CSE638 & AMS641: Advanced Algorithms **Lecturer:** Prof. Rezaul A. Chowdhury
Submitted by: Yoni Fogel, Arun Rathakrishnan and Amitav Paul **Date:** March 12, 2013

Homework # 1

1 Parallel BFS with Work Stealing

1.a

See Graph::serial _ bfs() in bfs.cc

1.b

- In *PARALLEL-BFS*, the following race conditions may occur, but will not affect the correctness of the output: - In *PARALLEL-BFS-THREAD*, multiple cores can be on line 5 with the same vertex v. It is then possible for two (or more) threads to call line 6 at the same time for v. This does not affect correctness because all threads are writing the same value. - In *PARALLEL-BFS-THREAD*, when stealing, we lock the thief and victim, but only protecting from other thieves. There can be a race condition if the victim is fast (or MIN-STEAL-SIZE $\approx |Q^{in}.q[\text{victim}]|$, where the victim explores some vertexes that were ALSO stolen. This does not affect correctness, it only slightly impacts performance.

1.c

- One vertex can be the end point of many vertexes. So, one vertex can be multiple times in Q^{in}. In case of stealing, due to overlapping of execution thief and victim may process the same vertex. So that same vertex can exist multiple times in Q^{in}.

- So, one vertex can not be more than one time in any $Q^{in}.q[i]$ because the check on line 5 of bfs-thread (and a single queue is serialized). It may end up in different queues (see race conditions above).

- We can add an additional field *c[v]* for each vertex *v* which will keep the track of the processor by which it discovered. i.e. $\mathbf{c}[\mathbf{v}] = \mathbf{i}$ in between line 6-7 in PARALLEL-BFS-THREAD.

And before expanding a vertex we check whether the vertex is discovered by this thread or not by checking *c[v]*. i.e. we can add ***if*** $(\mathbf{c}[\mathbf{v}] = \mathbf{i})$ before ***for*** loop at line 4 in PARALLEL-BFS-THREAD. So, only the processor who have won in the race condition for vertex v has right to expand it.
Note this does not affect correctness because exactly one processor will win the race (if it happens).

- A single vertex cannot be in queues from multiple levels because we store the length of the shortest path to each node. If we later find a longer path we do not overwrite it.

1.d

We consider phases of p steals each.
Let $F_i = \left\lceil \frac{|q_i|}{MIN-STEAL-SIZE} \right\rceil$

The potential of $Q_i = W_i = {F_i}^2$
Let

$$S_i = \begin{cases} 1, \text{ No steal attempts on this processor during this phase} \\ 0, \text{ otherwise} \end{cases}$$

When at least one steal attempt happens on a victim processor, the potential of the whole system drops by at least $W_i - (\frac{W_i}{4}) - (\frac{W_i}{4}) = \frac{W_i}{2}$ (Trivially for empty queues).

$X_i = W_i S_i$ is the portion of the potential that was involved in attempted steals. The potential drop of the whole system $\geq X_i/2$.

$E[S_i] = 1 - (1 - \frac{1}{p})^p \geq (1 - \frac{1}{e}) \geq \frac{1}{2}$

Now,

$$E\left[X\right] = \sum_{i=1}^{p} E\left[X_i\right] > \left(1 - \frac{1}{e}\right) \sum_{i=1}^{p} E[W_i] > \left(1 - \frac{1}{e}\right) E\left[W\right]$$

By Markov's inequality,

$$Pr\left(X < \beta[W]\right) = Pr\left(W - X > (1-\beta)[W]\right) < \frac{E\left[[W] - X\right]}{(1-\beta)\, E[W]} < \frac{1}{(1-\beta)\, e}$$

Setting $\beta = \frac{1}{2}$ in Lemma **??**, we get,

$$Pr\left(X < \frac{1}{2}\Phi_i\right) < \frac{2}{e} \implies Pr\left(X > \frac{1}{2}\Phi_i\right) \geq \left(1 - \frac{2}{e}\right) = \frac{1}{4}$$

Consider time steps, i and j such that $j > i$ and at least p steal attempts occur between time steps i (inclusive) and j (exclusive) then,

$$Pr\left(\Phi_i - \Phi_j > \frac{\Phi_i}{4}\right) > \frac{1}{4}.$$

Similar to Quicksort, Cilk Stealing, and other randomized algorithms, when a phase reduces a total by at least a constant fraction by at least a constant probability, takes $O(\log x)$ times the expected number of phases w.h.p in x to reduce to any constant.

1.e

Assuming there is at least one core to steal from, the probability that a core tries to steal $cp \log p$ times and never chooses any of those cores $\leq (1 - \frac{1}{p})^{cp \log p} = ((1 - \frac{1}{p})^p)^{clnp} = (\frac{1}{e})^{clnp} = \frac{1}{p^c}$

So MAX-STEAL-ATTEMPTS = $cp \ln p, c > 1$ means that if any cores have enough work to be stolen from, any core trying to steal will choose at least one of them with high probability in p.

Moreover, in reference of *Task 1(d)*, it can be shown that $cp^2 \ln p, c > 1$ (p cores have to steal before finishing bfs-thread) is more than the expected number of steal attempts before all steal attempts in the system fail.

1.f

MIN-STEAL-SIZE is assumed to be $\Theta(1)$.

T_p:

Waiting to launch: $O(Dp)$.

Stealing: $O(Dp \log p + D \log p \log n)$ With high probability in p.

Exploring: $O(D\Delta + \frac{n}{p} + \frac{m}{p})$ worst case.

Synchronization: $O(D \log p)$ worst case.

So, $T_p = O(D(p \log p + \log p \log n + \Delta) + \frac{n+m}{p})$ with high probability in p.

Total work:

Stealing: $O(Dp^2 \log p + Dp \log p \log n)$ with high probability in p.

Exploring: $O(n + m)$ worst case.

Synchronization: $O(Dp)$

So, total work is $O(D(p^2 \log p + p \log p \log n) + n + m)$

Lower Bound for size of the input graph where total work is work optimal ($O(n + m)$):

$m + n = \Omega(D(p^2 \log p + p \log p \log n))$.

1.g

Operate as normal. When you try to steal from a victim with fewer than MIN-STEAL-SIZE vertexes, switch to stealing half the out degrees remaining. Instead of $Dp \log p \log q_l$ steals with high probability, it's $Dp \log p \log(q_l \Delta)$ steals.

Assuming MIN-STEAL-SIZE $= \Theta(1)$ means it takes constant time to split the queue into two pieces, with additional information to say which outgoing edges should be done by the victim and which by the thief.

Without the assumption you can do a serial prefix-sum of the remaining nodes the first time you try to steal edges (instead of nodes). Then you can do MIN-STEAL-SIZE work for free (charge against one of the successful steals) Then you can fail to steal in $O(1)$ time and successfully steal in O(MIN-STEAL-SIZE) time.

The analysis from f remains the same, except that Δ disappears entirely.

1.h

We can replace ***for*** by ***parallel for*** in line 10 of *figure 2* and do all p spawns in parallel. This changes the time spent waiting to start from $O(Dp)$ to $O(D \log p)$ but both are dominated by other terms so it does not change the upper bound of either f or g..

1.i

See Graph::parallel_bfs() in bfs.cc You can do all 2^3 versions of the algorithm by independently turning on the optimizations from c, g, and h.

1.j

Input File	RT_{SBFS}	$RT_{PBFS^{(f)}}$	$RT_{PBFS^{(g)}}$	$SF_{PBFS^{(f)}}$	$SF_{PBFS^{(g)}}$
cage15	1	2	3	4	5
cage14	1	2	3	4	5
freescale	1	2	3	4	5
Wikipedia	1	2	3	4	5
kkt-power	1	2	3	4	5
RMAT100M	1	2	3	4	5
RMAT1B	1	2	3	4	5

Table 1: RT = Running Time, SBFS = Serial BFS, $PBFS^{(f)}$ = Parallel BFS for task 1(f), $PBFS^{(g)}$ = Parallel BFS for task 1(g) and SF = Speed Factor

1.k

Answer:

2 Lockfree Parallel BFS

2.a

Answer:

2.b

Answer: