# Parallel Breadth First Search

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

***Abstract*—This is an $n-1$ level abstract**

## I. Introduction

### A. Subsection Heading Here

Subsection text here.

*1) Subsubsection Heading Here:* Subsubsection text here.

## II. Related Work

### A. Terms

- $T_s$ 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.
- $W_p \le pT_p$ is the total amount of work for $p$ processing elements. Idle processing elements do not count towards $W_p$.

### B. Level-Synchronous BFS Algorithms

One approach for BFS is to find all nodes at distance $0 \le d \le |V|$ from the source before any nodes at distance $d' > d$.

The general approach for Level-Synchronous BFS can be seen in **??**.

[1] uses data structures called penants and bags

*Definition 2.1:* soup

## III. PPS(RENAME)

## IV. 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}
```

## 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

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

Fig. 1. General approach for Level-Synchronous BFS

```
PARALLEL-PREFIX-SUM( V, GRAIN-SIZE )
V[0 : n − 1] is a sequence of n integers. This function replaces V[i] with ∑_{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. 2.

```
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. 3.

```
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. 4.

```
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. 5.

```
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. 6.

```
PARALLEL-BFS( V, Γ, γ, s, p_max )
V[0 : n − 1] are the n nodes in the graph. Γ[u] is the sequence of adjacent nodes to node u. γ[u] = |Γ[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. 7.

```
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. 8.

```
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. 9.