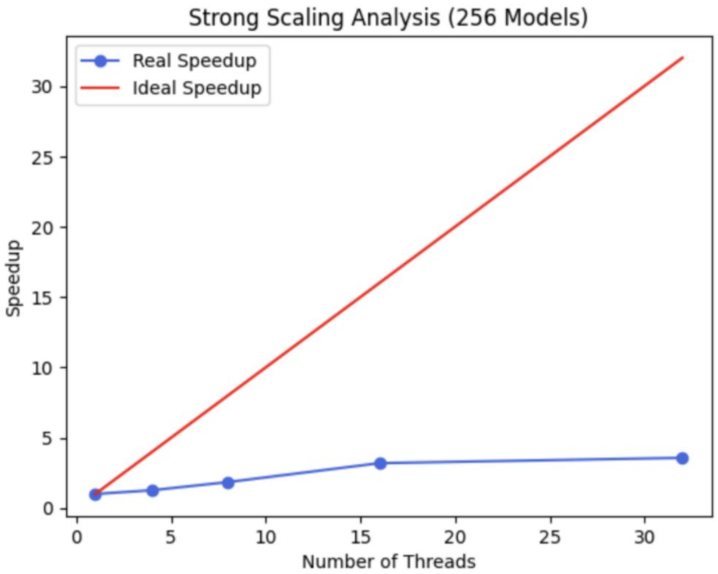




Strong Scaling Plots

- Strong scaling plots include ideal speedup and actual speedup for N threads, calculated by
  Speed-up(N) = Walltime(1) / Walltime(N)





Strong Scaling Analysis

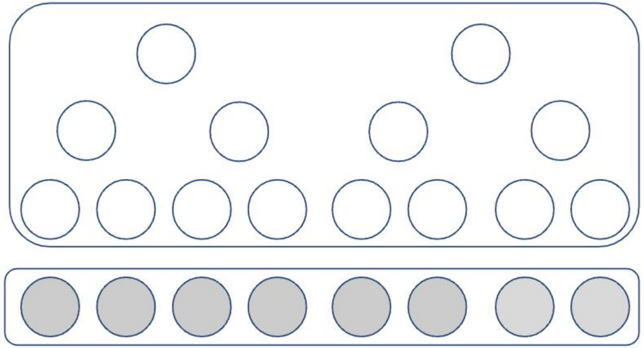
- Why do the plots look the way they do?
- 1. Worksharing Overhead
  - The local runtime of each epoch of model training is significantly higher for ASH than SH
  - This trend can be attributed to OpenMP overhead created by the thread pool and associated shared memory management

Solution Space: 128		Avg Local Runtime (s)	
Epochs	Models	SH	ASH (4 Threads)
1	128	0.35	1.08
2	64	0.62	2.07
4	32	1.2	4.08
8	16	2.26	8.58
16	8	4.45	16.39
32	4	8.91	18.34
64	2	17.63	22.04

Strong Scaling Analysis (cont'd)

- 2. Naive Strong Scaling
  - The ideal speedup is not actually theoretically true and over-simplifies ASH
  - Near the top of the ladder, where epochs are high, resources are not saturated
  - A more accurate formula for strong scaling would be very complex and include a sum in the denominator of Amdahl's Law
- We note that a published implementation only achieved 10x speedup with 25 workers on 1000+ models

Imagine our example from earlier with 8 threads instead of 2... can you see the idle time bottleneck?



Strong Scaling Analysis (cont'd)

- We cite Homework 4 Problem 2c to illustrate factor 2 from the previous slide
- The top of the ladder is bounded similar to part 2 below in the problem
- The complexity of a theoretical speedup of ASH stems from rung prioritization

You wrote a program which is composed of *three* parts executed consecutively:

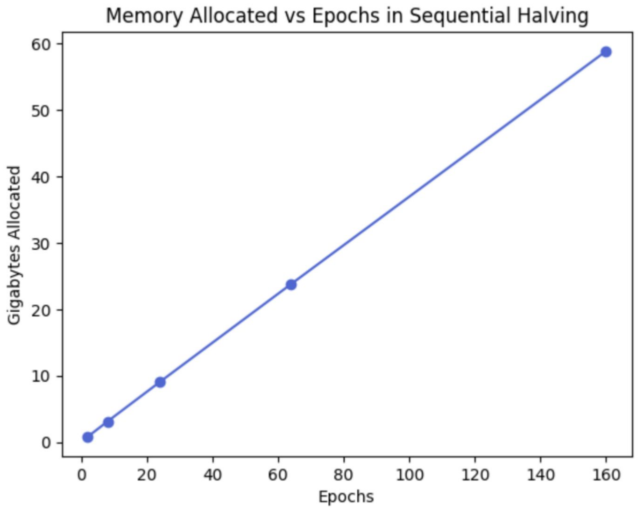
1. A part which you *cannot parallelize*, responsible for a fraction $f_1 = 0.01$ of the total running time.
2. A part which you *can parallelize with only 2 processors*. This part is responsible for $f_2 = 0.04$ of the time.
3. A part that can be parallelized with all processors, occupying the remaining time of the program execution.

According to the above information, what maximum speedup can you achieve, if you had no limitations on the number of processors p? How many processors do you need to obtain a speedup of *at least* 8?

$$8 \leq \frac{1}{0.01 + \frac{0.04}{\min(2,p)} + \frac{0.95}{p}}$$

Memory Analysis

- **Linear relationship** between bytes allocated and number of epochs, and so the number of memory allocated per epoch stays relatively *constant*
- **Scaling:** Running SH or ASH with a larger number of models, such as 1024 or 2048, would result in more epochs, requiring larger amounts of memory resources



Models	Training	Epochs	Bytes allocated	Bytes/Epoch
2	2	2	792,790,098	396395049
4	6	8	3,078,079,270	384759908.8
8	14	24	9,048,497,806	377020741.9
16	30	64	23,789,004,701	371703198.5
32	62	160	58,869,389,053	367933681.6

Resource Analysis

- We used **13.69 total core hours** for our sequential and parallelized code
- Code hours over node hours since we fixed **1 node per simulation** and varied core usage **depending on threadcount**

	SH (128)	SH (256)	ASH (128)	ASH (256)	ASH (512)
Core hours (sequential)	0.07381	0.1586	-	-	-
Core hours (4 threads)	-	-	0.2447	0.5005	1.1133
Core hours (8 threads)	-	-	0.3701	0.6877	1.5032
Core hours (16 threads)	-	-	0.4313	1.1678	2.1657
Core hours (32 threads)	-	-	0.7815	1.4219	3.0785

Table 3: Core Hour Data for SH and ASH Trials

Insights and Future Work

- Learned how to analyze parallelization efficacy using our data through various lenses (seq vs par comparison, strong scaling, discussions)
- **Scalability:** increase memory capacity
- **Productionizability:** continue to generalize our project beyond neural networks, accommodating for other classes of ML models
  - e.g. Decision Trees, Random Forests, etc
- **More parallelization:** parallelizing the training (batch gradient descent) so that one candidate model can have multiple cores simultaneously working on its training (epoch averaging)
  - enable multithreaded MPI

Thank You!

Questions?