# Parallelizing PageRank Algorithm using OpenMP

# Background and Motivation

- PageRank is a core algorithm in web-scale information retrieval
- Models the Web as a large directed graph
- Characteristics:
  - Iterative computation
  - Memory-bandwidth intensive
  - Highly irregular graph structure (cache miss)

**Project Goal:**

Design parallelized PageRank implementations using OpenMP.

## PageRank: Random Surfer Model

- A user repeatedly:
  - Follows an outgoing hyperlink with probability $d$
  - Jumps to a random page with probability $(1 - d)$
- The PageRank value represents the stationary probability of visiting a page
- Importance is defined recursively

A page is important if it is pointed to by other important pages

## Formula

$$PR^{(t+1)}(u) = \frac{1-d}{N} + \frac{d}{N}\sum_{k \in D} PR^{(t)}(k) + d\sum_{v \in In(u)} \frac{w_{v,u} \cdot PR^{(t)}(v)}{\sum_{z \in Out(v)} w_{v,z}}$$

- $t$: The number of iterations.
- $N$: total number of nodes
- $d$: damping factor (typically 0.85)
- $D$: set of dangling nodes (nodes with no outgoing edges)
- $In(u)$: in-neighbors of node $u$
- $Out(v)$: out-neighbors of node $v$
- $w_{v,u}$: weight of edge $(v \to u)$

# Graph Representation: CSR Format

- Graph stored using **Compressed Sparse Row (CSR)**
- Arrays:
  - `offset[]` : starting index of edges of nodes
  - `struct Edge m[]` : edges - destination node and weight

Advantages:

- Compact memory layout
- Efficient sequential edge traversal
- Well-suited for shared-memory systems

# Push vs Pull Traversal

## Push (Source-Centric)

- Iterate over outgoing edges
- Update destination nodes with new PageRank
- Parallel writes to shared memory (critical)
- Requires atomic operations or locks

## Pull (Destination-Centric)

- Iterate over destination nodes
- Read from incoming neighbors
- Each thread writes to a private location (lock-free)
- I will basically use this strategy

# Strategy 1: Dynamic Scheduling

```
#pragma omp parallel
{
// Pre-calculate normalized values to avoid an irregular memory access
// Also collect the dangling nodes
#pragma omp for schedule(static) reduction(+: dangling_sum)
  for (int i = 0; i < n; ++i) {
    if (out_w[i] != 0) {
      pr_normalized[i] = pr[i] * damping / (double) out_w[i];
    } else {
      dangling_sum += pr[i];
    }
  }

#pragma omp single
  base_score = (1.0 - damping + damping * dangling_sum) / n;

// Scheduling overhead
#pragma omp for schedule(dynamic, 64) reduction(+: diff)
  for (int u = 0; u < n; ++u) {
    double sum = base_score;
    const int start = converse->offset[u], end = converse->offset[u + 1];
    for (int i = start; i < end; ++i) {
      const int v = converse->m[i].v;
      const int w = converse->m[i].w;
      // Here we have one irregular memory access (also bottleneck)
      sum += w * pr_normalized[v];
    }
    pr_new[u] = sum;
    diff += fabs(sum - pr[u]);
  }
}
```

# Strategy 2: Edge-Balanced Static Partitioning

Distribute work based on number of edges

```
// Calculate partition boundaries based on edge counts
int *start_v = malloc(sizeof(int) * (num_threads + 1));
start_v[num_threads] = n;
#pragma omp parallel for schedule(static)
for (int i = 0; i < num_threads; i++) {
  // Find the node index where the cumulative edge count reaches i * (total_edges / threads)
  start_v[i] = lower_bound(converse->offset, 0, n + 1, i * e / num_threads);
}
// ...
// Distribute the nodes
#pragma omp parallel reduction(+: diff)
{
  const int tid = omp_get_thread_num();
  const int end_v = start_v[tid + 1];
  for (int u = start_v[tid]; u < end_v; u++) {
    // ...
  }
}
```

# Advantages and Limitations of Strategy 2

**Advantages:**

- Minimal scheduling overhead than dynamic scheduling
- Theoretically good load balance
- Improved cache locality

**Limitations:**

- Static partitioning cannot adapt at runtime
- Performance sensitive to the CPU Architecture
- Slowest thread determines total time (I will introduce this later.)

## Strategy 3: Approximate PageRank with Active Frontier

**Observation:**

- Many nodes converge early
- Updating them repeatedly wastes memory bandwidth

**Method:**

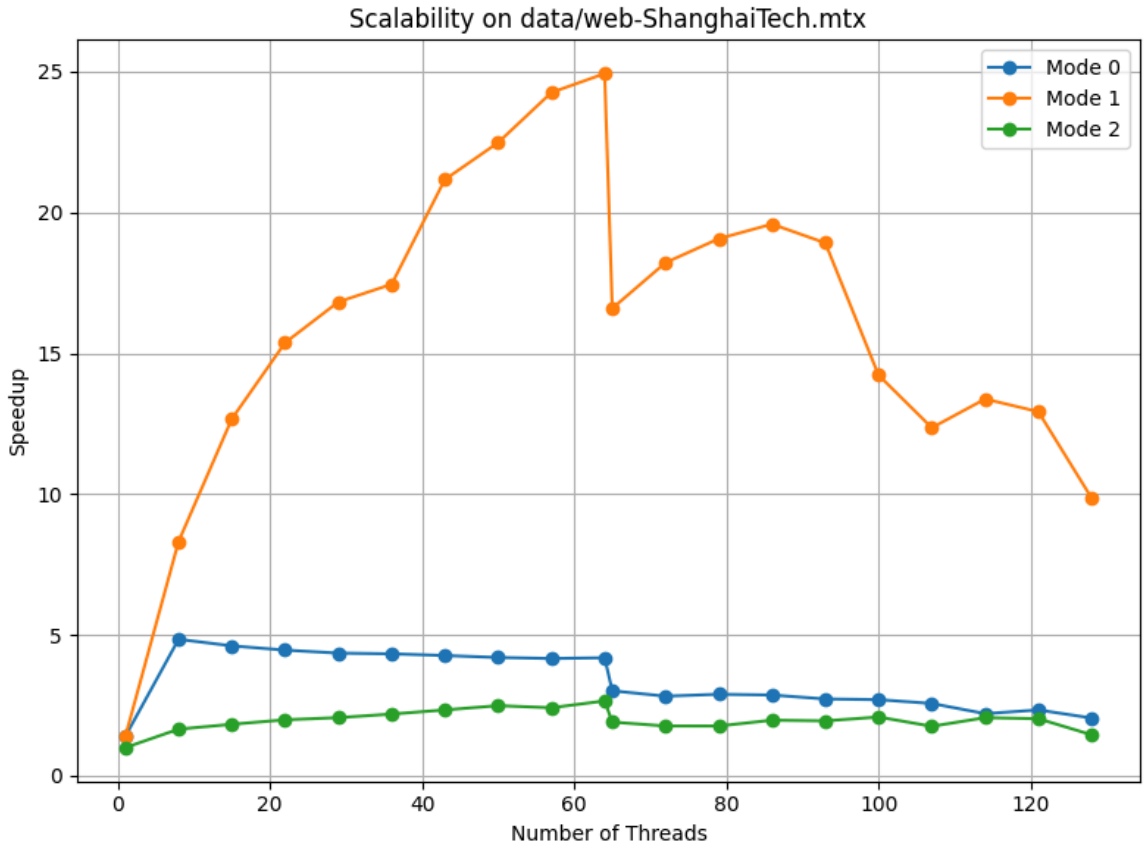
- Maintain a frontier of active nodes
- Only update nodes with significant change

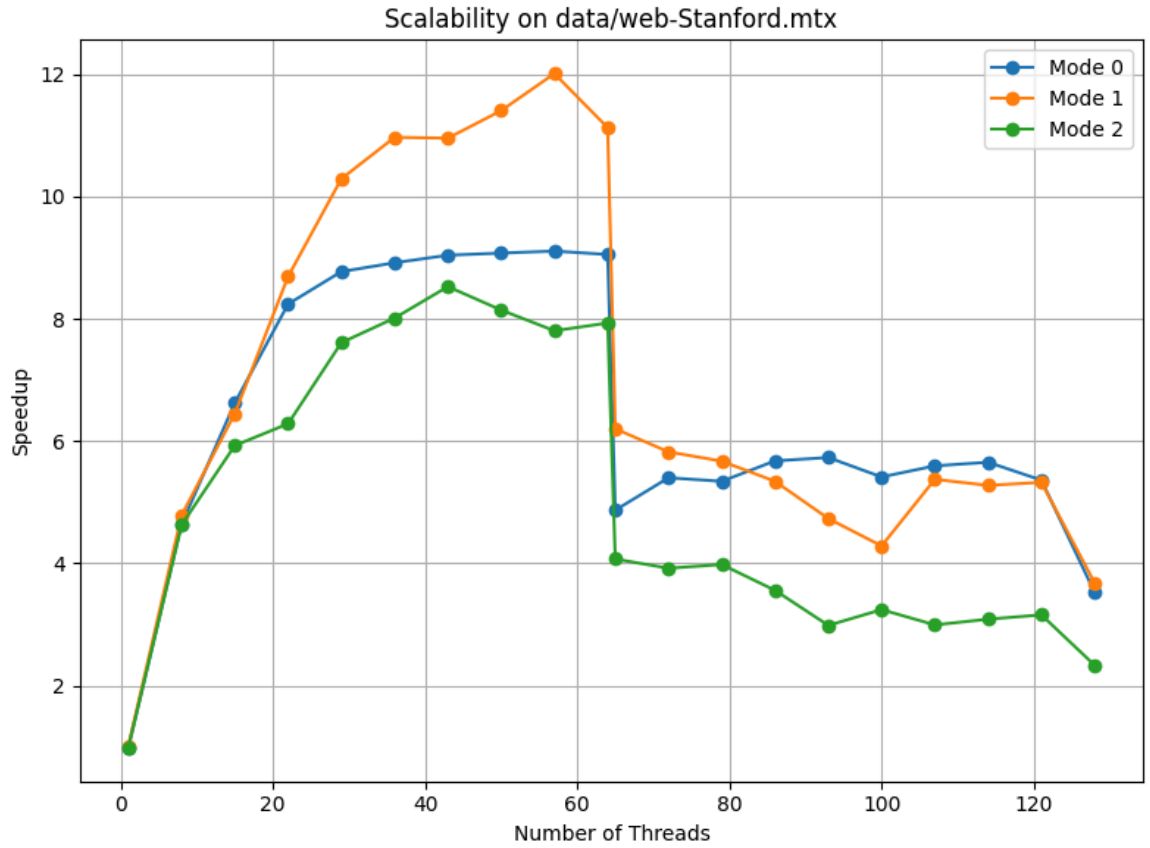
Introduces larger approximation error but somehow preserves ranking quality by choosing a smaller error $\epsilon$

```
// 1. Compute updates only for nodes in current 'frontier'
#pragma omp parallel for schedule(static)
for (int i = 0; i < frontier_size; i++) {
  const int u = frontier[i];
  // ... compute sum ...
  pr_new[u] = sum;
  // Mark if this node needs to be active next time
  next_idx_prefix[i] = fabs(sum - pr[u]) > eps;
}

// 2. Parallel prefix sum to calculate write positions (each thread calculates a block)
omp_prefix_sum(next_idx_prefix, frontier_size, prefix_block_sum);

// 3. Culling
#pragma omp parallel for schedule(static)
for (int i = 0; i < frontier_size; i++) {
  const int prev_val = (i == 0) ? 0 : next_idx_prefix[i - 1];
  if (next_idx_prefix[i] > prev_val) {
    next_frontier[next_idx_prefix[i] - 1] = frontier[i];
  }
}
```

AMD EPYC 7742 64-Core Processor (2250 MHz, 128 Threads, 2 NUMA Nodes)

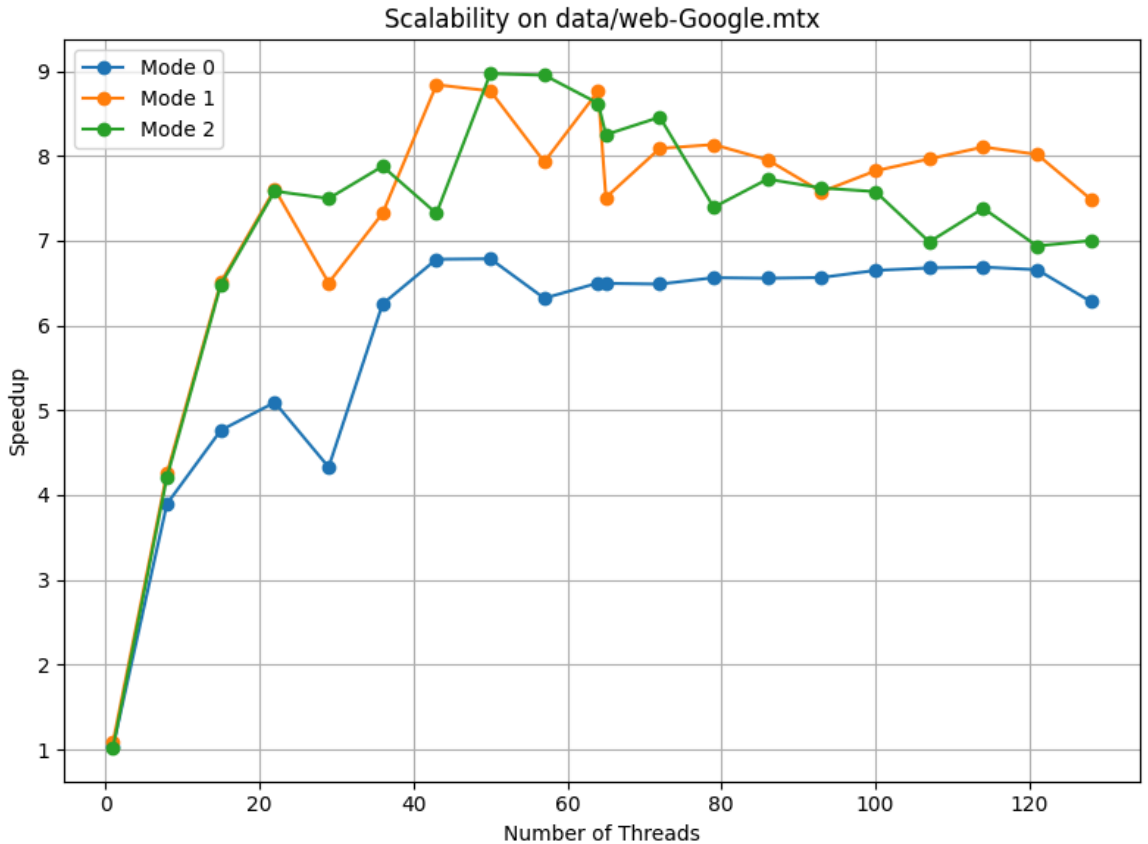
Strategy 1 & 2: $\epsilon = 10^{-6}$; Strategy 3: $\epsilon = 10^{-8}$

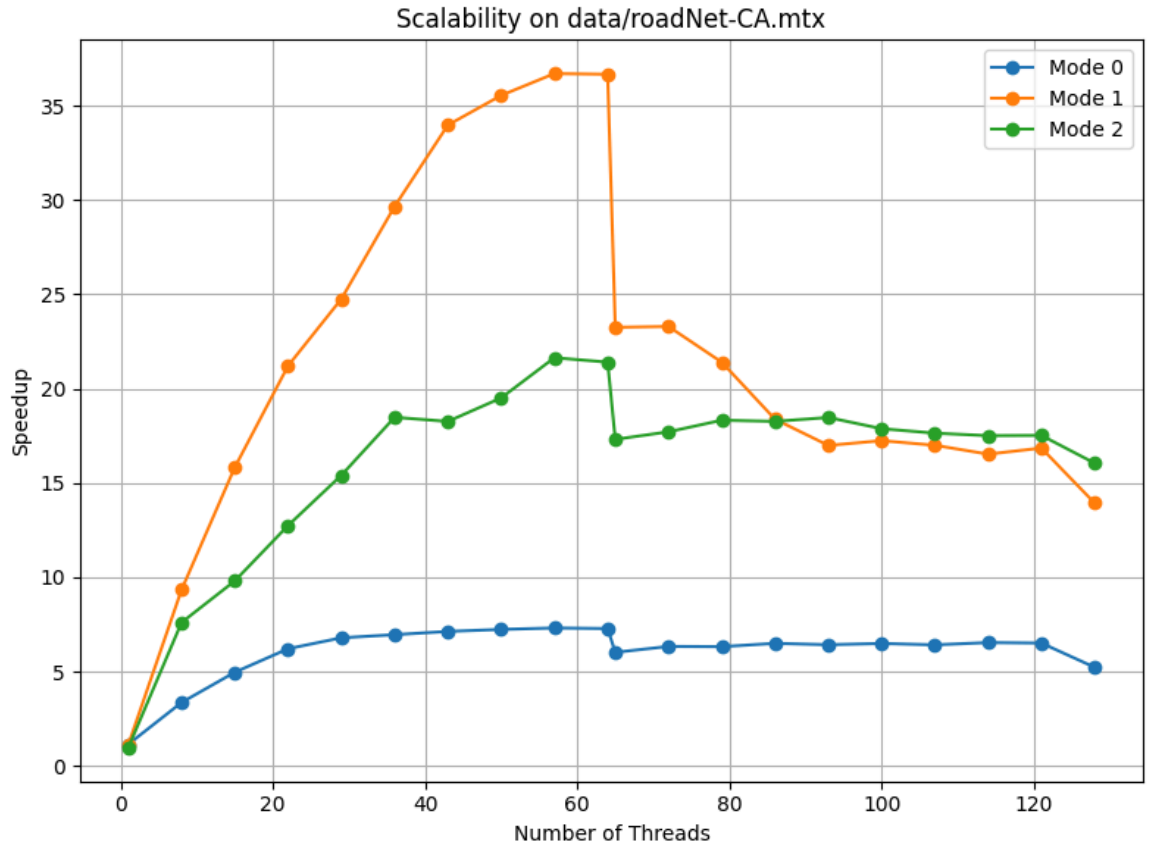


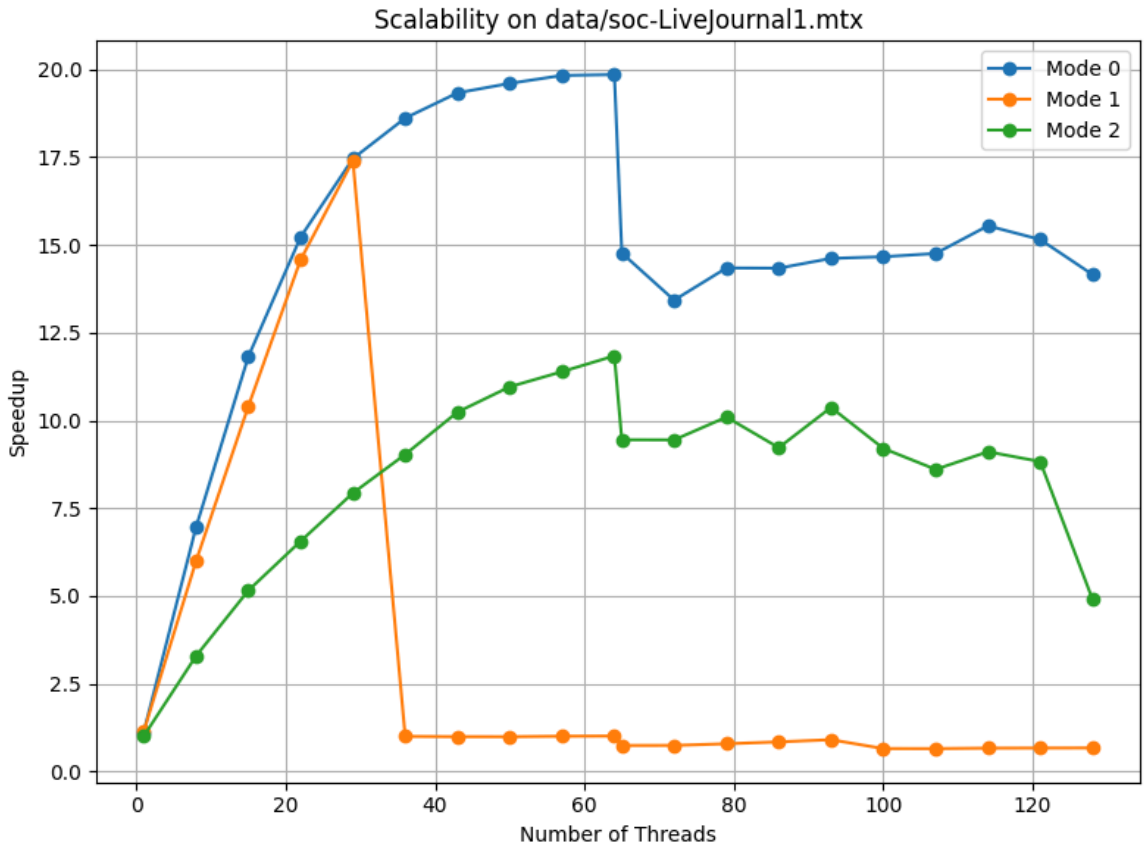



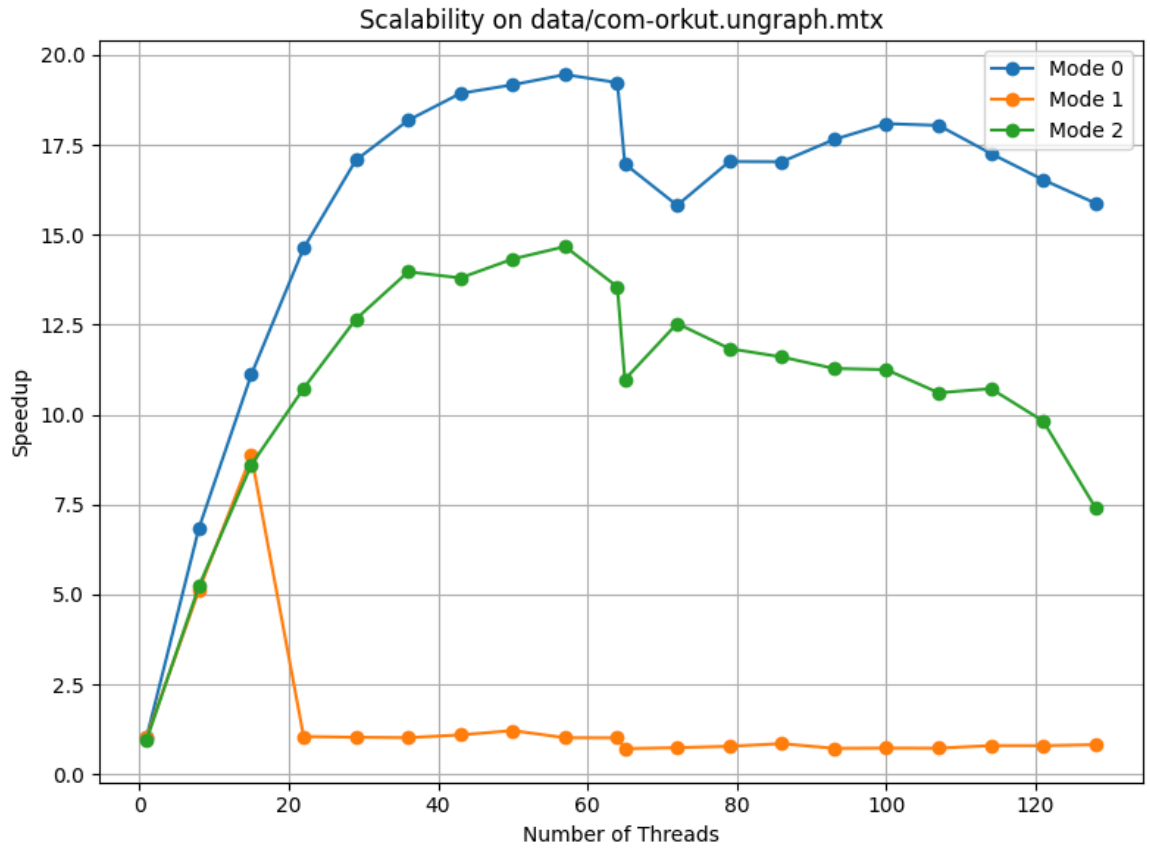

Strategy 2 works well since almost all pages will link to the homepage

(i.e. https://www.shanghaitech.edu.cn/)


Scalability on data/web-Google.mtx
Mode 0
Mode 1
Mode 2
Speedup
Number of Threads
1
2
3
4
5
6
7
8
9
0
20
40
60
80
100
120


Scalability on data/roadNet-CA.mtx
Mode 0
Mode 1
Mode 2
Speedup
Number of Threads
0
5
10
15
20
25
30
35
0
20
40
60
80
100
120




Both are social networks (friends). The neighbors are highly randomized.

Sharply declines at thread number 15~16 and 32~33.

Dynamic scheduling hides memory latency

# Conclusions

- Strategy 2 works better on the graphs of real Webs.
- No single parallel strategy is universally optimal (tradeoffs)
- Strategy selection depends on:
  - Degree distribution
  - Graph size
  - CPU/NUMA topology
- Approximation can significantly reduce runtime with acceptable error

# Thank you!