Multicore Parallel Min-Cost Flow Algorithm for CAD Applications

Hai Zhou
(Joint work with Yinghai Lu, Li Shang and Xuan Zeng)
Electrical Engineering and Computer Science
Northwestern University

July 7, 2009
Multicore Revolution

- Since 2004, $\mu$P frequency scaling has been flattened
- Only more cores in new generations
- Applications will not speed up automatically
- Who wants to upgrade?
- We are all doomed if computers are like washing machines
  - No industry growth
  - No exciting projects
  - No funding
CAD Challenges

- CAD problems are huge
- CAD problems are computationally intensive
- CAD software traditionally depends heavily on frequency scaling
Parallel programming is the only rescue!

Parallel programming is very very difficult!

- Automated parallelization is in general a failure
- Message passing based programming is too low level
- Multithreading is hard to get right due to data racing
Thinking Parallel

To get parallel, we have to think parallel

- With a small skull, we cannot think about true parallel
  - Number of possible scenario are exponential
- Nondeterministic Transactional Model is the best possible
Nondeterministic Transactional Model in UNITY

- An algorithm is an initialization followed by a loop
- The loop is an iterative execution of any command with a true guard
- Execution is atomic (i.e. a transaction)
- Order of execution is arbitrary (nondeterministic)
Ancient Wisdom

- Euclid’s GCD Algorithm \((a, b \in \mathbb{N})\)

Euclid’s alg.  

\[
\begin{align*}
x, y & := a, b \\
\textbf{do} & /* GCD(\textbf{x}, \textbf{y}) = GCD(a, b) */ \\
& \quad x > y \rightarrow x := x - y \\
& \quad x < y \rightarrow y := y - x \\
\textbf{od} & /* GCD(\textbf{x}, \textbf{y}) = GCD(a, b) \land x = y */ \\
& \text{output } x
\end{align*}
\]
Min-Cost Flow Problem

Timing-constrained optimization problems in CAD:

\[
\text{Min} \quad \sum_{(i,j) \in E} \text{cost}_{ij}(d(i,j)) \\
\text{s.t.} \quad \forall (i,j) \in E : p(i) + d(i,j) \leq p(j)
\]

Dual: min-cost flow problem:

\[
\text{Min} \quad \sum_{(i,j) \in E} w(i,j)f(i,j) \\
\text{s.t.} \quad \forall (i,j) \in E : 0 \leq f(i,j) \leq c(i,j) \\
\forall j \in V : \sum_{(i,j) \in E} f(i,j) = \sum_{(j,k) \in E} f(j,k)
\]
Optimality Condition

- Karush-Kuhn-Tucker condition for min-cost flow

\[ P_0 \triangleq \forall (i, j) \in E : 0 \leq f(i, j) \leq c(i, j) \]

\[ P_1 \triangleq \forall j \in V : \sum_{(i, j) \in E} f(i, j) = \sum_{(j, k) \in E} f(j, k) \]

\[ P_2 \triangleq \forall (i, j) \in E(f) : p(i) - w(i, j) \leq p(j) \]

- \( \epsilon \)-optimality (optimal if \( \epsilon < \frac{1}{|V|} \))

\[ P_2(\epsilon) \triangleq \forall (i, j) \in E(f) : p(i) - w(i, j) \leq p(j) + \epsilon \]
Parallelism by Nondeterministic Transactions

- Valid guarded commands can be executed in parallel if there is no conflict.
- Non-deterministic transactional programming for multicore algorithm design
  - Easy to reason (focus on isolated atomic commands)
  - Guaranteed correctness
  - Rich parallelism
Designing Min-Cost Flow Algorithm

- Post-condition revisited:

\[ P_0 \triangleq \forall (i, j) \in E : 0 \leq f(i, j) \leq c(i, j) \]
\[ P_1 \triangleq \forall j \in V : \sum_{(i, j) \in E} f(i, j) = \sum_{(j, k) \in E} f(j, k) \]
\[ P_2(\epsilon) \triangleq \forall (i, j) \in E(f) : w^p(i, j) \leq -\epsilon \]
\[ P_3 \triangleq P_0 \land P_1 \land P_2(\epsilon) \land \epsilon < 1/|V| \]

where \( w^p(i, j) \triangleq w(i, j) - p(i) + p(j) \).

- Design strategy: use \( P_0 \) as invariant, and all the other conditions as loop goals.
To Satisfy Loop Goals

- For $P1$
  - Push out excess $X(j) \triangleq \sum_{(i,j) \in E} f(i,j) - \sum_{(j,k) \in E} f(j,k)$
  - Keeping $P2(\epsilon)$: push only on admissible edge with $w^p(i,j) < 0$
  - If nowhere to push, increase self price: $p(i) = p(i) + \epsilon/2$

- For $P2(\epsilon)$
  - Remove residue edge by filling its capacity: $f(i,j) = c(i,j)$
  - $P0$ and $P1$ are kept

- For $\epsilon < 1/|V|$
  - Half $\epsilon$ when $P1$ and $P2(\epsilon)$
Goldberg’s algorithm

\[ f, p, \epsilon := 0, 0, \max_{(i,j) \in E} |w(i,j)| \]
do  /* P0 */
  \[ \exists (i,j) \in E(f) : X(i) > 0 \land -\epsilon \leq w^p(i,j) < 0 \rightarrow \text{push}(i,j) \]
  \[ \exists i \in V : X(i) > 0 \land \forall (i,j) \in E(f) : w^p(i,j) \geq 0 \rightarrow p(i) := p(i) + \epsilon / 2 \]
  \[ \exists (i,j) \in E(f) : w^p(i,j) < -\epsilon \rightarrow f(i,j) := f(i,j) + c_f(i,j) \]
P1 \land P2(\epsilon) \land \epsilon \geq 1/|V| \rightarrow \epsilon := \epsilon / 2
od  /* P0 \land P1 \land P2(\epsilon) \land P3 */
Good Features

- Correctness by construction
  - Post-condition is true when algorithm ends.
- Termination
  - No node distance decreases more than $3|V|$ times for one $\epsilon$.
- Parallelism exposed
  - $2|E| + |V| + 1$ guarded commands, many of which are independent.
General Principle

- **General ideas**
  - Bind one thread to each core.
  - Thread has same life span as program.
  - Each thread can execute every guarded command.
  - Executed command depends on available data on a core.
  - Data (or their tokens) move among cores.

- **Advantages**
  - Long live threads to avoid overhead on creating/destroying threads
  - Thread bound to core to avoid preemption
  - Cores are keeping busy
Multicore Min-Cost Flow Program

- Same program for each core

\[
\text{while } \epsilon > 1/|V| \\
\quad \text{if get some active nodes } V_a \\
\quad \quad \text{for } i \in V_a \\
\quad \quad \quad \text{for } (i,j) \in E(f) \\
\quad \quad \quad \quad \{ \text{if } (w^p(i,j) < -\epsilon) f(i,j) := f(i,j) + c_f(i,j) \}
\quad \quad \quad \text{elseif } (w^p(i,j) < 0) \text{ push}(i,j) \}
\quad \quad \text{end for}
\quad \quad \text{if } (X(i) > 0) \{ \text{relabel}(i) \}
\quad \text{end for}
\quad \text{elseif Sync on idle}
\quad \quad \epsilon := \epsilon / 2
\quad \quad \text{activate } V
\quad \text{end while}
\]
Scheduling for Each Thread

- Iteratively fetch active nodes from a global queue $Q$
- Check active nodes for enabled commands
- Execute each enabled command atomically
- Put new active node into the global queue
Atomicity Enforcement

- Atomic semantics of commands
  - Transactional memory: natural but immature
  - Mutual exclusion by atomic *Compare-And-Swap*

```cpp
if (node->token.compare_and_swap(BUSY, IDLE) == IDLE)
  -> Execute the command;
```
How to Detect Termination

- No thread can terminate if there is one busy
- Take a global snapshot: Termination Detection Barrier
  - TDBARRIER holds a counter implemented by atomic integer
  - Counter initialized to zero
  - Once a thread idle/active, it decrements/increments counter
  - Counter being zero means global condition achieved
Load Balancing

- Dynamically adjust length of local $b_k = \frac{q_{in}}{q_{out}}$

  \[
  \begin{align*}
  &\text{if } n_{active} \leq n_{total} \times 0.75 \\
  &\quad b_k = b_k / 2 \\
  &\text{else if } n_{active} + \frac{L}{b_k} \geq n_{total} \\
  &\quad b_k = b_k \times 2
  \end{align*}
  \]
Performance Improvement

- Speed-up not optimal for voltage island assignment

Caught by huge connectivity of ground node.
Performance Improvement

- Convert ground node to a ground network

Single Ground

Ground Network

〇: PI  〇: PO  〇: ground node

Ground Network

avg. #edges = 4
Experiment Setup

- Implemented in multithreaded (TBB) C++.
- Compiled once and runs for different number of cores
- Application: voltage island assignment [Ma and Young ICCAD08]

\[ \begin{align*}
\text{Min} & \quad \sum_{(i,j) \in E} \text{power}_{ij}(v(i,j)) \\
\text{s.t.} & \quad \forall (i,j) \in E : p(i) + d_{ij}(v(i,j)) \leq p(j) \\
& \quad \forall i \in V : 0 \leq p(i) \leq \phi \\
& \quad \forall (i,j) \in E : v(i,j) \in \text{Voltage}
\end{align*} \]

- Linux server with two dual-core 3.0GHz CPUs and 2GB RAM, up to 4 cores.
## Effectiveness of Ground Network

<table>
<thead>
<tr>
<th>Cases</th>
<th>Single Ground</th>
<th>Ground Network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#Contentions</td>
</tr>
<tr>
<td>n10</td>
<td>0.00</td>
<td>1.25</td>
</tr>
<tr>
<td>n30</td>
<td>58.50</td>
<td>1.03</td>
</tr>
<tr>
<td>n50</td>
<td>196.75</td>
<td>1.25</td>
</tr>
<tr>
<td>n100</td>
<td>908.75</td>
<td>1.31</td>
</tr>
<tr>
<td>n200</td>
<td>6111.00</td>
<td>1.07</td>
</tr>
<tr>
<td>n300</td>
<td>8809.00</td>
<td>1.02</td>
</tr>
</tbody>
</table>
## Speedup Rates on Voltage Island Assignment

| Cases | $|V|/|E|$            | Speedup Rate of 2C | Speedup Rate of 4C |
|-------|------------------|--------------------|--------------------|
|       |                  | AVG | MIN | MAX | AVG | MIN | MAX |
| n200  | 1344/2329        | 1.61| 1.40| 1.81| 2.26| 1.99| 2.96|
| n300  | 2209/3834        | 1.44| 1.17| 1.84| 1.90| 1.31| 2.44|
| n600  | 4414/7662        | 1.46| 1.26| 1.60| 2.24| 1.87| 2.64|
| n800  | 5376/9322        | 1.73| 1.52| 1.99| 2.78| 2.32| 3.31|
| n900  | 6619/11490       | 1.44| 1.15| 1.97| 2.15| 1.65| 2.51|
| n1000 | 6720/11653       | 1.76| 1.51| 2.02| 2.92| 2.36| 3.30|
| n1200 | 8824/15318       | 1.53| 1.27| 1.95| 2.54| 2.17| 3.41|
| n1400 | 9410/16319       | 1.83| 1.67| 2.03| 3.16| 2.86| 3.44|
| n1600 | 10752/18646      | 1.57| 1.47| 1.69| 2.72| 2.30| 3.05|
| AVG   | -                | 1.59| 1.38| 1.88| 2.52| 2.09| 3.01|
Conclusions

- Parallel CAD unavoidable under multicore revolution
- Parallelism better explored in Nondeterministic Transactional algorithms
- A systematic multicore implementation based nondeterministic transactional algorithm
- Min-cost flow solver with application on voltage assignment demonstrates effectiveness
- Extending to other CAD applications
Thank you!