Experimental Approaches in Computer Science

Dror Feitelson
Hebrew University

Lecture 12 – Experimental Algorithmics
Case studies

- Online scheduling
- Matrix multiplication
- Maximum flow
Online scheduling
Problem definition: Given $n$ jobs with known processing times assign them to $m$ identical machines so as to minimize the makespan.

- Graham's list scheduling [1966]: assign each job to the machine with the least assigned load so far.

- Claim: Graham's simple greedy algorithm is $(2 - \frac{1}{m})$-competitive.

Online: assign each job before you know about subsequent jobs.
Proof:
Let $c^*$ denote the optimal makespan then $c^* \geq p_{\text{max}}$ [accommodate longest job]
and $c^* \geq 1/m \sum p_j$ [accommodate total processing needed]

assume job $k$ is the last one to terminate then it starts no later than $1/m \sum_{j \neq k} p_j$
because no machine is idle before all jobs start
Its termination time is then no later than its start time + processing time:

\[ c_k \leq \frac{1}{m} \sum_{j \neq k} p_j + p_k \]
\[ \leq \frac{1}{m} \sum_j p_j + (1 - \frac{1}{m})p_k \]
\[ \leq c^* + (1 - \frac{1}{m})c^* \]
\[ = (2 - \frac{1}{m})c^* \]
Worst case: many small jobs followed by one long job

Improvements:

- Bartal et al. [1995]: 1.986-competitive algorithm
- Karger et al. [1996]: 1.945-competitive algorithm
- Albers [1997]: 1.923-competitive algorithm
- All use various conditions to sometimes select a machine that is not the least loaded for short jobs (leaving the least loaded for the long job)
- Question: is this generally good, or does it just avoid certain pathological cases?
Experimental evaluation:
[Albers & Schroder, J. Exp. Alg. 7(3), 2002]

- Use real-world job sizes
  - Parallel machines (MPPs at CTC, KTH)
  - Vector machine (Cray at PSC)
  - Workstation (Sun in Germany)
- Use distributions
- Create sequences of 10000 jobs, and tabulate running ratio of achieved makespan to optimal for m=10
Results KTH:

relatively low variance, so ratio stabilized after some fluctuations; Graham is best
Results Cray:

Occasional big job similar to average so far.

Graham suffers because loads are balanced, and one machine will need to work much more; others leave machines less loaded in anticipation of such jobs.
job sizes have a heavy tail: some are so big they dominate the average. This causes both the online algorithm and the optimal makespan to be essentially equal, and the ratio drops to 1
Exponential:

Relatively low variability leads to quick convergence.

Similar results for uniform, Erlang, and hyperexponential with various parameter values
Effect of number of machines ($m$):

- All previous results were for $m=10$
- When $m$ grows, it takes longer for ratios to stabilize, because more jobs are needed to fill the machines
- Also, the effect of jobs that are similar to the average load is changed – given that the load is distributed on more machines, these jobs now look huge, and their effect is to reduce the ratio rather than to enlarge it
The bottom line: it depends on the workload

- Graham's simple greedy algorithm is best when job variance is low
- Other algorithms, mainly Albers and Bartal, may reduce sensitivity to large jobs
- When the variance is extremely big due to a heavy tail, the algorithm has little effect
Matrix Multiplication
Problem definition:

- Use the straightforward $n^3$ algorithm
- Take into account the memory hierarchy
  - Cache capacity
  - Cache associativity
  - Contention for the system bus
  - Memory latency
- An instance of algorithm engineering

[0x et al. J. Exp. Alg. 4(3), 1999]
Idea 1: use tiling

- Use tiles that fit into the cache, to avoid capacity misses
- Retain ratio of multiple operations per given data
Idea 2: use prefetching

- In each phase prefetch the data needed in the next phase
- If all data is in the cache, computation does not use the system bus at all
- Bus is therefore free for use by prefetching
- Need to time the prefetches so as to avoid evicting needed data (assumes LRU cache replacement)
Tile size constraints

- Computation per tile multiplication is $O(P_1 P_2 P_3)$
- Data to prefetch is $O(P_1 P_2 + P_2 P_3 + P_1 P_3)$
- Also need to write back $C$ tile of $P_1 P_3$
- Enough time if $P_1 P_2 P_3 > P_1 P_2 P_3 + 2P_1 P_3$
- Enough space if $2(P_1 P_2 + P_2 P_3 + P_1 P_3) < C$
- Can reduce prefetching/writeback by reusing $C$ tile for full row of $A$ tiles and column of $B$ tiles
Idea 3: copy to avoid conflicts

- Copy tiles to different addresses so that they fall in different cache associativity sets
- Assuming $k$-way associativity, ensure that each set is used only $k/2$ times
- Simple example:
  - 2-way associativity
  - Interleave tiles from the different matrices
  - Use offset that is a multiple of the way size
  - Being 2-way allows 2 tiles from each matrix to be cache resident
Implementation:

- IBM PowerPC model 604
- Use fma (floating multiply-add) instruction, which is ideal for matrix/vector multiplication
  - Theoretical peak of 266 MFLOPS
- Don't use dcbt (data cache block touch) instruction for prefetching, but rather a register load
  - dcbt doesn't work when TLB misses
  - Can't be triggered from source level
Performance:
better and more predictable than highly tuned code
Maximum Flow
Problem definition:

given a graph \( G = (V, E) \),
with two distinguished nodes \( s \) and \( t \),
where each edge \( e \) has capacity \( c(e) \),
find the maximum possible flow from \( s \) to \( t \)

we'll focus on unit capacity (\( c(e) = 1 \) for all edges)
Flow definition:

A flow is a function \( f : V \times V \rightarrow \mathbb{R} \) such that

- \( f(u, v) \leq c(u, v) \) [capacity constraint]
- \( f(u, v) = -f(v, u) \) [anti-symmetry]
- \( \sum_v f(u, v) = 0 \) [conservation constraint]

(holds for all \( u \) except \( s \) and \( t \))

The value to maximize is \( \sum_v f(s, v) \)
Main algorithms:

- Path augmentation
- Preflow push-relabel
Path augmentation

- Invariant: always maintain a legitimate flow
- Start with a 0 flow
- At each step
  - Find a path from $s$ to $t$ that has capacity to spare
  - Add a flow along this path
- Terminate when no additional paths can be found
- Complexity: $O(E \ |f|)$ with integer capacities, $|f|$ is max

Variants: BFS? DFS?
Preflow push-relabel

- Invariant: maintains a preflow (allow excess input to a node)
- Initially $s$ is at level $|V|$, $t$ and all others at 0
- For all overflowing nodes (starting with $s$) fill outgoing links to nodes at lower level to capacity
- If all unsaturated outbound links are to nodes at same or higher level, relabel the node to level one higher than lowest unsaturated neighbor
- At end, nodes with excess flow will migrate to above the source and push the excess

- Complexity: $O(V^2 E)$

Variants: order of push and relabel ops, use of optimizations
Optimizations:

• Global relabel
  – Push and relabel are local operations
  – State may drift away from global optimum
  – Optimization is to do a global scan and relabel all nodes consistently in one sweep

• Gap heuristic:
  – If there are no nodes with label $d$, all those with higher labels return excess to $s$
  – Saves the need to raise their level by single steps to above $|V|$
Experimental questions:

• Augment or push?

• What is the effect of variants and optimizations?

• How does this depend on different input graph instances?

[Cerkassky et al. J Exp. Alg. 3(8), 1998]
Methodology: use random graphs from various different families
### Experimental results

**Table 1. Summary of results. Blank is good, o is fair, and • is poor.**

<table>
<thead>
<tr>
<th></th>
<th>DFS</th>
<th>BFS</th>
<th>LDS</th>
<th>AR</th>
<th>FIFO</th>
<th>LO</th>
<th>HI</th>
</tr>
</thead>
<tbody>
<tr>
<td>fewg</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>manyg</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hi-lo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>grid</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hexa</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rope</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>zipf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>karz</td>
<td></td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>rmfuC</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmfuL</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rmfuW</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blow</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>puff</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>saus</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>squa</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wave</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experimental results

Plots for graph families
Lines for algorithms
Conclusions:

- No single algorithm is best for all graph types.
- Both BFS and DFS (path augmentation) are not robust, with bad performance for many graph families.
- The best push-relabel methods are generally more robust than the best augmented flow.
- The added heuristics are important for the achieved performance.