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
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

- \( M=3, J=3 \)

<table>
<thead>
<tr>
<th>Job 1</th>
<th>1st op</th>
<th>2nd op</th>
<th>3rd op</th>
</tr>
</thead>
<tbody>
<tr>
<td>m=3, t=1</td>
<td>m=2, t=5</td>
<td>m=1, t=2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Job 2</th>
<th>m=2, t=2</th>
<th>m=1, t=6</th>
<th>m=3, t=1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Job 3</th>
<th>m=3, t=6</th>
<th>m=1, t=3</th>
<th>m=2, t=2</th>
</tr>
</thead>
</table>

- Machine #1
  - 2,2
  - 3,2
  - 1,3

- Machine #2
  - 2,1
  - 1,2
  - 2,3

- Machine #3
  - 1,1
  - 3,1
  - 3,3

0 1 7 11 13
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

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Makespan</td>
</tr>
<tr>
<td>double</td>
<td>Generation</td>
</tr>
<tr>
<td>int</td>
<td>Chromosome</td>
</tr>
<tr>
<td>int[]</td>
<td></td>
</tr>
</tbody>
</table>

- 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

• Best individual the mapper has seen
  – Make a copy, ID = null

[Nowicki & Smutnicki, 1996]
Local Search Example

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

```
<table>
<thead>
<tr>
<th>Machine #1</th>
<th>Machine #2</th>
<th>Machine #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>3,2</td>
<td>1,1</td>
</tr>
<tr>
<td>2,2</td>
<td>1,2</td>
<td>3,3</td>
</tr>
<tr>
<td>1,3</td>
<td>2,3</td>
<td>3,3</td>
</tr>
</tbody>
</table>
```

0 17 14 13
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
  - Random ID ➔ Arbitrarily ordered list

[Verma et al, 2009]
Reproduction

• Crossover (parent chromosome L1, L2)
  – Randomly select a segment from L1
  – Insert L1’ to L2
  – Remove redundant operations from L2

• Mutation
  – 1%
  – Importance of mutation decreases as population grows

[Park et al, 2003]
Experiment (1)

- **JSSP instances**

<table>
<thead>
<tr>
<th>Name</th>
<th>#Jobs</th>
<th>#Machines</th>
<th>Optimal Makespan</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT10</td>
<td>10</td>
<td>10</td>
<td>930</td>
</tr>
<tr>
<td>FT20</td>
<td>20</td>
<td>5</td>
<td>1165</td>
</tr>
<tr>
<td>LA40</td>
<td>15</td>
<td>15</td>
<td>1222</td>
</tr>
<tr>
<td>SWV14</td>
<td>50</td>
<td>10</td>
<td>2968</td>
</tr>
</tbody>
</table>

- **The cluster**
  - 414 physical nodes, each with 2 single-core processors, 4GB memory, 400GB hard drives
  - Run with 1000 mappers and 100 reducers

Part of NSF’s CLuE Program and Google/IBM Academic Cloud Computing Initiative
SWV14 (50x10 Problem).

- \(p = 10^5\)
- \(p = 10^6\)
- \(p = 10^7\)
- optimal

Y-axis: Makespan
X-axis: Generation
Experiment (2)

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