A Parallel Implementation of the Push-Relabel Max-Flow Algorithm with Heuristics

6.884 Final Project, Spring 2010
Victoria Popic, Javier Velez
Background

- **Applications**
  - resource allocation, scheduling, linear programming problems, graph problems (max bipartite matching)

- **Algorithms**
  - augmenting paths (Ford and Fulkerson, Edmonds-Karp, Dinitz)
  - preflow-push (Goldberg and Tarjan) – best in practice Goldberg's push-relabel *hipr* algorithm
Max-Flow Push-Relabel Algorithm

- \( G = (V, E), s, t; c(u, v); f(u, v); |f| \)
- \textit{preflow}: allow excess flow at a vertex
- assign a distance from sink value to each vertex; \( d(s) = |V|, d(t) = 0 \)

\begin{align*}
PUSH(u, v) & \quad \text{// Applicability: } u \text{ is active, } \delta = \min(e(u), c_f(u, v)) \quad f(u, v) \leftarrow f(u, v) + \delta \quad f(v, u) \leftarrow f(v, u) - \delta \quad excess(u) = excess(u) - \delta \quad excess(v) = excess(v) + \delta \\
RELABEL(u) & \quad \text{// Applicability: } u \text{ is active, } \\
& \quad \forall w \in V, if c_f(u, w) > 0 \Rightarrow d(u) \leq d(w) \quad d(u) \leftarrow \min(d(w) | (u, w) \in E_f) + 1 \\
DISCHARGE(u) & \quad \text{// Applicability: } u \text{ is active} \quad \text{while } excess(u) \neq 0 \quad \text{PUSH or RELABEL } (u) \\
\end{align*}

- ordering for discharge: FIFO / LIFO; highest distance nodes first (best)
RMF Graphs Parametrized by \((a,b)\)
Trees Parametrized by \((L, d, m)\)
HI-PR (Goldberg) Data Structures

bucket

* active
* inactive

node

d
excess
* prevNode
* nextNode

bucket

* active
* inactive

active nodes, d = 1

buckets
Global Relabeling Heuristic

- **backwards BFS** from sink: computes exact distances of nodes from the sink
- updates buckets and node data (distance and current arc)

for each (node $i$: inactive and active list of bucket $k$)
  for all neighbors $j$ s.t. $(j, i)$ is an **admissible** arc
    update $j$: $j.d = k+1$, $j.current = j.first$
    if($j.excess > 0$)
      add $j$ to $(k+1)$ bucket’s active list
    else
      add $j$ to $(k+1)$ bucket’s inactive list
Global Heuristic Time

Global Update Heuristic Time
Frame Graph, \( a=40 \)

- Blue line: Total Time
- Red line: Global Update Time

Time (sec)

\( b \)
Parallel Global Relabeling Heuristic with Pennants and Bags

- use **Bag reducers** to store the nodes in the buckets during search (4 Bag reducers for 2 levels of active and inactive lists)
- after we’re done computing layer $k$, *set the pointers of bucket $k$* to the nodes in the active and inactive reduced bag
- we need to maintain a **node chain inside** our bags
  - modify bag’s **INSERT(node)** and **MERGE(bag)** to maintain pointers between all the nodes inside the bag
- race: when checking if a node has been visited already, use atomics/locks to avoid duplicates in the buckets
Parallel Global Relabeling Results

- $rmf$ graph ($a=100$, $b=100$) $|V| = 1,000,000$, $|E| = 4,950,000$

- global update time: serial = 7.848 (s), parallel = 3.932 (s)  
  speedup = 2

Cilkview Results

Parallelism = 36.34

Burdened Parallelism = 14.14

Speedup Estimate

- 2 procs: 1.79 - 2.00
- 4 procs: 2.94 - 4.00
- 8 procs: 4.34 - 8.00
- 16 procs: 5.71 - 16.00
- 32 procs: 6.77 - 32.00
Testing for Memory Bandwidth: extra work

Parallelism = 36.34

Parallelism = 25.74
Testing Memory Bandwidth: Running 8 Independent Copies of Serial Code

- 1 copy serial code alone: 7.848 (s)
- 8 independent copies: accounts for factor of 2 slowdown (i.e. speedup of 2 instead of 4)
Concurrent Global Relabeling Heuristic

- all processors have to be suspended in order to do global relabeling – instead we should run it *concurrently* with push-relabel
- Anderson and Setubal '92 introduced the concept of a *global relabeling wave*
- each vertex stores a wave number – the global-relabeling wave that most recently updated it
- we only push flow between vertices with same wave number; both nodes need to be locked
- no distance relabeling operation should decrease the distance label of a node; node should be locked during relabel and global-relabeling operations
Parallel Push-Relabel

- parallel discharge in approximate highest-label first order:
  - discharge-chain
  - coarsened-discharge
  - local-queues
    [keep a local list of activated nodes]
- lock-free push-relabel
Discharge-Chain

- spawn a discharge-chain: let the processor proceed discharging its newly activated node with the highest distance label – if it exists and if its distance is $\geq$ to the global highest distance of an active node

```c
MAIN_{discharge-chain}()

while ActiveNodeSet \neq \emptyset
    u \leftarrow \max_{v \in ActiveNodeSet} d(v)
    cilk_spawn DISCHARGE-CHAIN(u)
```
Coarsened-Discharge

- gather a batch of active nodes to discharge into an array starting from the highest-label bucket, run a *cilk_for* loop over these nodes
- number of nodes gathered, $T$, can be varied to improve performance

```c
MAIN_{coarsened\text{-}discharge}()

while ActiveNodeSet \neq \emptyset
    // Grab the top $T$ active nodes
    a \leftarrow \text{buckets[top $T$ elements]}
    cilk_for u \in a
        \text{DISCHARGE}(u)
```
In-Out Local Thread Queues
(Anderson and Setubal '92; Bader '06)

- each thread has a local input queue of buckets and a local output queue
- threads grab active nodes to discharge from *global* buckets and place newly activated nodes into their local output queue
- when output queue is filled, the nodes in the output queue are transferred back to the global buckets
- Variables (need to be adjusted dynamically):
  - *thr_in* = how many active nodes to grab
  - *thr_out* = size of the output queue / when to sync with the global buckets

*current implementation needs to be optimized*
In-Out Local Thread Queues

- **Global buckets**
- **Local buckets**
- **thr_in**
- **thr_out**
- **current_worked_id**
- **Input queue**
- **Output queue**

**ACTIVE NODES**
## Parallel Push-Relabel Results

<table>
<thead>
<tr>
<th>algorithm</th>
<th>sequential</th>
<th>parallel</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>discharge-chain</td>
<td>126.63</td>
<td>108.98</td>
<td>1.16</td>
</tr>
<tr>
<td>discharge-chain-concurrent</td>
<td>131.8</td>
<td>54.07</td>
<td>2.44</td>
</tr>
<tr>
<td>coarsened-discharge</td>
<td>85.83</td>
<td>116.79</td>
<td>1.16</td>
</tr>
<tr>
<td>local-queues</td>
<td>176.85</td>
<td>166.31</td>
<td>1.064</td>
</tr>
<tr>
<td>discharge-chain</td>
<td>94.44</td>
<td>86.11</td>
<td>1.1</td>
</tr>
<tr>
<td>discharge-chain-concurrent</td>
<td>116.35</td>
<td>65.57</td>
<td>1.77</td>
</tr>
<tr>
<td>coarsened-discharge</td>
<td>102.24</td>
<td>133.3</td>
<td>0.77</td>
</tr>
<tr>
<td>local-queues</td>
<td>186.8</td>
<td>202.51</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Running times (in seconds) of the parallel push-relabel algorithms.
- Parallel times were obtained on 8 workers.
- `rmfl` graph, a=50 and b = 1000, has 2,500,000 nodes and 12,297,500 edges;
  - `rmfw` graph, a=200 and b=50, has 2,000,000 nodes and 9,920,000 edges.
- The `hipr` algorithm runs in **88.77 s** on `rmfl` and **126.66 s** on `rmfw`
Discharge-Chain Results

Cilkview plot: speedup for parallel push-relabel using discharge-chain on rmf(a = 50, b = 1000) **without** concurrent global-relabeling
Best: Discharge-Chain with Concurrent Global-Relabeling

Parallel push-relabel using discharge-chain with concurrent global-relabeling: speedup of ~2 on rmfl graphs
Coarsened-Discharge Results

Cilkview plot: speedup for parallel push-relabel using coarsened-discharge on rmf(a = 50, b = 1000) **without** concurrent global-relabeling
Lock-Free Push-Relabel (Hong'08)

- Push only to the 'lowest' neighbor
- Lift yourself if no lower neighbor
- Done completely in parallel (per node!)
- Except .... Termination is a problem
  - Must figure out when no node has any excess
  - This now requires a barrier (aka a Lock!)
- Oh, and tons of Compare-And-Swap ops.
Lock-Free: Exactly How Bad?

- Original Push-Relabel: \( O(N^2 E) \)
- “Lock Free” (without termination): \( O(N^2 E) \)
- Highest Active Nodes First (hi_pr): \( O(N^2 E^{1/2}) \)
- Tarjan Dynamic Trees: \( O(N^2 \log(N^2 / E)) \)
- \( E^{1/2} \) Slower, but potentially \( N \) Parallelism
Lock-free: Push-Uplift

\( \text{PUSH}_{\text{lock-free}}(u,v) \)

// Applicability: \( \text{excess}(u) > 0 \)
\( \delta = \min(\text{excess}(u), c_f(u,v)) \)
\text{FETCH-AND-SUBTRACT}_{\text{atomic}}(f(u,v) - \delta)
\text{FETCH-AND-ADD}_{\text{atomic}}(f(v,u) + \delta)
\( e_u = \text{SUBTRACT-AND-FETCH}_{\text{atomic}}(\text{excess}(u) - \delta) \)
\( e_{\text{old}} = \text{FETCH-AND-ADD}_{\text{atomic}}(\text{excess}(v) + \delta) \)
\text{if } e_{\text{old}} = 0 \& \delta > 0 \& u \notin \{\text{source}, \text{sink}\}
   // v gained positive excess and became active
   \text{FETCH-AND-ADD}_{\text{atomic}}(\text{active-node-count} + 1)
   \text{LOCAL-ADD-TO-STRATA-OUTSET}(v)
\text{if } e_u = 0 \& \delta > 0
   // u just became inactive
   \text{FETCH-AND-SUBTRACT}_{\text{atomic}}(\text{active-node-count} - 1)

\( \text{UPLIFT}_{\text{lock-free}}(u) \)

// Applicability: \( \text{excess}(u) > 0 \& \forall (u,v) | c_f(u,v) > 0 \),
// \( d(u) \geq d(v) \)
// First, find min distance of admissible arc neighbors
\( h \leftarrow \min\{ d(v) | (u,v) \in G \& c_f(u,v) > 0 \} \)
\text{if } \{v | (u,v) \in G \& c_f(u,v) > 0\} = \emptyset
  \text{return false}
\text{else}
  d(u) \leftarrow h + 1
  \text{return true}
Lock-Free: Order Heuristic – STRATA Data Structure

**STRATA:**

// number of layers in array
long num-layers
// the distance of lowest layer no including bottom
long lowest-layer
// max distance of any node in outset
long max-distance
// min distance of any node in outset
long min-distance
std::vector<node*> top
std::vector<node*>* layers
std::vector<node*> bottom
// num elements in layers not including outset
long num-elements
// nodes which are locally in this strata
// but are not in layers yet
std::vector<node*> outset
// num layers which will be operated in parallel
// not including top
long num-active-layers
// bookkeeping variable with the last
// noderange mapping to this strata
long node-range-last
Lock-Free Results

Work Comparison Lockfree HI–PR Algorithms

- HI–PR
- HI–PR_{parBFS}
- Lockfree
- HI–PR_{no heuristics}

number of PUSH operations

number of nodes