Parallel Breadth First Search

Yonatan Fogel
Computer Science
Stony Brook University
Stony Brook, New York, 11744

***Abstract*—We have developed a multithreaded implementation of breadth-first search (BFS) of arbitrary graphs using the Cilk++ extensions to C++. We analyze our FBFS algorithm under the CILK model, where synchronizing n tasks takes $\mathcal{O}(\log n)$ time. We introduce a lower bound for T_p in level-synchronous BFS. FBFS achieves high work-efficiency by applying parallel prefix sum to split up all work at a given distance from a source vertex.**

FBFS algorithm deterministically matches the lower bound for T_p, and is asymptotically optimal for T_1. PBFS saves energy by letting processors become idle whenever possible: total non-idle work among all p processors is $\mathcal{O}(T_1)$. Our algorithm is non-deterministic in that it contains benign races which may affect performance (by up to a constant factor) but not its correctness. These races can be fixed with mutual-exclusion locks which will slow down FBFS empirically.

In particular, we show that for a graph $G = (V, E)$ with diameter D, FBFS runs in time $\mathcal{O}((|V| + |E|)/p + D \log p)$ on p processors, which means that it attains near-perfect linear speedup if $p \ll (|V| + |E|)/(D \log(|V| + |E|))$

I. Introduction

A. Subsection Heading Here

Subsection text here.

1) Subsubsection Heading Here: Subsubsection text here.

II. Related Work

A. Terms

- $G = (V, E)$ is the graph.
- V is the set of nodes.
- $n = |V|$.
- $m = |E|$.
- $\Gamma(u)$ is the set of nodes adjacent to node u.
- T_s is the running time for the most efficient serial algorithm.
- $T_1 \equiv$ WORK is the running time for a parallel algorithm running on one processing elements.
- T_p is the running time for a parallel algorithm running on p processing elements.
- $T_\infty \equiv$ SPAN is the critical path for a parallel algorithm or the time it takes given infinite processing elements.
- $W_p \leq pT_p$ is the total amount of work for p processing elements. Idle processing elements do not count towards W_p.

We consider only ***level-synchronous*** BFS algorithms, which find all nodes at distance $0 \leq d \leq |V|$ before any nodes at distance $d' > d$.

The standard serial approach for level-synchronous BFS can be seen in Fig. 1.

The bottlenecks for parallelization are the FIFO queue and the DIST array. The FIFO is fast but is inherently serial. When running in parallel, there is a benign race condition on lines 10–13 of Fig. 1. Multiple threads can enqueue the same vertex. The race is benign and rare: it can create extra work but does not affect correctness. This race is sometimes dealt with mutual exclusion, atomic instructions, or by simply performing the extra work when the race occurs.

Existing algorithms

MIT-Bag [1] uses reducer hyperobjects and a new ***bag*** data structure to replace the FIFO. This algorithm relies on a work-stealing scheduler and runs in $T_p = \mathcal{O}\left(n/p + m/p + D \lg^3(n/D)\right)$ w.h.p in n. MIT-Bag uses mutual exclusion to deal with the race conditions on DIST for analysis, but in practice recommends simply doing the extra work.

CRAY-BFS [2] uses special hardware available on the Cray MTA-2 platform. It writes to the FIFO in parallel by using atomic increments, and uses special hardware mutexes for every 64-bit word.

Block-BFS [3] uses a block-accessed queue to replace the FIFO. It uses atomic primitives to allocate blocks from the FIFO that are small, but not so small so that they do not use atomics too often.

Distinguished-BFS [4] is a non level-synchronous BFS algorithm for arbitrary sparse graphs. Distinguished-BFS runs in $T_p = \mathcal{O}(n^\epsilon)$ time with $\mathcal{O}\left(mn^{1-2\epsilon}\right)$ processors, if $m \geq n^{2-3\epsilon}$. The basic idea is to contract the graph to distinguished and then superdistinguished vertexes, at which point the graph will be dense.

III. BFS(RENAME)

BFS algo

```
parallel for u gets 0 .. n - 1

parallel for i = 0 to p-1
if i = 0
var{offset} = 0
else
var{offset} = D[\frac{iN}{p}
```

`var{offset}`

IV. Parallel Prefix Sum

Parallel prefix sum is a common primitive used in many parallel algorithms. In the CILK model, the standard ($\mathcal{O}(\log n)$ parallel fors) parallel prefix sum on n elements runs in $T_\infty = \mathcal{O}\left(\log^2 n\right)$ or $T_p = \mathcal{O}\left(n/p + \log^2 p\right)$. In the PRAM model it runs in $T_\infty = \mathcal{O}(\log n)$.

We introduce a divide-and-conquer version of parallel prefix sum that runs in $T_p = \mathcal{O}(n/p + \log p)$.

V. Analysis

VI. Conclusion

VII. Future Work

TODO "WRITE" these instead of what we have now.

`Add labels and links for each of these`

- Optimize for false sharing
  - From parallel Prefix Sum ¡- pretty easy, just set appropriate GRAIN-Size and align properly.
  - For Dist array.
    - can use radix sort for large levels where num degrees $= \Omega(n^c)$
    - Each proc can create an unsorted list of all cache lines its degrees touch, then each proc takes $\#cachelines/p$ cache lines
  - For find sublist
- Optimize for cache-misses
- Make processor-oblivious
- Make cache-oblivious? Is it already?
- Optimize T_p, T_1, W_p for PRAM model

References

[1] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 303–314. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810534

[2] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *Proceedings of the 2006 International Conference on Parallel Processing*, ser. ICPP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 523–530. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2006.34

[3] E. Saule and U. V. Catalyurek, "An early evaluation of the scalability of graph algorithms on the intel mic architecture," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, ser. IPDPSW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1629–1639. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2012.204

[4] J. Ullman and M. Yannakakis, "High-probability parallel transitive closure algorithms," in *Proceedings of the second annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '90. New York, NY, USA: ACM, 1990, pp. 200–209. [Online]. Available: http://doi.acm.org/10.1145/97444.97686

```
SERIAL-BFS( V, Γ, s )
for each vertex u ∈ V
    DIST[u] ← ∞
DIST[s] ← 0
ENQUEUE( Q_in, v )
while Q_in ≠ ∅ do
    Q_out ← ∅
    while Q_in ≠ ∅ do
        u ← DEQUEUE( Q_in )
        for each vertex v ∈ Γ(u) do
            if DIST[v] = ∞ then
                DIST[v] ← DIST[u] + 1
                ENQUEUE( Q_out, v )
    Q_in ← Q_out
```

Fig. 1. Standard serial level-synchronous breadth-first search implementation on a graph $G = (V, E)$ with source vertex $s \in V$. $\Gamma(u)$ is the adjacency list for node u.

PARALLEL-BFS(V, Γ, γ, s, p_{max})

$V[0 : n-1]$ are the n nodes in the graph. $\Gamma[u]$ is the sequence of adjacent nodes to node u. $\gamma[u] = |\Gamma[u]|$. s is the source vertex from which distance is calculated. p_{max} is the maximum number of processors to use. Returns DIST$[0 : n-1]$ which represents the distance from s to each vertex.

```
parallel for u ← 0 to n − 1 do
    DIST[u] ← ∞
    OWNER[u] ← ∞
DIST[s] ← 0
OWNER[s] ← 0
if γ[s] = 0 then
    return DIST
INPUT ← array[0 : 0]
INPUT[0] ← s
LEVEL ← 0
p ← 1
while |INPUT| ≠ 0 do
    LEVEL ← LEVEL + 1
    N ← |INPUT|
    WORK ← array[0 : N − 1]
    parallel for u ← 0 to N − 1 do
        WORK[u] ← γ[INPUT[u]]                                   {}
    PARALLEL-PREFIX-SUM( WORK, ⌊N/p⌋ )
    W ← WORK[N − 1]
    p ← MIN( p_max, W )
    SUBLIST ← FIND-SUBLIST( WORK, W, p )
    Q ← LEVEL-TO-QUEUES( INPUT, WORK, SUBLIST, DIST, Γ, γ, W, p, LEVEL )
    SIZES ← array[0 : p − 1]
    parallel for i ← 0 to p − 1 do
        Q-NEW ← queue
        for v in Q[i] do
            if OWNER[v] = i then
                Q-NEW.ENQUEUE( v )
        Q[i] ← Q-NEW
        SIZES[i] ← |Q[i]|
    PARALLEL-PREFIX-SUM( SIZES, 1 )
    INPUT ← array[0 : SIZES[p − 1]]
    parallel for i ← 0 to p − 1 do
        if i = 0 then
            OFFSET ← 0
        else
            OFFSET ← SIZES[i − 1]
        for j ← OFFSET to OFFSET + |Q[i]| do
            INPUT[OFFSET] ← Q[i].DEQUEUE( )
```

Fig. 2.

```
FIND-SUBLIST( WORK, W, p )
    N ← |WORK|
    p_n ← MIN( p, N )
    SUBLIST ← array[0 : p − 1]
    RANGESSTART ← array[0 : p_n − 1]
    RANGESEND ← array[0 : p_n − 1]
    parallel for i ← 0 to p_n − 1 do
        if i = 0 then
            FIRSTDEGREE ← 0
        else
            FIRSTDEGREE ← WORK[ ⌊ iN / p_n ⌋ − 1]
        FIRSTDEGREENEXT ← WORK[ ⌊ (i+1)N / p_n ⌋ − 1]
        RANGESSTART[i] ← ⌈ p·FIRSTDEGREE / W ⌉
        RANGESEND[i] ← ⌈ p·FIRSTDEGREENEXT / W ⌉ − 1
    parallel for i ← 0 to p_n − 1 do
        if RANGESSTART[i] ≤ RANGESEND[i] then
            parallel for j ← RANGESSTART[i] to RANGESEND[i] do     {Use cores RANGESSTART[i] to RANGESEND[i]}
                SUBLIST[j] ← i
    return SUBLIST
```

Fig. 3.

```
LEVEL-TO-QUEUES( INPUT, WORK, SUBLIST, DIST, Γ, γ, W, p, LEVEL )
    N ← |WORK|
    p_n ← MIN( p, N )
    Q ← array[0 : p − 1]
    parallel for i ← 0 to p − 1 do
        Q[i].CLEAR( )
        FIRSTDEGREE ← ⌊ iW / p ⌋
        WORKITEMS ← ⌊ (i+1)W / p ⌋ − FIRSTDEGREE
        VERTEX ← BINARY-SEARCH-FOR-INDEX( WORK, ⌊ N·SUBLIST[i] / p_n ⌋, ⌊ N·(SUBLIST[i]+1) / p_n ⌋, FIRSTDEGREE + 1 )
        DEGREE ← FIRSTDEGREE − WORK[VERTEX − 1]
        while WORKITEMS > 0 do
            u ← INPUT[VERTEX]
            LIMIT ← MIN( WORKITEMS + DEGREE, γ[u] )
            for j ← DEGREE to LIMIT do
                v ← Γ[u][j]
                if DIST[v] = ∞ then
                    DIST[v] ← LEVEL          {Benign race condition. All threads write the same value.}
                    OWNER[v] ← i             {Benign race condition. One thread's value will win.}
                    if γ[v] > 0 then
                        Q[i].ENQUEUE( v )
            WORKITEMS ← WORKITEMS − DEGREE
            DEGREE ← 0
            VERTEX ← VERTEX + 1
    return Q
```

Fig. 4.

```
PARALLEL-PREFIX-SUM( V, GRAIN-SIZE )
V[0 : n − 1] is a sequence of n integers. This function replaces V[i] with Σ_{0≤j≤i} V[j]
    if |V| > 1 then
        PARALLEL-PREFIX-SUM-UP( V, GRAIN-SIZE, 0, n )
        PARALLEL-PREFIX-SUM-DOWN( V, GRAIN-SIZE, 0, n, false, 0 )
```

Fig. 5.

```
PARALLEL-PREFIX-SUM-UP( V, GRAIN-SIZE, start, limit )
    size ← limit − start
    if size ≤ GRAIN-SIZE then
        return SERIAL-PREFIX-SUM( V, start, limit )
    else
        mid ← ⌊ (start+limit) / 2 ⌋
        x ← spawn PARALLEL-PREFIX-SUM-UP( V, GRAIN-SIZE, start, mid )
        y ← PARALLEL-PREFIX-SUM-UP( V, GRAIN-SIZE, mid, limit )
        sync
        V[limit − 1] ← x + y
        return x + y
```

Fig. 6.

```
PARALLEL-PREFIX-SUM-DOWN( V, start, limit, rightmost_excluded, partial_sum )

size ← limit − start
if size ≤ GRAIN-SIZE then
    SERIAL-PREFIX-SUM-DOWN( V, start, limit, partial_sum, rightmost_excluded )
    return
else
    mid ← ⌊(start+limit)/2⌋
    sum_left ← V[mid − 1]
    spawn PARALLEL-PREFIX-SUM-DOWN( V, GRAIN-SIZE, start, mid, false, partial_sum )
    if ¬rightmost_excluded then
        V[limit − 1] ← V[limit − 1] + partial_sum
    if limit − mid > 1
        PARALLEL-PREFIX-SUM-DOWN( V, GRAIN-SIZE, mid, limit, true, partial_sum + sum_left )
```

Fig. 7.

```
SERIAL-PREFIX-SUM( V, start, limit )
V[start] ← V[start] + partial_sum
for i ← start + 1 to limit − 1 do
    V[i] ← V[i] + V[i − 1]
return V[limit − 1]
```

Fig. 8.

```
SERIAL-PREFIX-SUM-DOWN( V, start, limit, rightmost_excluded, partial_sum )

for i ← start to limit − 2 do
    V[start] ← V[start] + partial_sum
if ¬rightmost_excluded then
    V[limit − 1] ← V[limit − 1] + partial_sum
```

Fig. 9.