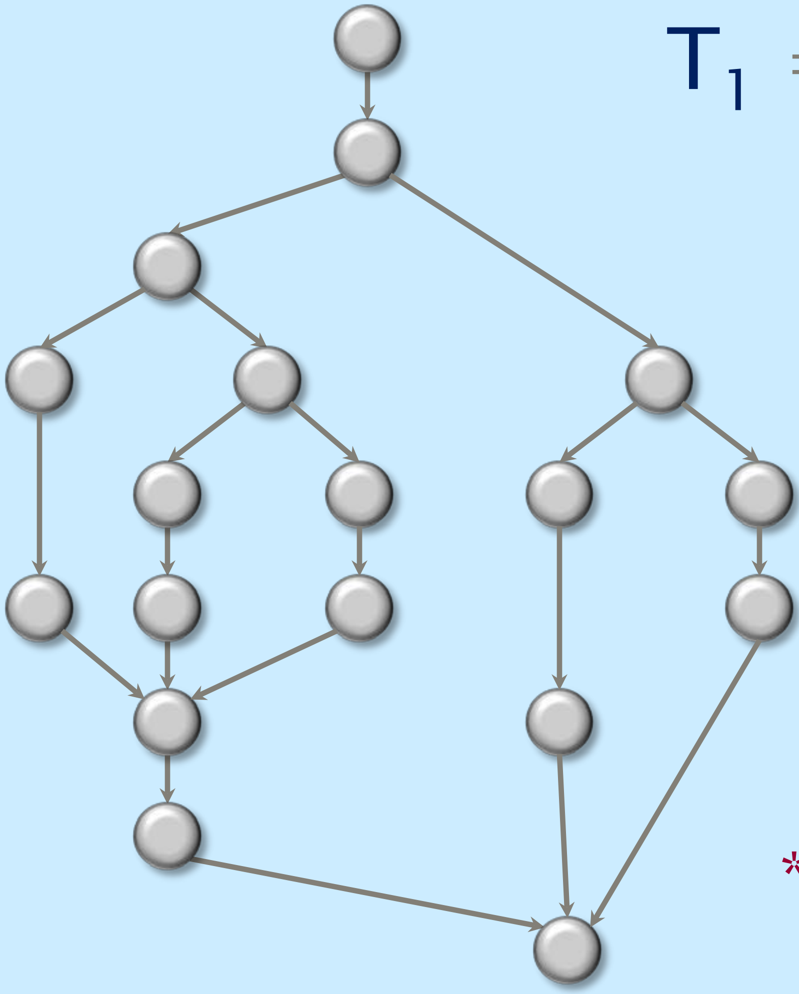
CS 240A : Examples with Cilk++

- Divide & Conquer Paradigm for Cilk++
- Solving recurrences
- Sorting: Quicksort and Mergesort
- Graph traversal: Breadth-First Search

Thanks to Charles E. Leiserson for some of these slides

Work and Span (Recap)

T_P = execution time on P processors



T_1 = ***work*** T_∞ = ***span****

Speedup on p processors

- T_1/T_P

Parallelism

- T_1/T_∞

*Also called ***critical-path length*** or ***computational depth***.

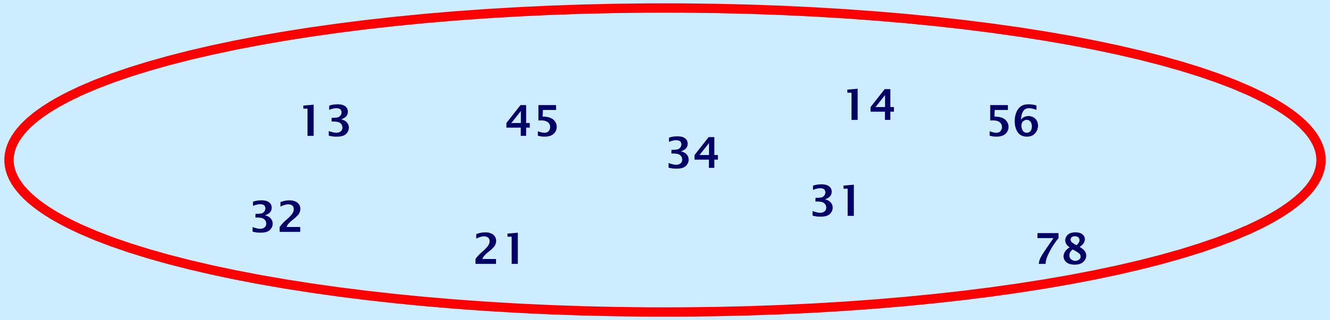
Sorting

- Sorting is possibly the most frequently executed operation in computing!
- **Quicksort** is the fastest sorting algorithm in practice with an average running time of O(N log N), (but $O(N^2)$ worst case performance)
- **Mergesort** has worst case performance of O(N log N) for sorting N elements
- Both based on the recursive **divide-and-conquer** paradigm

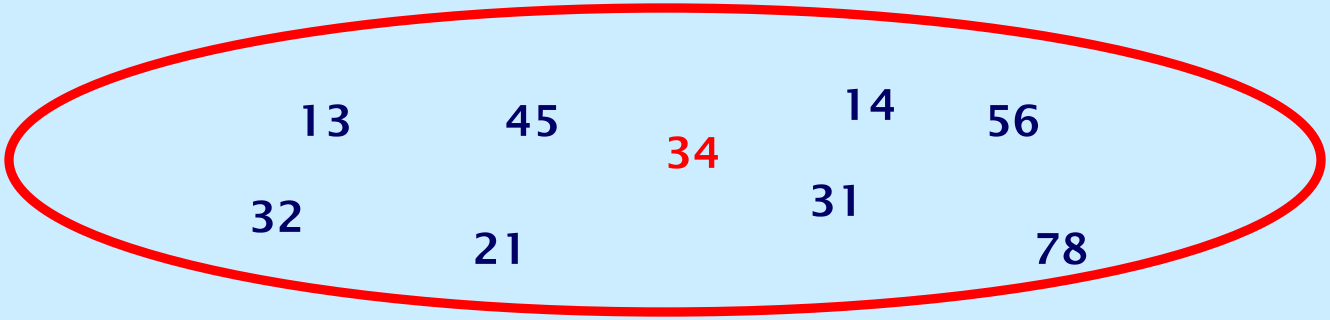
QUICKSORT

- Basic Quicksort sorting an array S works as follows:
  - If the number of elements in S is 0 or 1, then return.
  - Pick any element *v* in S. Call this pivot.
  - Partition the set $S-\{v\}$ into two disjoint groups:
    - $S_1 = \{x \varepsilon S-\{v\} \mid x \leq v\}$
    - $S_2 = \{x \varepsilon S-\{v\} \mid x \geq v\}$
  - **Return quicksort(S_1) followed by v followed by quicksort(S_2)**

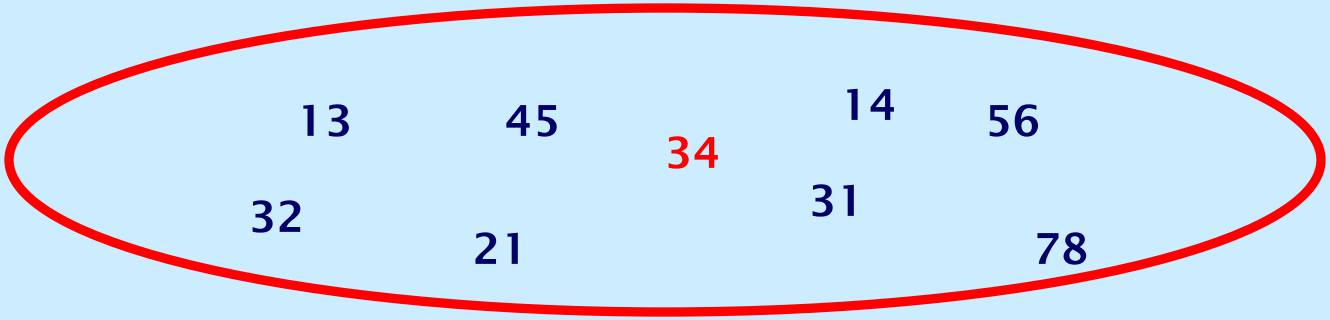
QUICKSORT



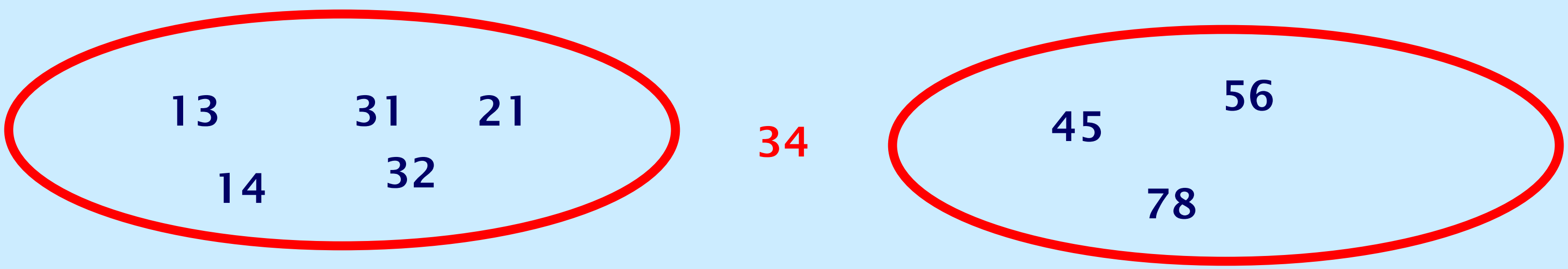
Select Pivot



QUICKSORT



Partition around Pivot



QUICKSORT

13 31 21
14 32

34

45 56
78

Quicksort recursively

13 14 21 31 32

34

45 56 78

13 14 21 31 32 34 45 56 78

Parallelizing Quicksort

- Serial Quicksort sorts an array S as follows:
  - If the number of elements in S is 0 or 1, then return.
  - Pick any element *v* in S. Call this **pivot**.
  - Partition the set S–{v} into two disjoint groups:
    - $S_1 = \{x \in S-\{v\} \mid x \leq v\}$
    - $S_2 = \{x \in S-\{v\} \mid x \geq v\}$
  - Return **quicksort(S_1)** followed by **v** followed by **quicksort(S_2)**

Not necessarily so !

Parallel Quicksort (Basic)

- The second recursive call to *qsort* does not depend on the results of the first recursive call
- We have an opportunity to speed up the call by making both calls in parallel.

```
template <typename T>
void qsort(T begin, T end) {
  if (begin != end) {
     T middle = partition(
                   begin,
                   end,
                   bind2nd( less<typename iterator_traits<T>::value_type>(),
                            *begin )
                );
     cilk_spawn qsort(begin, middle);
     qsort(max(begin + 1, middle), end);
     cilk_sync;
  }
}
```

Performance

- ./qsort 500000 -cilk_set_worker_count 1
  >> 0.083 seconds
- ./qsort 500000 -cilk_set_worker_count 16
  >> 0.014 seconds
- Speedup = T_1/T_{16} = 0.083/0.014 = **5.93**

- ./qsort 50000000 -cilk_set_worker_count 1
  >> 10.57 seconds
- ./qsort 50000000 -cilk_set_worker_count 16
  >> 1.58 seconds
- Speedup = T_1/T_{16} = 10.57/1.58 = **6.67**

Measure Work/Span Empirically

- cilkscreen -w ./qsort 50000000

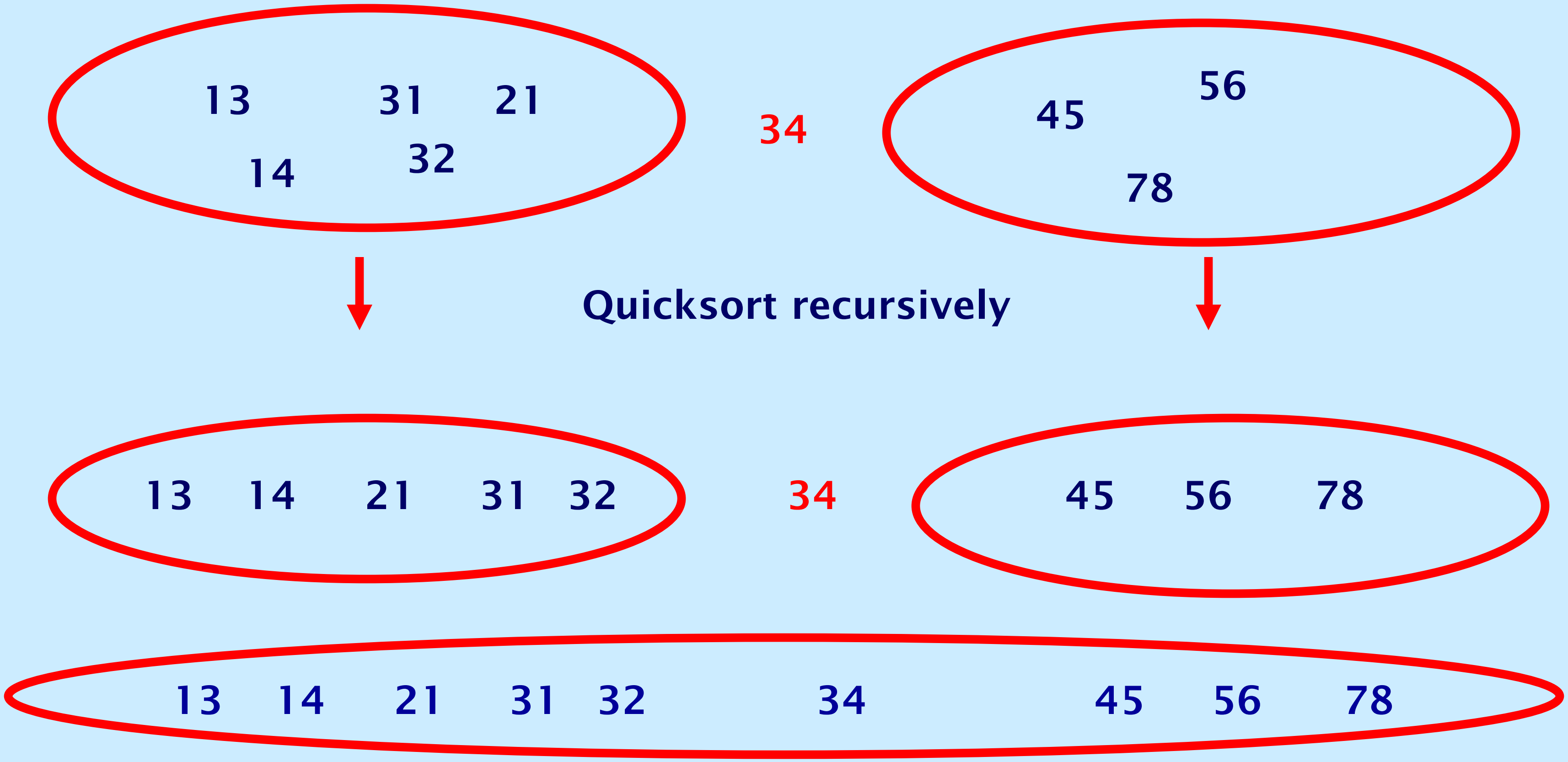
  Work = 21593799861
  Span = 1261403043
  Burdened span = 1261600249
  Parallelism = ***17.1189***
  Burdened parallelism = 17.1162
  #Spawn = 50000000
  #Atomic instructions = 14

- cilkscreen -w ./qsort 500000

  Work = 178835973
  Span = 14378443
  Burdened span = 14525767
  Parallelism = ***12.4378***
  Burdened parallelism = 12.3116
  #Spawn = 500000
  #Atomic instructions = 8

```
workspan ws;
ws.start();
sample_qsort(a, a + n);
ws.stop();
ws.report(std::cout);
```

Analyzing Quicksort



Assume we have a "great" partitioner
that always generates two balanced sets

Analyzing Quicksort

- Work:

$$T_1(n) = 2T_1(n/2) + \Theta(n)$$
$$2T_1(n/2) = 4T_1(n/4) + 2\,\Theta(n/2)$$
$$\ldots.$$
$$\ldots.$$
$$+ \quad n/2\, T_1(2) = n\, T_1(1) + n/2\, \Theta(2)$$

$$\boxed{T_1(n) = \Theta(n \lg n)}$$

•Partitioning
•not parallel !

- Span recurrence: $T_\infty(n) = T_\infty(n/2) + \Theta(n)$

Solves to $\boxed{T_\infty(n) = \Theta(n)}$

Analyzing Quicksort

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(\lg n)$$

Not much !

- Indeed, partitioning (i.e., constructing the array $S_1 = \{x \in S - \{v\} \mid x \leq v\}$) can be accomplished in parallel in time $\Theta(\lg n)$
- Which gives a span $T_\infty(n) = \Theta(\lg^2 n)$
- And parallelism $\Theta(n/\lg n)$

Way better !

- Basic parallel qsort can be found under `$cilkpath/examples/qsort`

The Master Method (Optional)

The ***Master Method*** for solving recurrences applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\ ,^{*}$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

IDEA: Compare $n^{\log_b a}$ with $f(n)$.

* The unstated base case is $T(n) = \Theta(1)$ for sufficiently small n.

Master Method — Case 1

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \gg f(n)$$

Specifically, $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.

Solution: $T(n) = \Theta(n^{\log_b a})$.

Master Method — CASE 2

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \approx f(n)$$

Specifically, $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n))$.

Ex(qsort): $a = 2$, $b=2$, $k=0$ $\rightarrow$ $T_1(n)=\Theta(n \lg n)$

Master Method — Case 3

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \ll f(n)$$

Specifically, $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and $f(n)$ satisfies the ***regularity condition*** that $a\,f(n/b) \le c\,f(n)$ for some constant $c < 1$.

Solution: $T(n) = \Theta(f(n))$.

Example: Span of qsort

Master Method Summary

$$T(n) = a\,T(n/b) + f(n)$$

CASE 1: $f(n) = O(n^{\log_b a - \epsilon})$, constant $\epsilon > 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$, constant $\epsilon > 0$, and regularity condition

$\Rightarrow T(n) = \Theta(f(n))$.

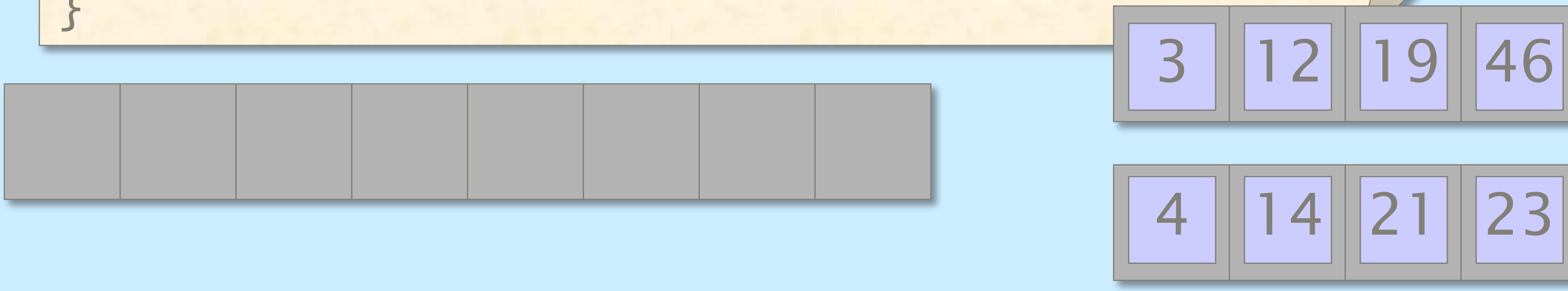
MERGESORT

- Mergesort is an example of a recursive sorting algorithm.
- It is based on the divide-and-conquer paradigm
- It uses the merge operation as its fundamental component (which takes in two sorted sequences and produces a single sorted sequence)
- Simulation of Mergesort
- Drawback of mergesort: Not in-place (uses an extra temporary array)

Merging Two Sorted Arrays

```
template <typename T>
void Merge(T *C, T *A, T *B, int na, int nb) {
  while (na>0 && nb>0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na>0) {
    *C++ = *A++; na--;
  }
  while (nb>0) {
    *C++ = *B++; nb--;
  }
}
```

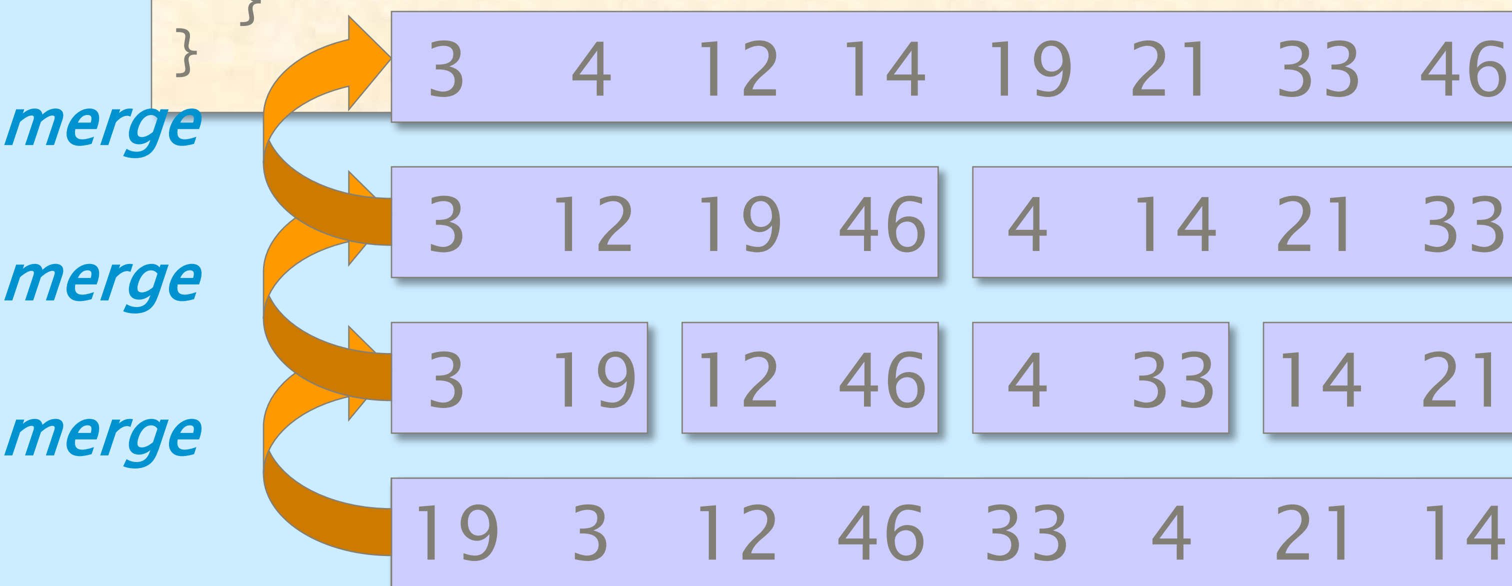
Time to merge n elements = $\Theta(n)$.



Parallel Merge Sort

```
template <typename T>
void MergeSort(T *B, T *A, in
  if (n==1) {
    B[0] = A[0];
  } else {
    T* C = new T[n];
    cilk_spawn MergeSort(C, A, n/2);
               MergeSort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    Merge(B, C, C+n/2, n/2, n-n/2);
    delete[] C;
  }
}
```

A: input (unsorted)
B: output (sorted)
C: temporary



Work of Merge Sort

```
template <typename T>
void MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T* C = new T[n];
    cilk_spawn MergeSort(C,
               MergeSort(C+
    cilk_sync;
    Merge(B, C, C+n/2, n/2,
    delete[] C;
  }
}
```

CASE 2:

$n^{\log_b a} = n^{\log_2 2} = n$

$f(n) = \Theta(n^{\log_b a} \lg^0 n)$

Work: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$

Span of Merge Sort

```
template <typename T>
void MergeSort(T *B, T *A, i
   if (n==1) {
      B[0] = A[0];
   } else {
      T* C = new T[n];
      cilk_spawn MergeSort(C,
                 MergeSort(C+n/2,                 n-n/2);
      cilk_sync;
      Merge(B, C, C+n/2, n/2, n-       );
      delete[] C;
   }
}
```

CASE 3:

$n^{\log_b a} = n^{\log_2 1} = 1$

$f(n) = \Theta(n)$

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(n)$

$= \Theta(n)$

Parallelism of Merge Sort

Work: $T_1(n) = \Theta(n \lg n)$

Span: $T_\infty(n) = \Theta(n)$

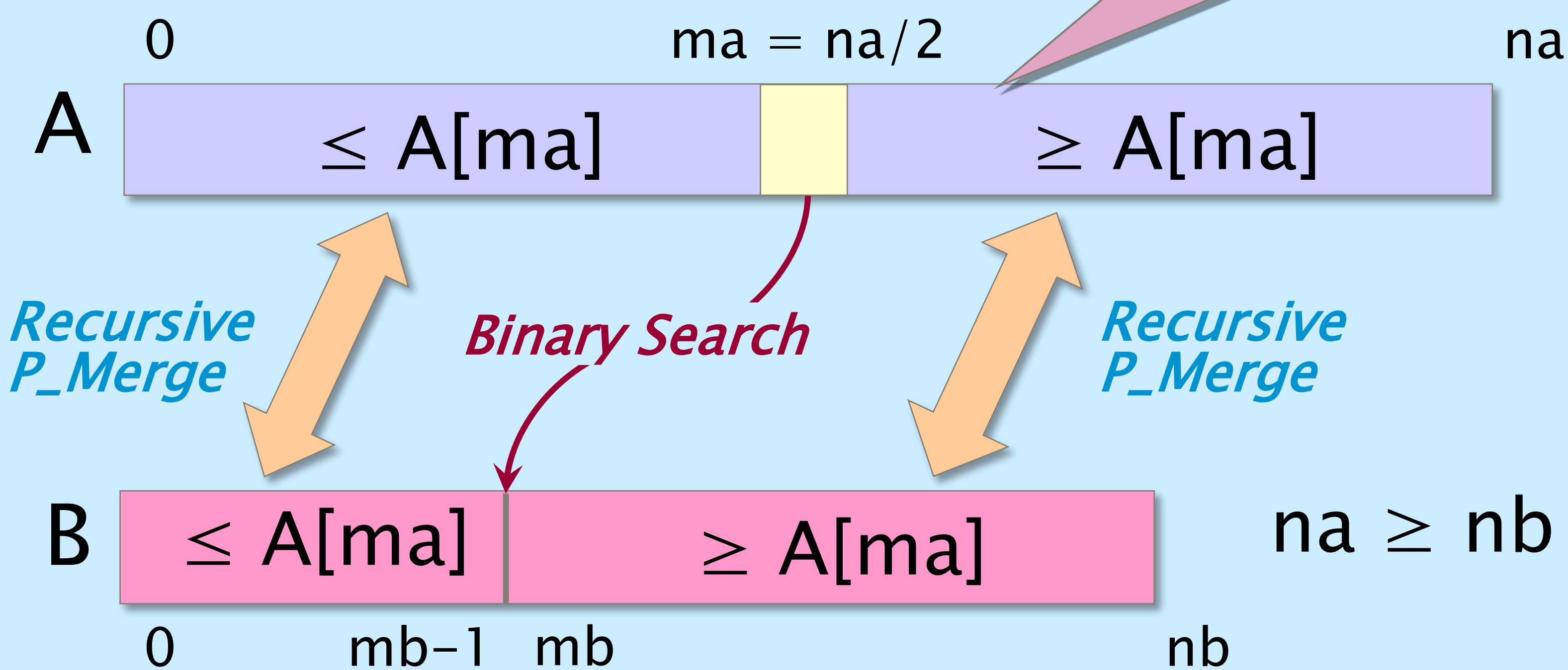
PUNY!

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(\lg n)$$

We need to parallelize the merge!

Parallel Merge

Throw away at least $na/2 \ge n/4$



KEY IDEA: If the total number of elements to be merged in the two arrays is $n = na + nb$, the total number of elements in the larger of the two recursive merges is at most $(3/4)\, n$.

Parallel Merge

```
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
  if (na < nb) {
    P_Merge(C, B, A, nb, na);
  } else if (na==0) {
      return;
  } else {
    int ma = na/2;
    int mb = BinarySearch(A[ma], B, nb);
    C[ma+mb] = A[ma];
    cilk_spawn P_Merge(C, A, B, ma, mb);
    P_Merge(C+ma+mb+1, A+ma+1, B+mb, na-ma-1, nb-mb);
    cilk_sync;
  }
}
```

Coarsen base cases for efficiency.

Span of Parallel Merge

```
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
  if (na < nb) {
    ⋮
    int mb = BinarySearch(A[ma]
    C[ma+mb] = A[ma];
    cilk_spawn P_Merge(C, A, B
    P_Merge(C+ma+mb+1, A+ma+1,
    cilk_sync;
  }
}
```

CASE 2:

$n^{\log_b a} = n^{\log_{4/3} 1} = 1$

$f(n) = \Theta(n^{\log_b a} \lg^1 n)$

Span: $T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n)$

$= \Theta(\lg^2 n)$

Work of Parallel Merge

```
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
  if (na < nb) {
    ⋮
    int mb = BinarySearch(A[ma], B, nb);
    C[ma+mb] = A[ma];
    cilk_spawn P_Merge(C, A, B, ma, mb);
    P_Merge(C+ma+mb+1, A+ma+1, B+mb, na-
    cilk_sync;
  }
}
```

HAIRY!

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,

where $1/4 \le \alpha \le 3/4$.

Claim: $T_1(n) = \Theta(n)$.

Analysis of Work Recurrence

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,
where $1/4 \le \alpha \le 3/4$.

Substitution method: Inductive hypothesis is $T_1(k) \le c_1 k - c_2 \lg k$, where $c_1, c_2 > 0$. Prove that the relation holds, and solve for c_1 and c_2.

$$
\begin{aligned}
T_1(n) &= T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1(\alpha n) - c_2 \lg(\alpha n) \\
&\quad + c_1(1-\alpha)n - c_2 \lg((1-\alpha)n) + \Theta(\lg n)
\end{aligned}
$$

Analysis of Work Recurrence

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,
where $1/4 \le \alpha \le 3/4$.

$$
\begin{aligned}
T_1(n) &= T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1(\alpha n) - c_2\lg(\alpha n) \\
&\quad + c_1(1-\alpha)n - c_2\lg((1-\alpha)n) + \Theta(\lg n)
\end{aligned}
$$

Analysis of Work Recurrence

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$,
where $1/4 \le \alpha \le 3/4$.

$$
\begin{aligned}
T_1(n) &= T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1(\alpha n) - c_2\lg(\alpha n) \\
&\quad + c_1(1-\alpha)n - c_2\lg((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1 n - c_2\lg(\alpha n) - c_2\lg((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1 n - c_2 (\ \lg(\alpha(1-\alpha)) + 2 \lg n\ ) + \Theta(\lg n) \\
&\le c_1 n - c_2 \lg n \\
&\quad - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\le c_1 n - c_2 \lg n
\end{aligned}
$$

by choosing c_2 large enough. Choose c_1 large enough to handle the base case.

Parallelism of P_Merge

Work: $T_1(n) = \Theta(n)$

Span: $T_\infty(n) = \Theta(\lg^2 n)$

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$$

Parallel Merge Sort

```
template <typename T>
void P_MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T C[n];
    cilk_spawn P_MergeSo
    P_MergeSort(C+n/2, A+n/2          2);
    cilk_sync;
   P_Merge(B, C, C+n/2, n/    n-n/2);
  }
}
```

CASE 2:

$n^{\log_b a} = n^{\log_2 2} = n$

$f(n) = \Theta(n^{\log_b a} \lg^0 n)$

Work: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$

Parallel Merge Sort

```
template <typename T>
void P_MergeSort(T *B, T *A, int n) {
   if (n==1) {
      B[0] = A[0];
   } else {
      T C[n];
      cilk_spawn P_MergeSor
      P_MergeSort(C+n/2, A+n/2            );
      cilk_sync;
     P_Merge(B, C, C+n/2, n/       -n/2);
   }
}
```

CASE 2:

$n^{\log_b a} = n^{\log_2 1} = 1$

$f(n) = \Theta(n^{\log_b a} \lg^2 n)$

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n)$

$= \Theta(\lg^3 n)$

Parallelism of P_MergeSort

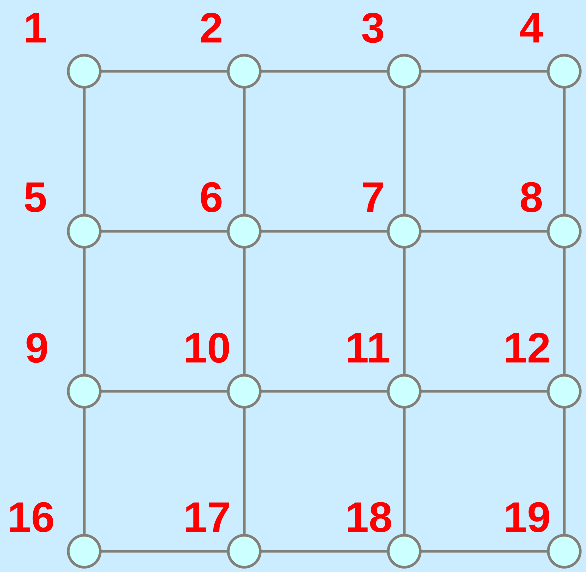
Work: $T_1(n) = \Theta(n \lg n)$

Span: $T_\infty(n) = \Theta(\lg^3 n)$

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$$

Breadth First Search

- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$



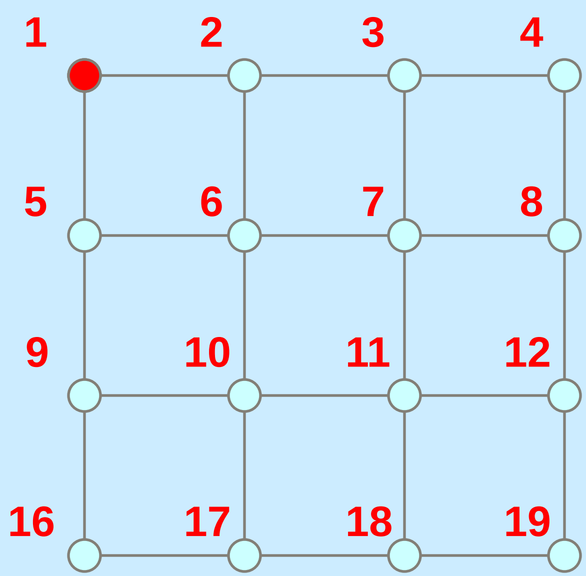
Graph: **G(E,V)**

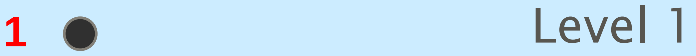
E: Set of edges (size m)

V: Set of vertices (size n)

Breadth First Search

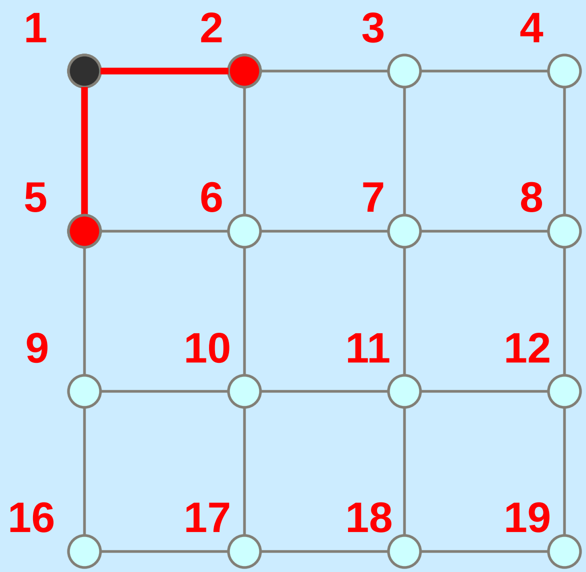
- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$

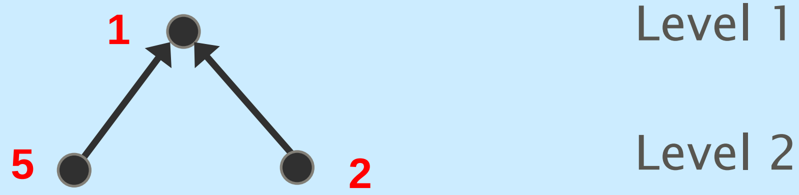




Breadth First Search

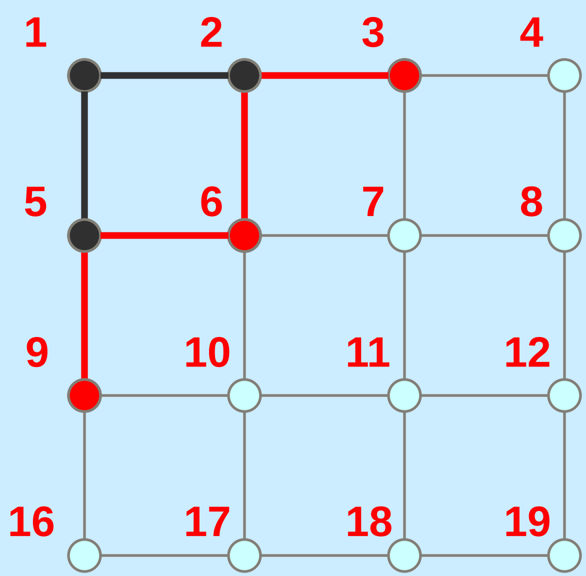
- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$

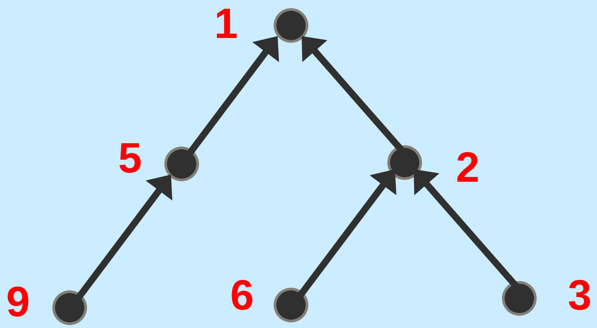




Breadth First Search

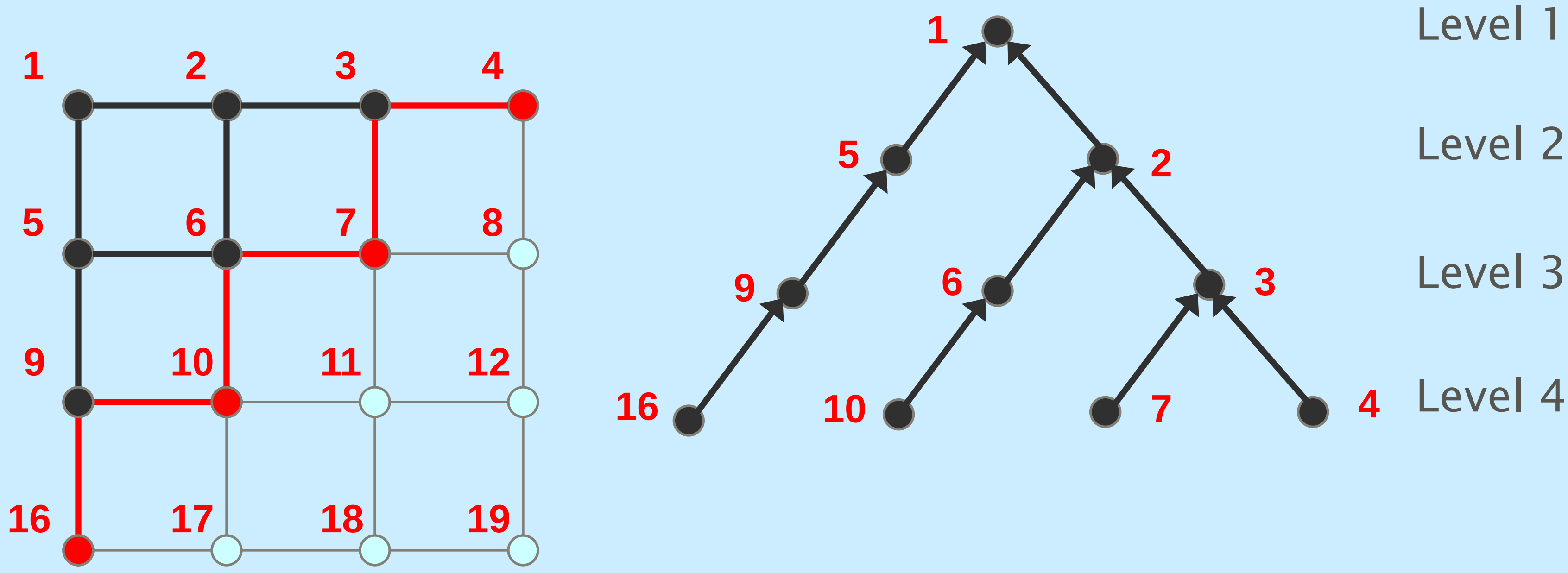
- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$





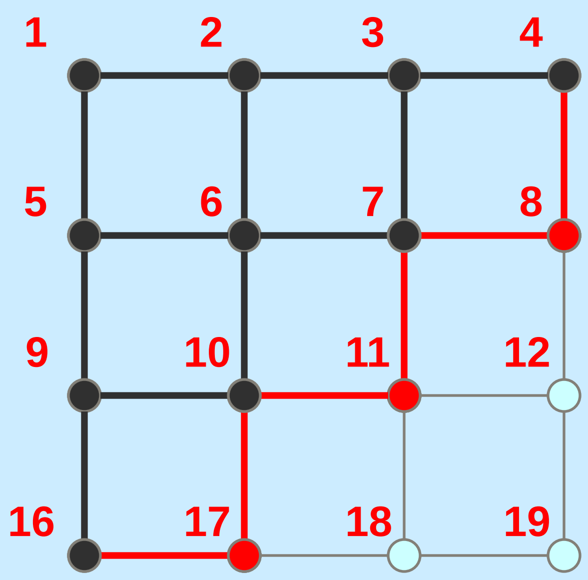
Breadth First Search

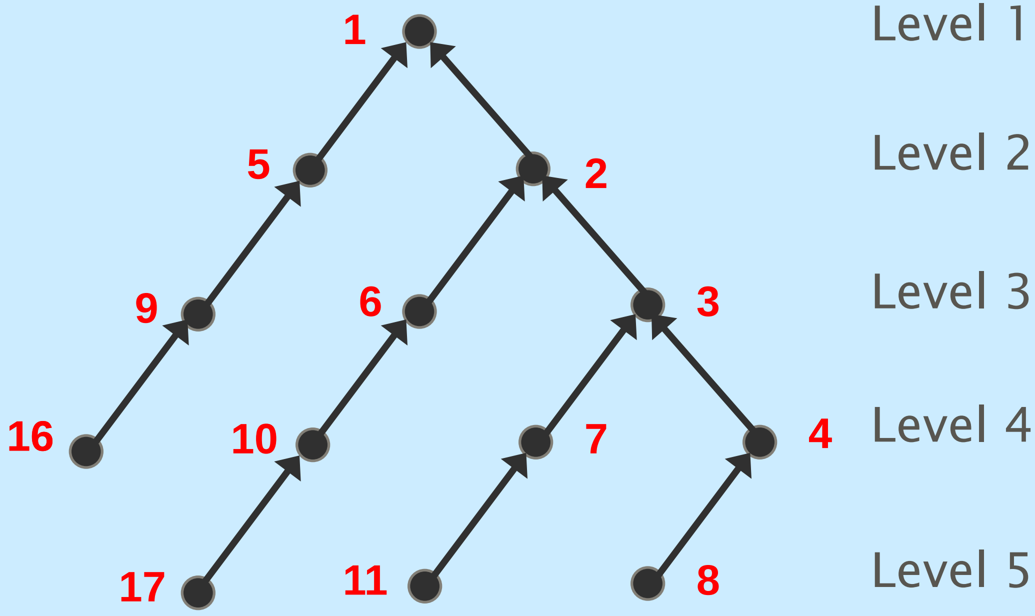
- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$



Breadth First Search

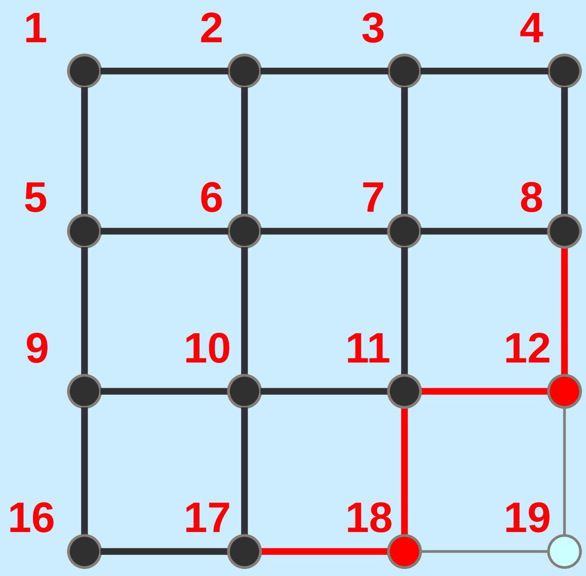
- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$

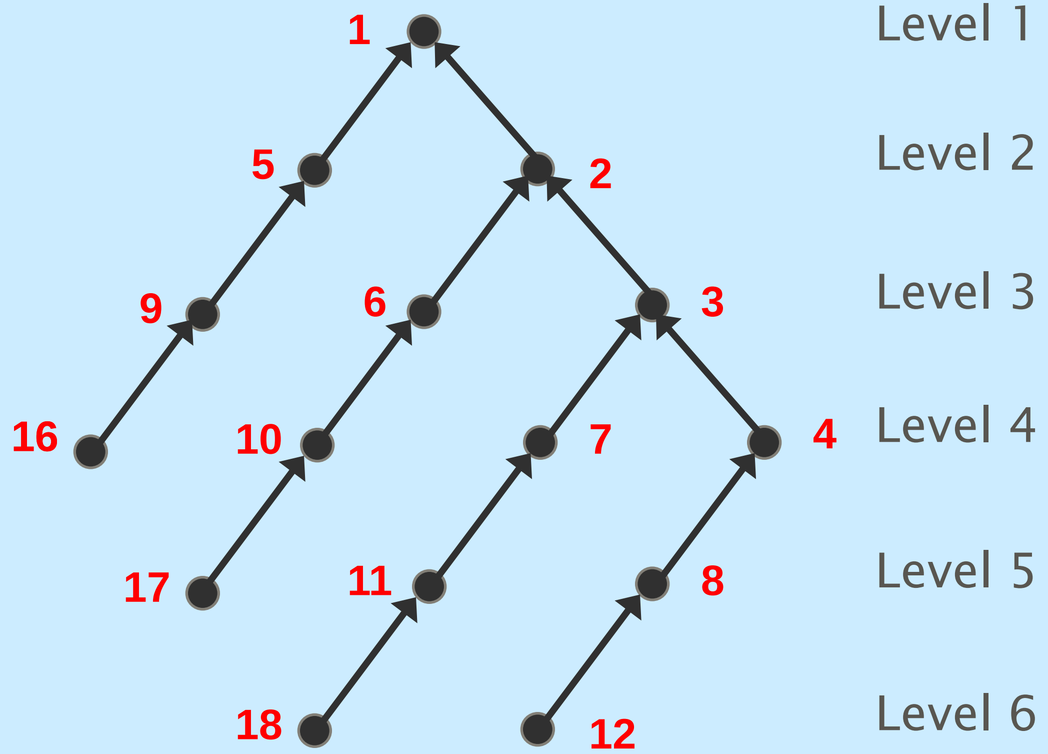




Breadth First Search

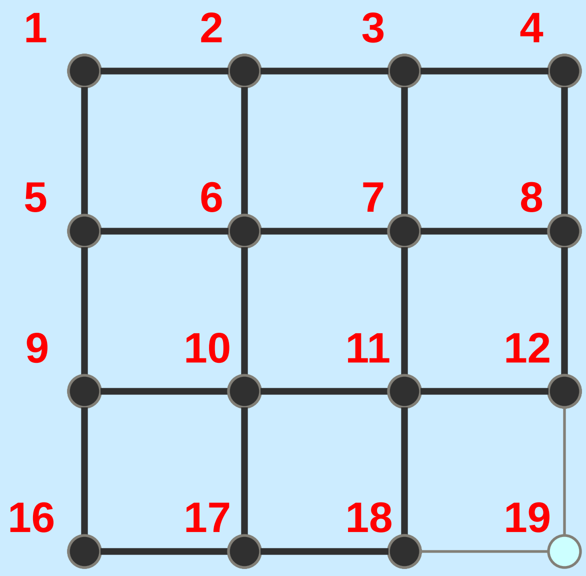
- Level-by-level graph traversal
- Serially complexity: $\Theta(m+n)$

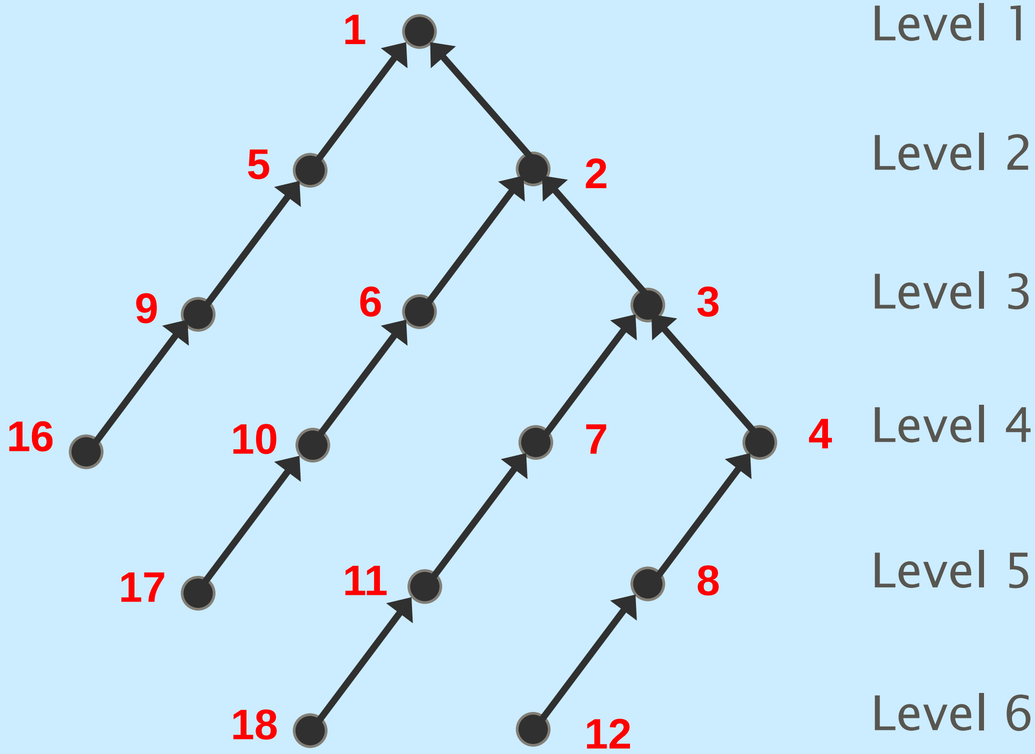




Breadth First Search

- Who is parent(19)?
  - If we use a queue for expanding the frontier?
  - Does it actually matter?





Parallel BFS

- Way #1: A custom

Bag<T> has an associative reduce function that merges two sets

```
void BFS(Graph *G, Vertex root
{
    Bag<Vertex> frontier(roo
    while ( ! frontier.isEmpty() )
    {
        cilk::hyperobject< Bag<Vertex> > succbag();
        cilk_for (int i=0; i< frontier.size(); i++)
        {
            for( Vertex v in frontier[i].adjacency() )
            {
                  if( ! v.unvisited() )
                        succbag() += v;
            }
        }
        frontier = succbag
    }
}
```

operator+=(Vertex & rhs) also marks rhs “visited”

Parallel BFS

- Way #2: Concurrent writes + List reducer

```
void BFS(Graph *G, Vertex root)
{
    list<Vertex> frontier(root);
    Vertex * parent = new Vertex[n];
    while ( ! frontier.isEmpty() )
    {
        cilk_for (int i=0; i< frontier.size(); i++)
        {
            for( Vertex v in frontier[i].adjacency() )
            {
                if ( ! v.visited() )
                    parent[v] =
            }
        }
        ...
    }
}
```

An intentional data race

How to generate the new frontier?

Parallel BFS

```
void BFS(Graph *G, Vertex root)
{
    ...
    while ( ! frontier.isEmpty( ) ) {
        ...
        hyperobject< reducer_list_append<Vertex> > succlist();
        cilk_for (int i=0; i< frontier.size(); i++)
        {
            for( Vertex v in frontier[i].adjacency() ) {
                if ( parent[v] == frontier[i] )
                {
                    succlist.push_back(v);
                    v.visit(); // Mark "visited"
                }
            }
        }
        frontier = succlist.getValue();
    }
}
```

Run cilk_for loop again

!v.visited() check is not necessary. Why?

Parallel BFS

- Each level is explored with $\Theta(1)$ span
- Graph G has at most d, at least d/2 levels
  - Depending on the location of root
  - d=diameter(G)

Work: $T_1(n) = \Theta(m+n)$

Span: $T_\infty(n) = \Theta(d)$

Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta((m+n)/d)$

Parallel BFS Caveats

- d is usually small
- d = lg(n) for scale-free graphs
  - But the degrees are not bounded ☹
- Parallel scaling will be memory-bound
- Lots of burdened parallelism,
  - Loops are skinny
  - Especially to the root and leaves of BFS-tree
- You are not “expected” to parallelize BFS part of Homework #5
  - You may do it for “extra credit” though ☺