Scaling Populations of a Genetic Algorithm for Job Shop Scheduling Problems using MapReduce

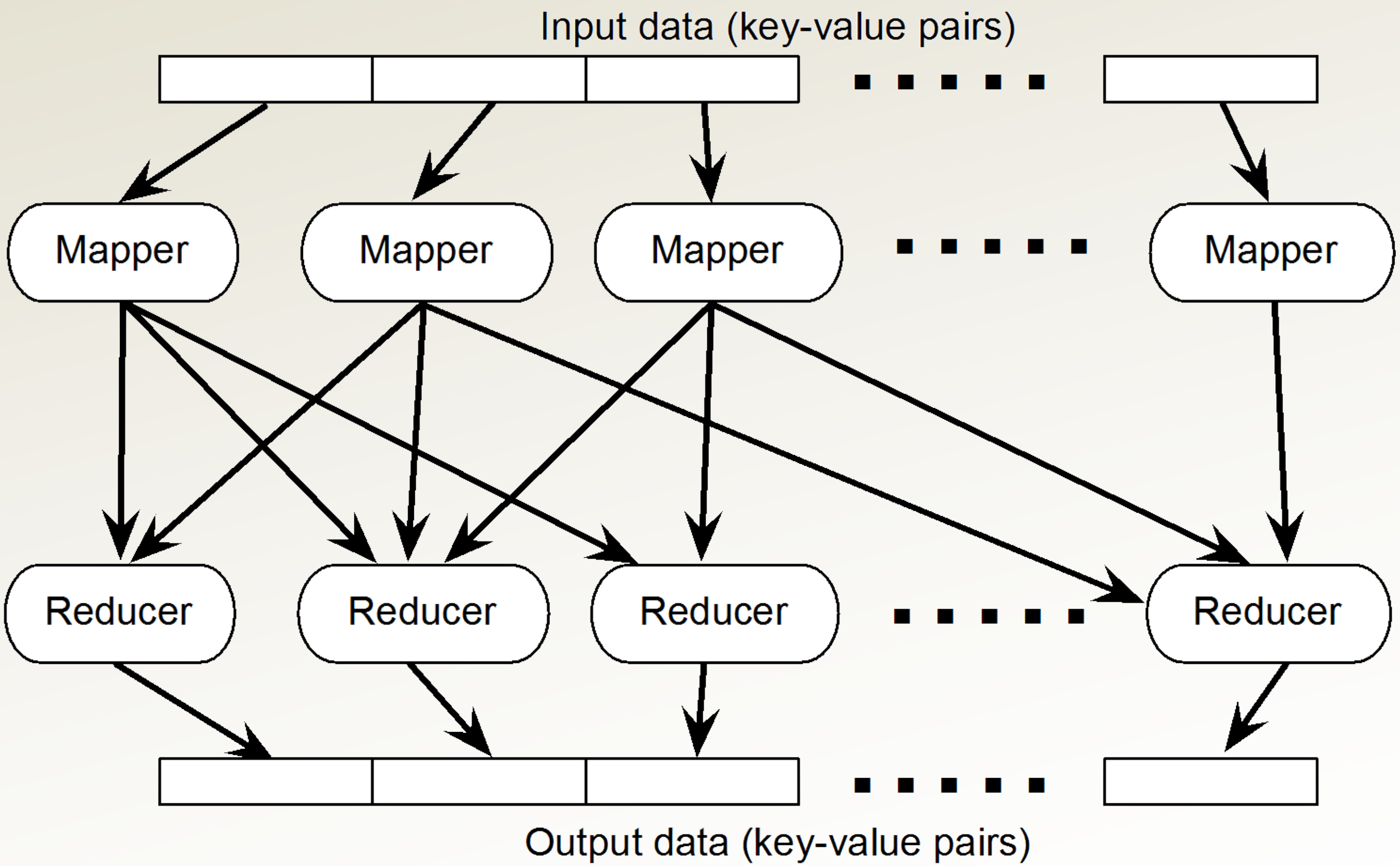
Di-Wei Huang and Jimmy Lin

University of Maryland

Introduction

- Genetic algorithms (GA)
  - Alternative methods for approaching hard problems
  - Inspired by Darwinian evolution, “evolve” a set of potential solutions (“population”) to the problem
- MapReduce
  - Allows us to explore GA’s ability to solve hard problems with much larger populations than typical experiments (a few hundreds)

MapReduce


Input data (key-value pairs)
Mapper
Mapper
Mapper
Mapper
Reducer
Reducer
Reducer
Reducer
Output data (key-value pairs)

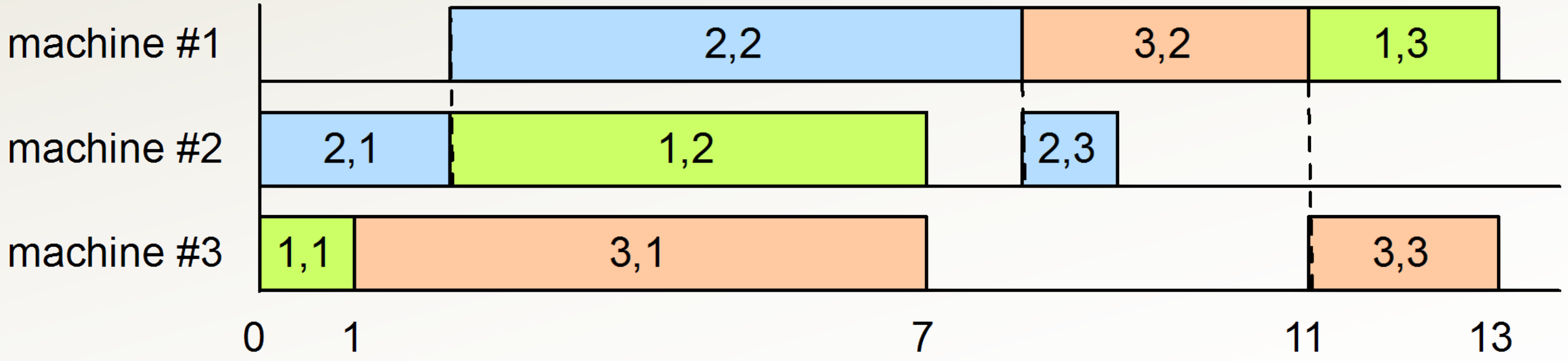
The Problem

- Job Shop Scheduling Problem (JSSP)
  - *M* machines and *J* jobs
  - Each job consists of an ordered list of operations
    - E.g., *M* operations for each job
  - Each operation
    - Requires to be run on a certain machine
    - Requires a certain uninterrupted running time
    - Precedence constraints
  - Goal: minimizing the time required to complete all jobs (i.e., **makespan**)

Example JSSP

- M=3, J=3

	1st op	2nd op	3rd op
Job 1	m=3, t=1	m=2, t=5	m=1, t=2
Job 2	m=2, t=2	m=1, t=6	m=3, t=1
Job 3	m=3, t=6	m=1, t=3	m=2, t=2



JSSP

- Applications in operation research
- NP-hard
  - A generalization of TSP
- No exact solution so far
- Heuristics
  - Large-scale GA with MapReduce

GA Overview

1. Population initialization
   – Each individual encodes a feasible schedule
2. Fitness evaluation
   – Computing the makespan of each individual
3. Selection & Reproduction
   – Individuals with shorter makespan are given higher probabilities to reproduce
   – Crossing over good individuals to generate a new population (the next generation)

GA with MapReduce

- Each generation of GA is run by an iteration of MapReduce
  - Mapper: fitness evaluation (Step 2)
  - Reducer: selection & reproduction (Step 3)
- Initialization (Step 1) is run by a separate, mapper-only MapReduce job

Representation

- Encoding schedules as strings
  - Strings: chromosomes
- Chromosome as ordered list of operations
  - A schedule can be built by inserting operations in the specified order
  - Example chromosome:
    - J=3, each has 3 operations
    - [1, 2, 2, 1, 3, 3, 3, 2, 1] – encode by job numbers
    - #occurrences of a job number determine specific operations

Data Structure

- Key-value pair for mappers and reducers

Key	Value		
ID	Makespan	Generation	Chromosome
double	int	int	int[]

  - ID: random [0, 1)
  - Makespan: fitness value
  - Generation: which generation does this individual belong to?

Initialization

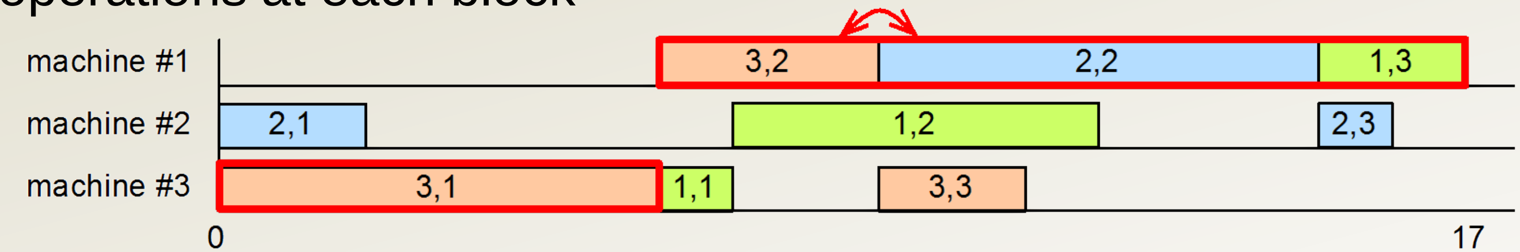
- Good initial population reduces the number of generations
  - Starting a new iteration of MapReduce is expensive
- [Giffler & Thompson, 1960]
  - Random active schedules
    - Subset of all possible schedules
    - The optimal schedule is active
  - Separate mapper-only MapReduce job

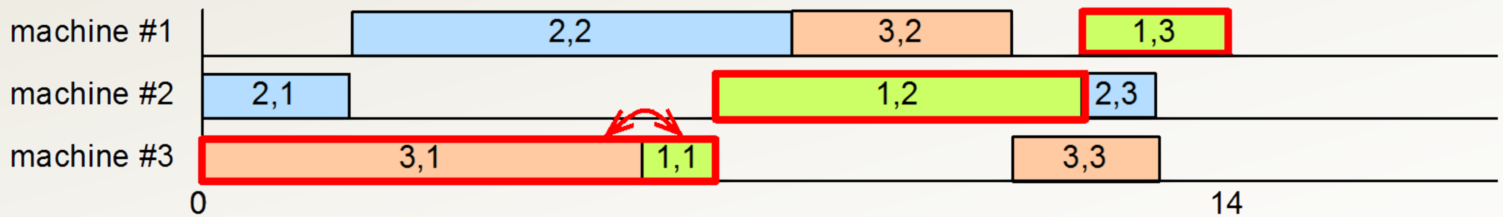
Mapper: Fitness Evaluation

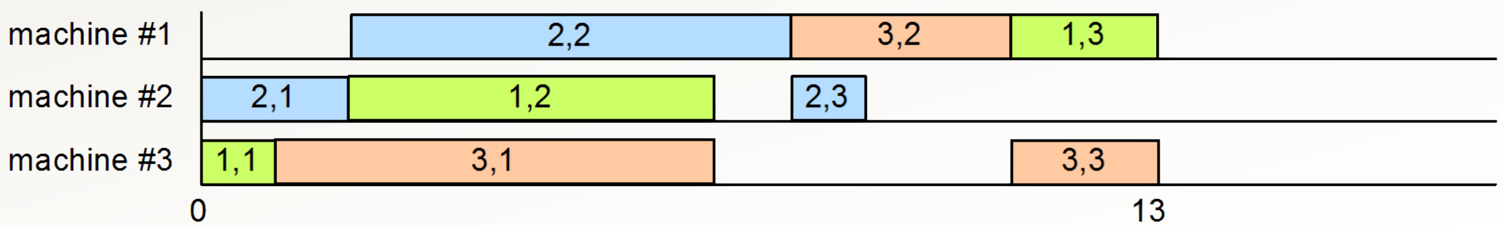
- Building schedules
  - Inserting operations at the earliest available spot in schedule, in the order specified by the chromosome
  - Computing makespan
- Local search (to reduce #generations)
  - Swapping operations on critical path [Nowicki & Smutnicki, 1996]
- Best individual the mapper has seen
  - Make a copy, ID = null

Local Search Example

- Identifying critical paths and swapping the first and/or last pairs of operations at each block





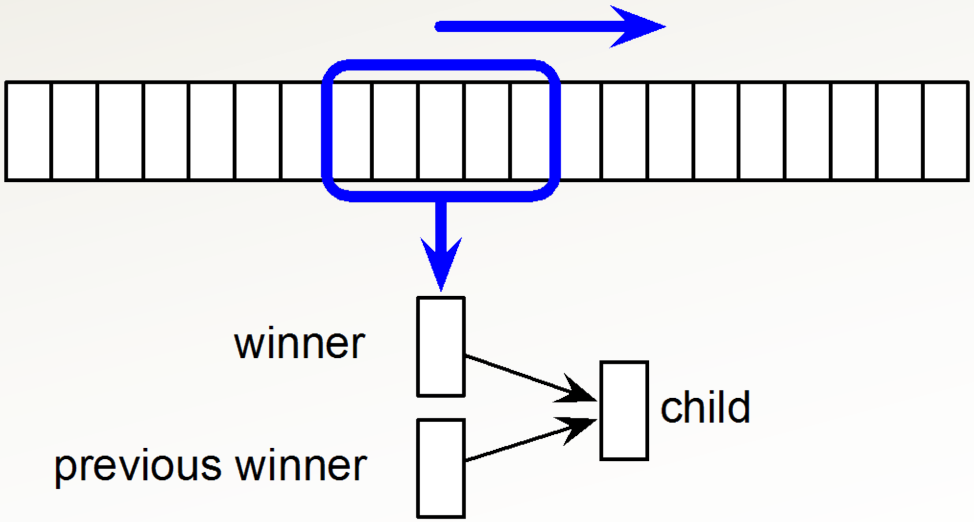


Partitioner

- If ID == null, send to Reducer #0
  - Best individuals reported by each mapper are sent to Reducer #0
- Otherwise, send to Reducer #h(ID)%r
  - h: hash function
  - r: number of reducers
  - IDs are randomly generated, so individuals are sent to a random reducer

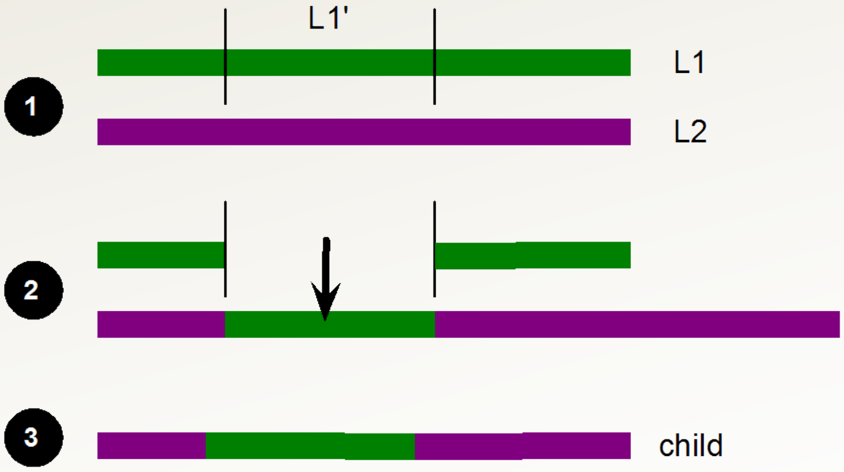
Reducer: Selection & Reproduction

- Tournament selection
  - Randomly pick *s*=5 individuals and select the fittest among them for reproduction
- Sliding window-based approximation [Verma et al, 2009]
  - Random ID ➔ Arbitrarily ordered list



Reproduction

- Crossover (parent chromosome L1, L2) [Park et al, 2003]
  - Randomly select a segment from L1
  - Insert L1' to L2
  - Remove redundant operations from L2



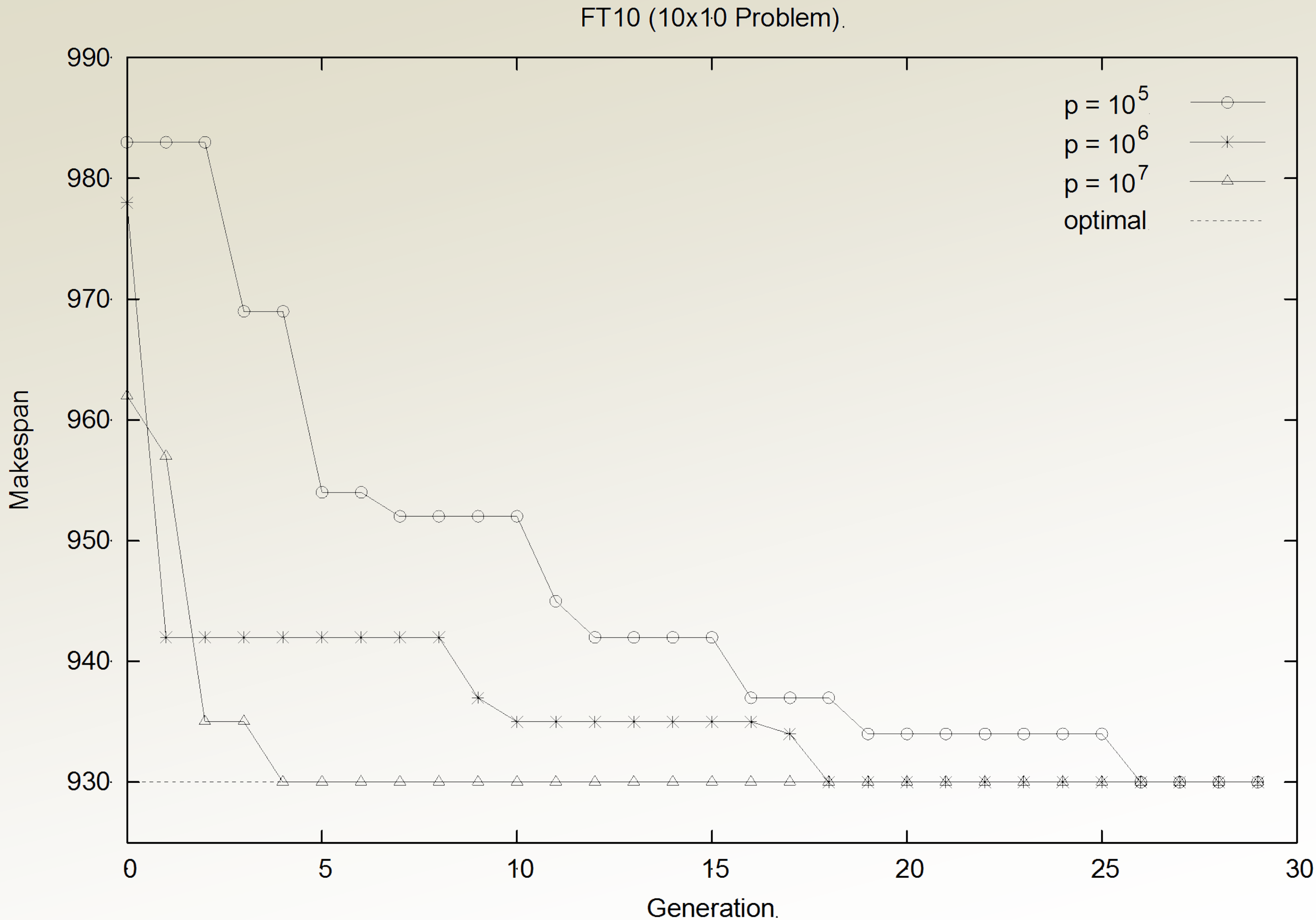
- Mutation
  - 1%
  - Importance of mutation decreases as population grows

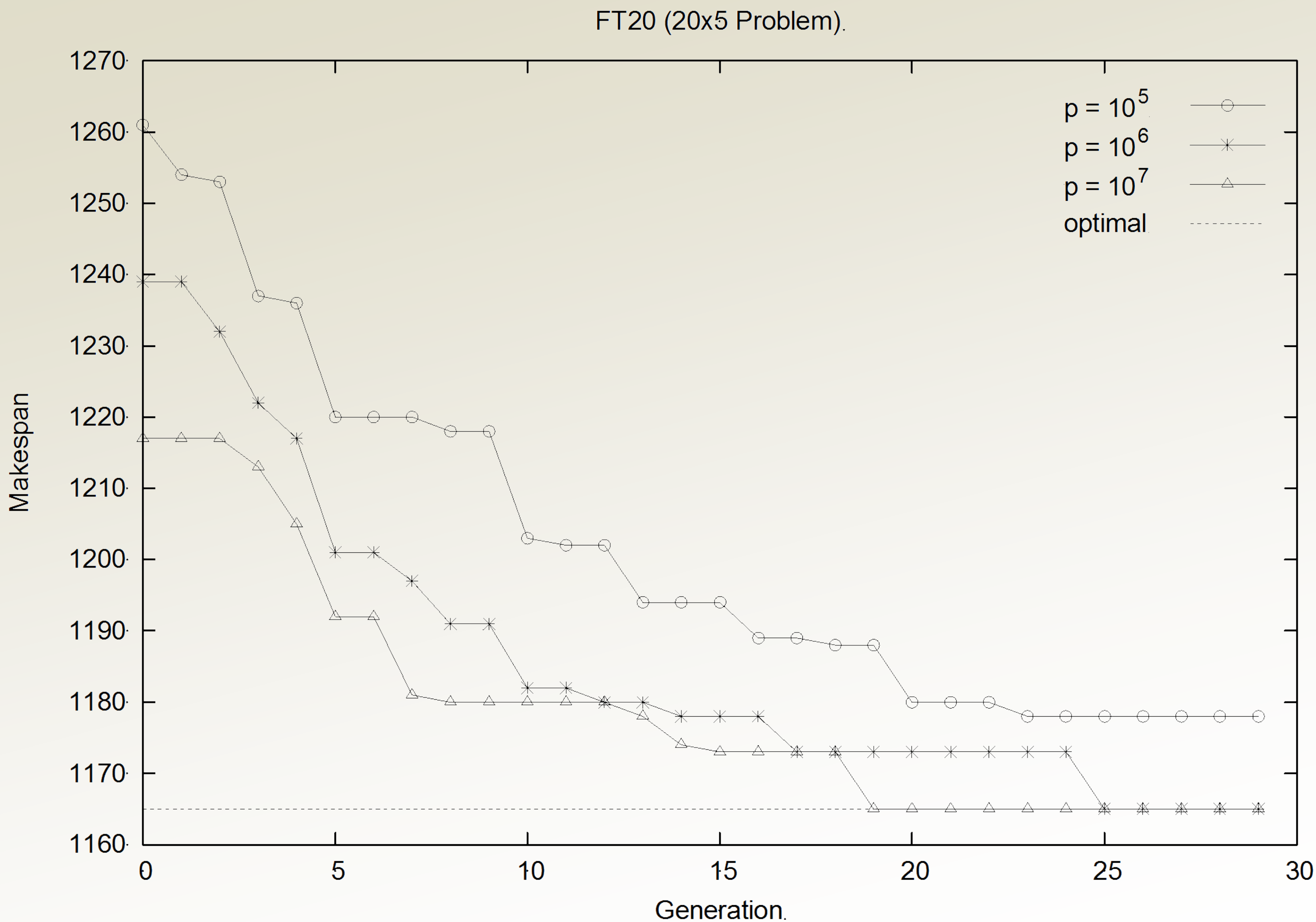
Experiment (1)

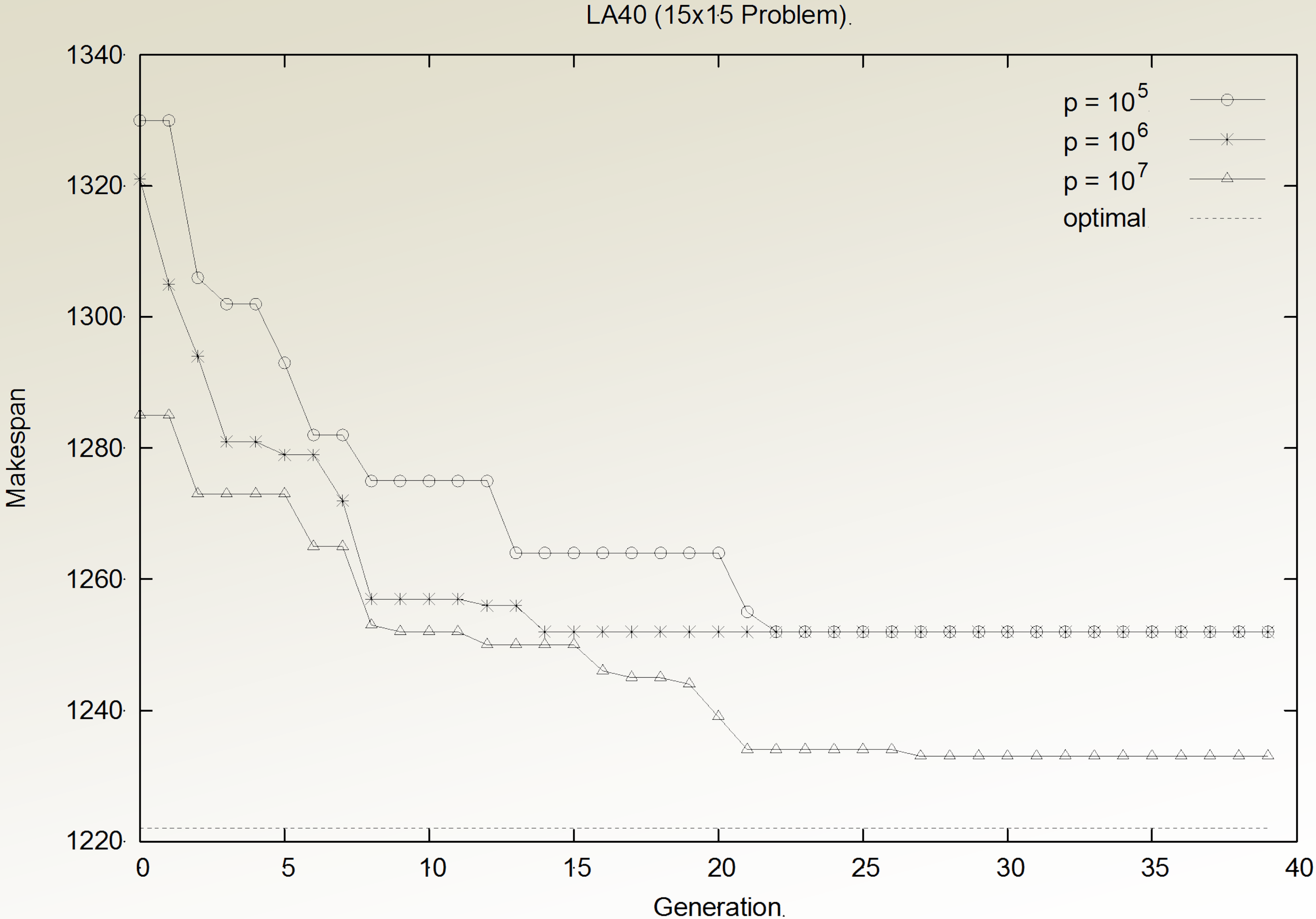
- JSSP instances

Name	#Jobs	#Machines	Optimal Makespan
FT10	10	10	930
FT20	20	5	1165
LA40	15	15	1222
SWV14	50	10	2968

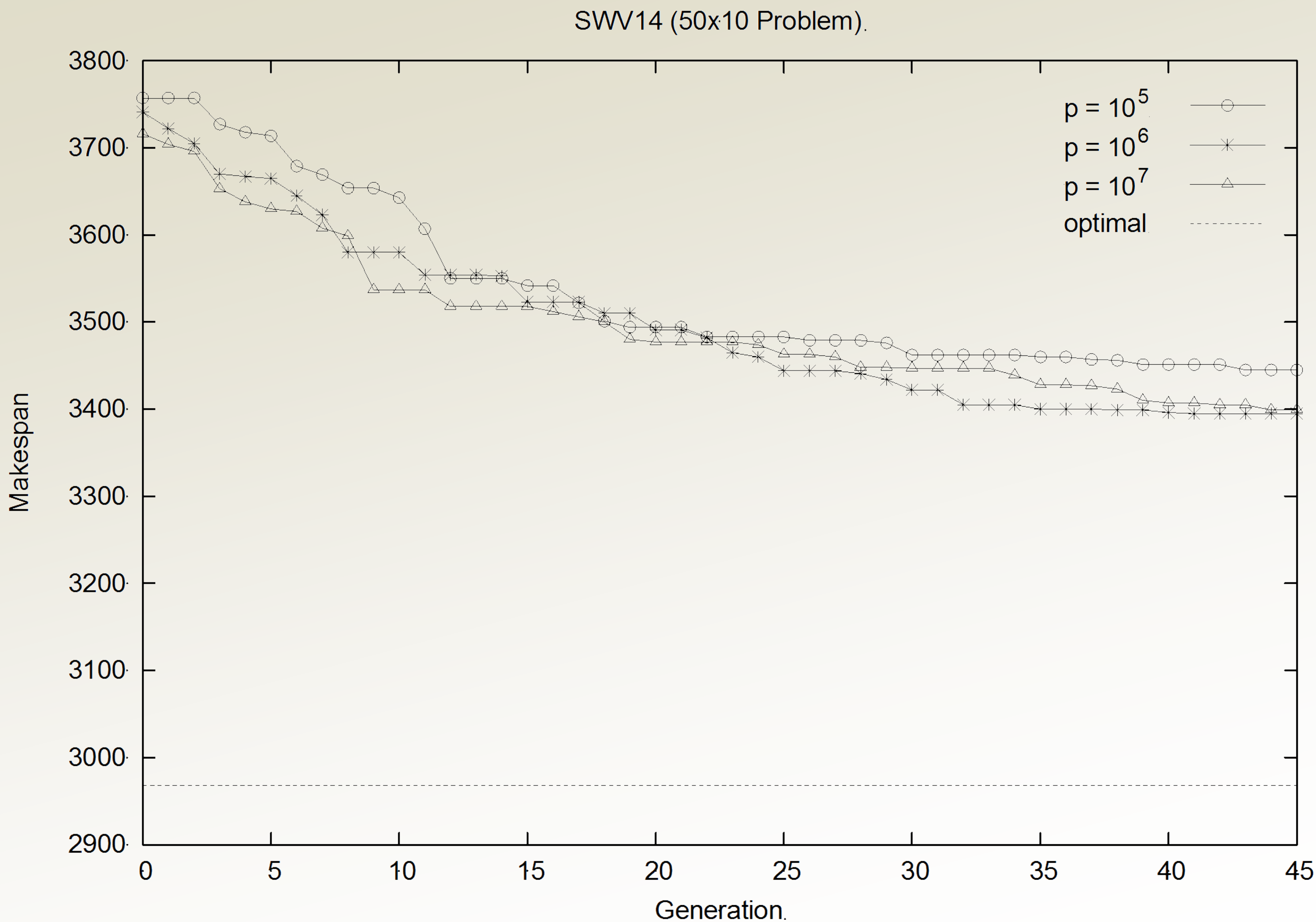
- The cluster Part of NSF's CLuE Program and Google/IBM Academic Cloud Computing Initiative
  - 414 physical nodes, each with 2 single-core processors, 4GB memory, 400GB hard drives
  - Run with 1000 mappers and 100 reducers


FT10 (10x10 Problem)
990
980
970
960
950
940
930
Makespan
0
5
10
15
20
25
30
Generation
p = 10^5
p = 10^6
p = 10^7
optimal


FT20 (20x5 Problem).
Makespan
Generation
p = 10^5
p = 10^6
p = 10^7
optimal
1270
1260
1250
1240
1230
1220
1210
1200
1190
1180
1170
1160
0
5
10
15
20
25
30

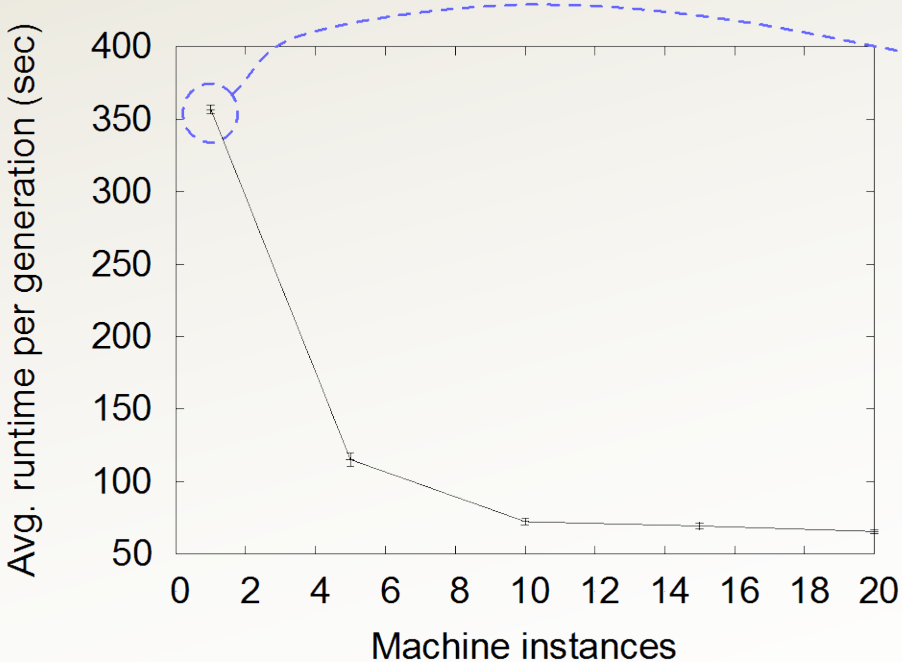


LA40 (15x15 Problem)
1340
1320
1300
1280
1260
1240
1220
Makespan
0
5
10
15
20
25
30
35
40
Generation
p = 10^5
p = 10^6
p = 10^7
optimal


SWV14 (50x10 Problem)
Makespan
Generation
p = 10^5
p = 10^6
p = 10^7
optimal
3800
3700
3600
3500
3400
3300
3200
3100
3000
2900
0
5
10
15
20
25
30
35
40
45

Experiment (2)

- Effects of cluster size (1 – 20)
  - Amazon EC2
- LA40 with population size 10,000



total	356.1 sec
map()	331.2 (93%)
reduce()	7.2 (2%)
overhead	17.7 (5%)

Conclusion

- Implementation of GA with modern features tackling a real-world problem using MapReduce
- Larger population (up to 10^7)
  - Better solution to JSSP
  - Fewer generations (good for MapReduce)
  - Tradeoffs between #generations (sequential) and population size (parallel)
- Effects of cluster sizes
  - A rough guideline to choose cluster size