Roll up and Conquer

John Feo
Cray Inc

CUG 2006
Lugano, Switzerland

Introduction

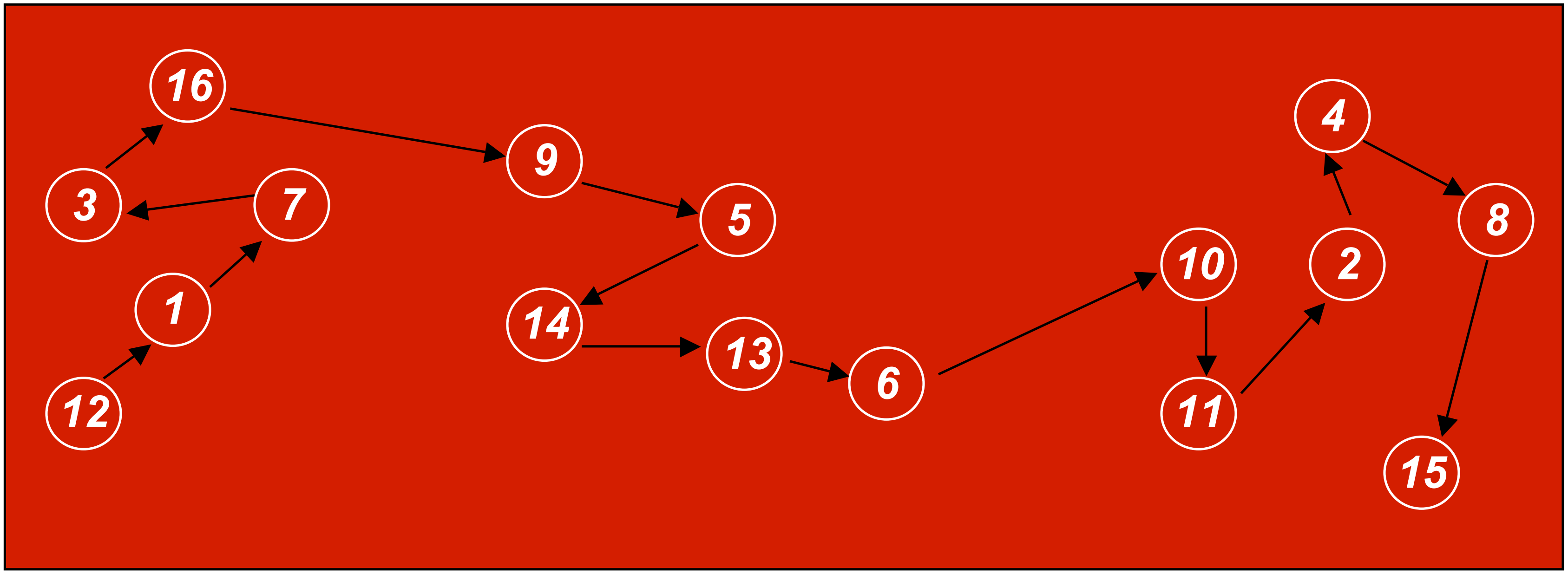
- Divide-and-conquer is a well known technique that divides a large problem into smaller problems, solves the small problems, and then reduces their solutions

- An opposite technique appears to work well for many graph problems
  - roll up the graph into super nodes
  - solve the problem for the graph of super nodes
  - extend the solution of each super node to its included nodes

Prefix operations on lists

- $\mathbf{P}(1)$ = initial value
  $\mathbf{P}(i) = \mathbf{P}(i - 1) \oplus \mathbf{V}(i), \qquad i > 1$
- If initial value is 1, operator is +, and $\mathbf{V}(i)$ is 1 for all i, then $\mathbf{P}(i)$ is the rank of node i
- List ranking is a common procedure that occurs in many graph algorithms and is considered the HPL of graph algorithms

A linked list



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
List	7	4	16	8	14	10	3	15	5	11	2	1	6	13	0	9
Rank	2	13	4	14	7	10	3	15	6	11	12	1	9	8	16	5

Sequential algorithm

- Find start of list

$$\frac{n^2+n}{2} - \sum_{i=1}^{n} List(i)$$

- Walk list from start to end
- $O(n)$ in number of instructions and time

Parallel algorithm (cyclic reduction)

- **Rank**(i) = 1, $1 \le i \le n$
- **Rank**(**List**(i)) += **Rank**(i), $1 \le i \le n$, **List**(i) $\neq 0$
- **List**(i) = **List**(**List**(i)), $1 \le i \le n$, **List**(i) $\neq 0$
- Repeat steps 2 and 3 until **List**(i) = 0, $\forall\ i$

Steps 2 and 3

Rank	1	1	1	1	1	1
List	2	3	4	5	6	0

Rank	1	2	2	2	2	2
List	3	4	5	6	0	0

Rank	1	2	3	4	4	4
List	5	6	0	0	0	0

Rank	1	2	3	4	5	6
List	0	0	0	0	0	0

$O(n \text{ Log } n)$ in number of instructions

$O\left(\frac{n \text{ Log } n}{P}\right)$ in time

We can do better …

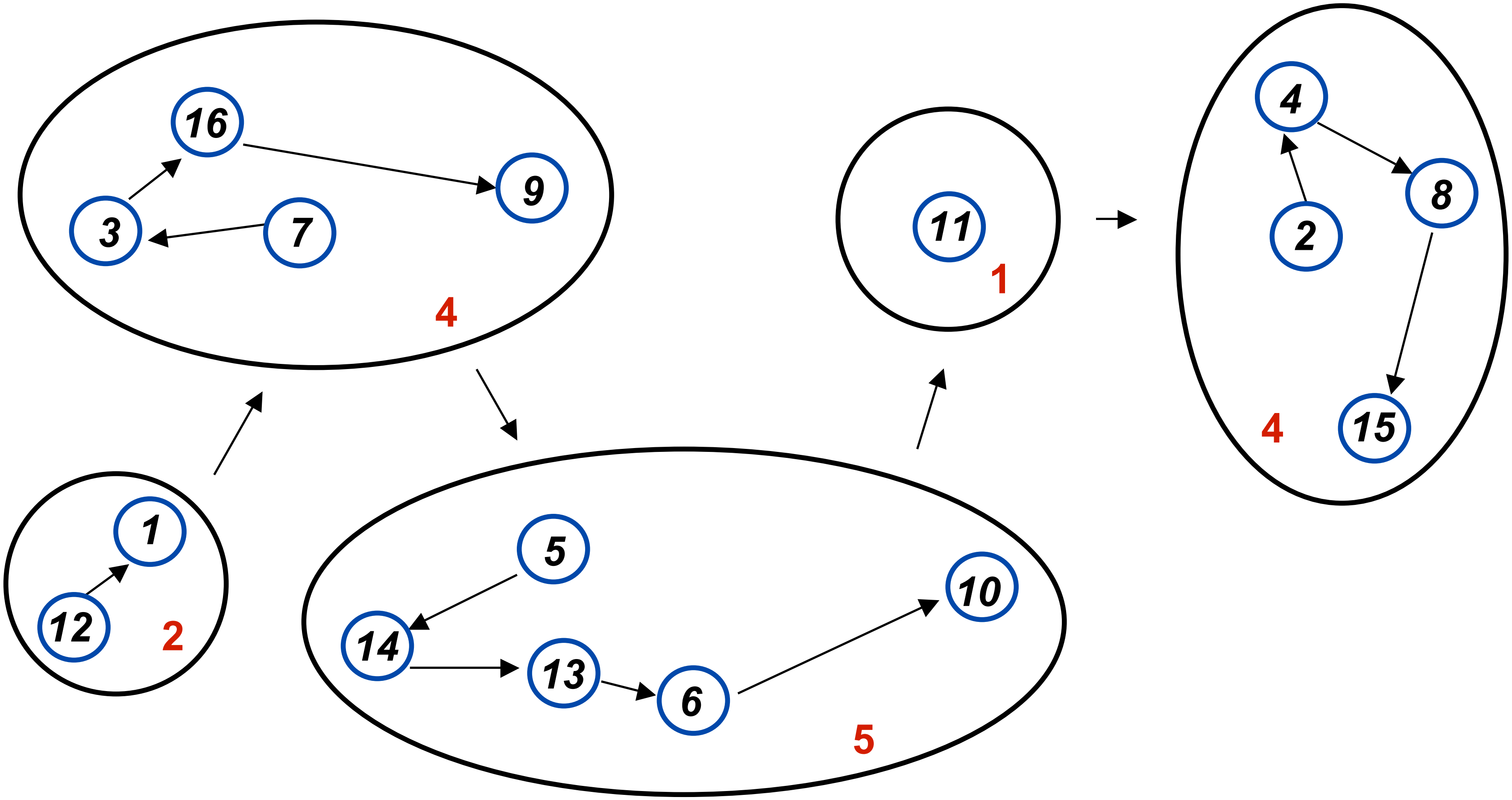
- By **rolling up** the graph into super nodes we can reduce both the number of operations and time to O(n)

- First, run the sequential algorithm to roll up the graph, and then run the parallel algorithm on the rolled up graph; finally, run the sequential algorithm on each super node to compute the rank of each original node

- O(n) + O(S log S) + O(n), where S is the number of super nodes

- A variation of the Hellman & JaJa algorithm for list ranking and the algorithm used by the MTA compiler to parallelize recurrences

First step



Second step

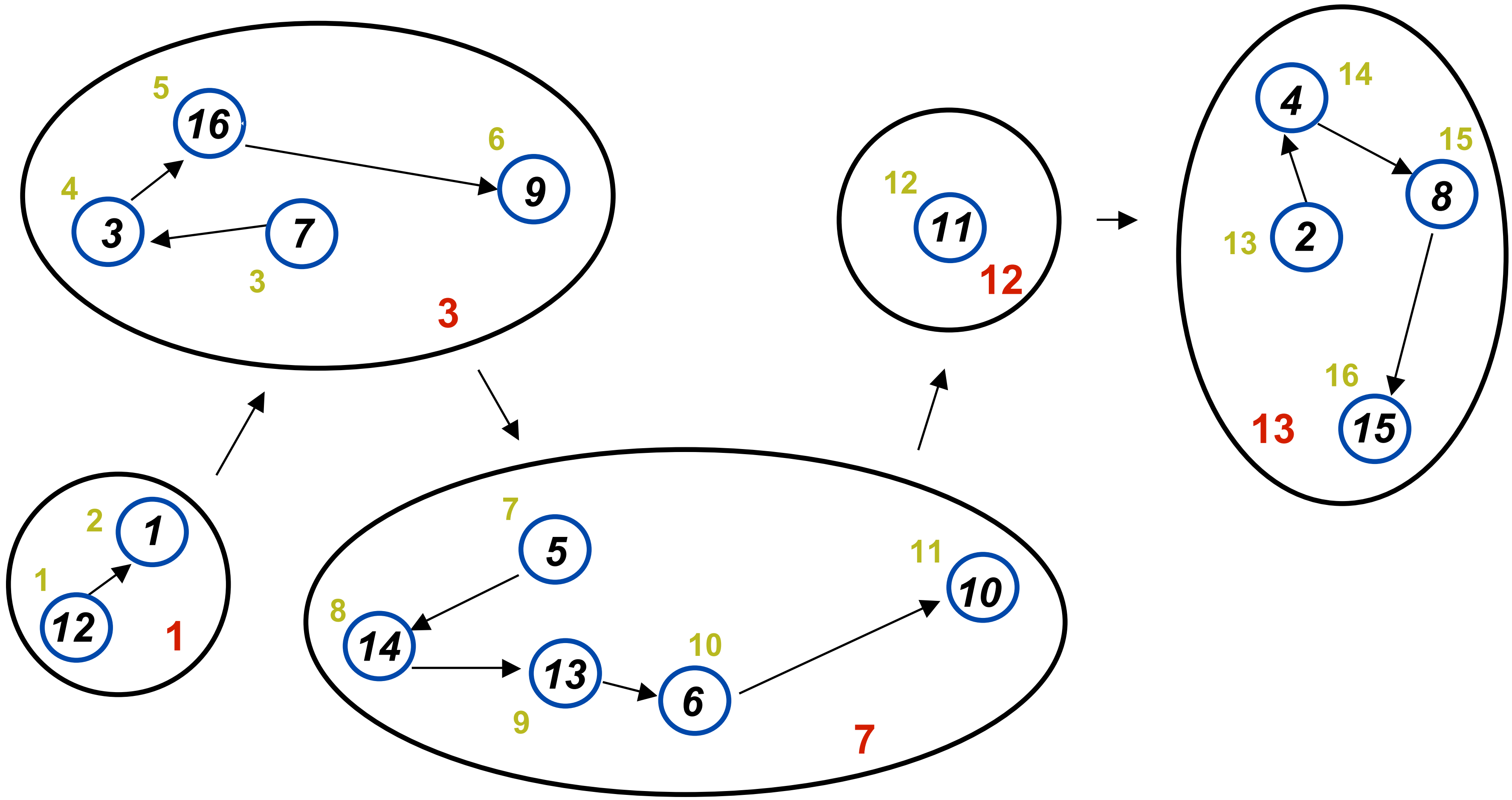
Rank	1	2	4	5	1
List	2	3	4	5	0

Rank	1	3	6	9	6
List	3	4	5	0	0

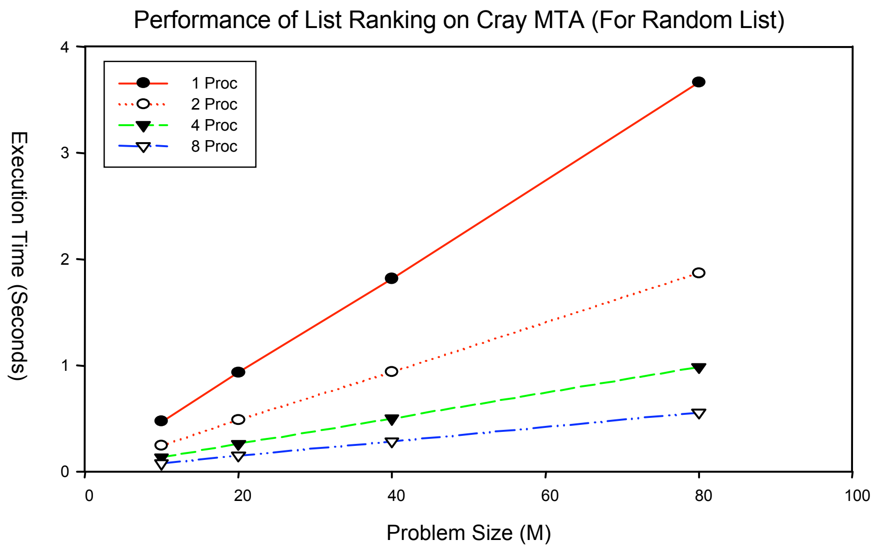
Rank	1	3	7	12	12
List	5	0	0	0	0

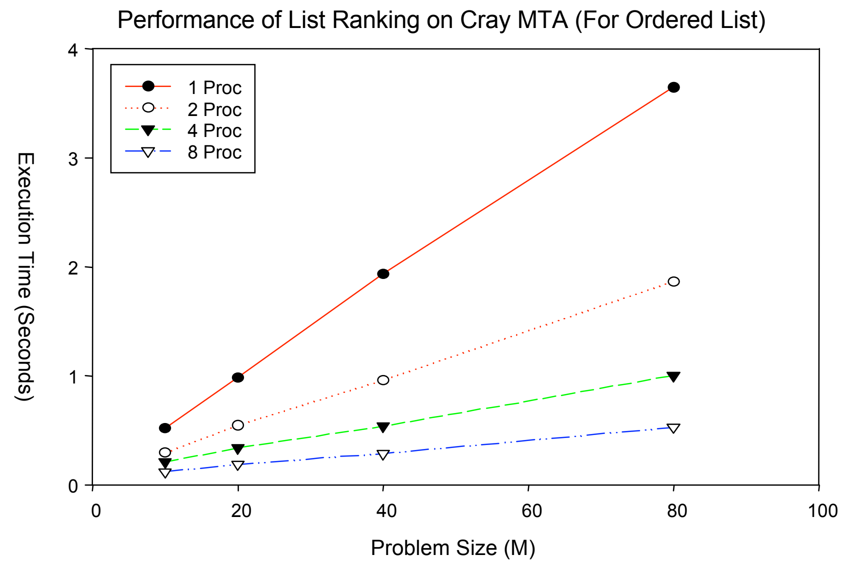
Rank	1	3	7	12	13
List	0	0	0	0	0

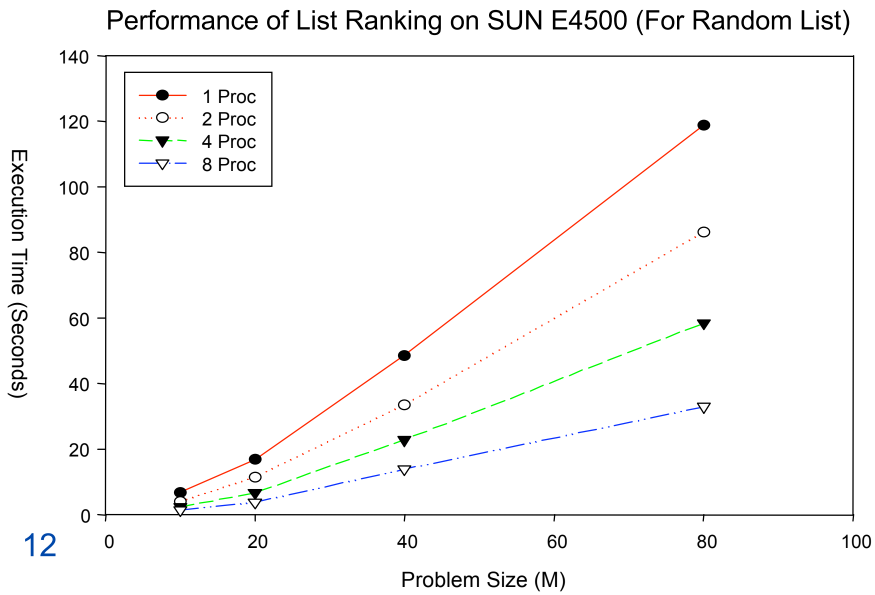
Third step

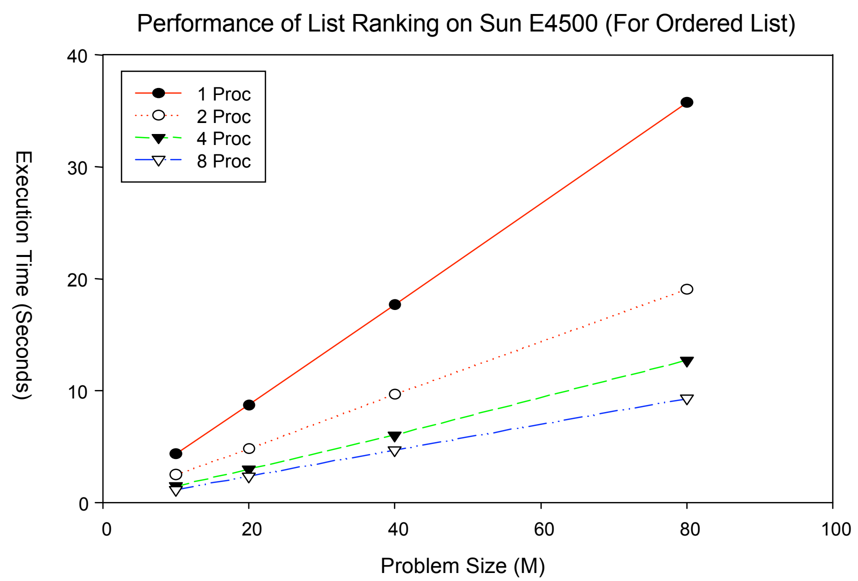

5
16
6
9
4
3
7
3
3
12
11
12
14
4
15
8
13
2
16
13
15
2
1
1
12
1
7
5
11
10
8
14
13
10
6
9
7

Performance comparison



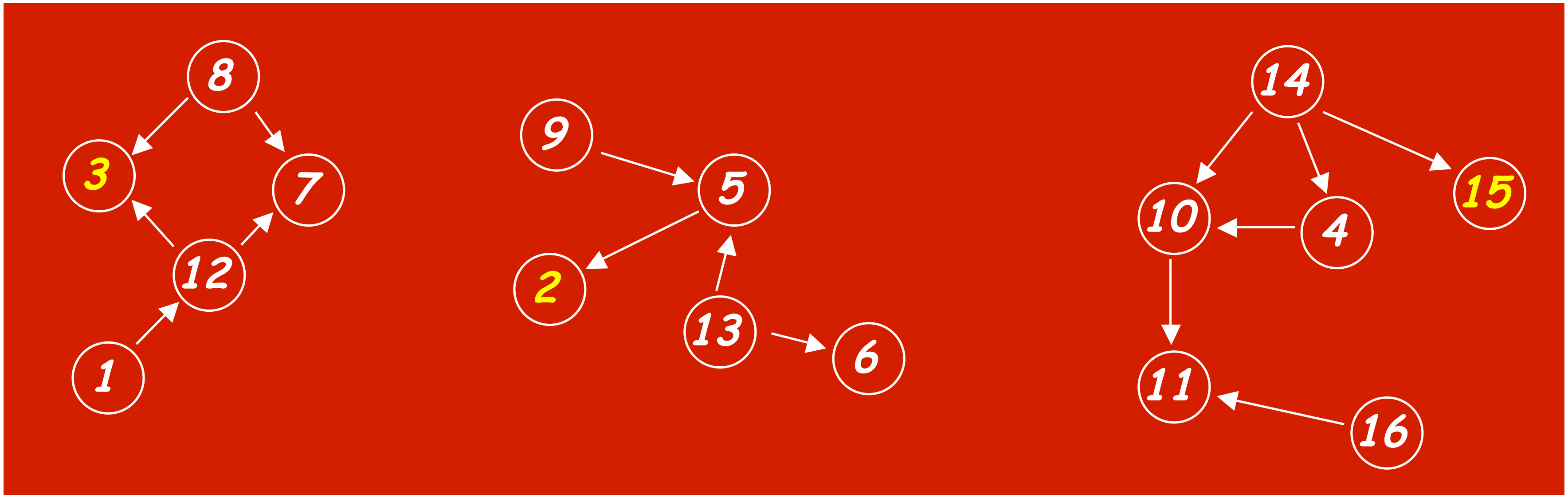






Connected Components

- Label all nodes in a graph such that Label[v] = Label[w] if and only if there is an undirected path between v and w



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Label	3	2	3	15	2	2	3	3	2	15	15	3	2	15	15	15

Sequential algorithms

- Typically, based on either depth-first or breath-first search

```
for all nodes v
     Label[v] = v
for all nodes v
     if v is unvisited, DFS(v)
```

where DFS(v) is

```
for all unvisited neighbors w of v
     Label[w] = Label[v]
     DFS(w)
```

- If we parallelize the loop in the main program, multiple labels move through a component $\Rightarrow$ to merge or clean up is expensive

- If we parallelize the loop in DFS, parallelism is limited for nodes with small degree

Parallel algorithm

- CRCW PRAM algorithm (Shiloach-Vishkin)

```
until no label changes

    for all edges (v, w)          // hook components together
        if Label[v] < Label[w] && Label[v] == Label[Label[v]]
            Label[Label[v]] = Label[w]

    for all nodes v               // compress components
        if Label[v] != Label[Label[v]]
            Label[v] = Label[Label[v]]
```

- Parallel and performance is good, but
  - Outer loop may iterate many times
  - if Label[*v*] = C for many *v*, then **Label[Label[*v*]]** gets hot

Hybrid approach

```
// STEP 1, DFS w/o cleanup, rolls up the graph
  for all nodes v                                              ← parallel loop
       if v is unvisited, mark v as a root and DFS(v)
```

where DFS(v) is

```
  for all unvisited neighbors w of v                           ← parallel loop
       Label[w] = Label[v]
       DFS(w)                                                  ← recursive call
  for all visited neighbors w of v                             ← parallel loop
       store (Label[v], Label[w]) uniquely in a hash table

// STEP 2, run PRAM on rolled up graph
  repeat until no label changes
       for all edges (v, w) in the hash table                  ←parallel loop
            if Label[v] < Label[w] && Label[v] = Label[Label[v]]
                 Label[Label[v]] = Label[w]
       for all roots v                                         ← parallel loop
            if Label[v] != Label[Label[v]]
                 Label[v] = Label[Label[v]]

// STEP 3, relabel
  for all roots v                                              ← parallel loop
       if v is unvisited, mark v as a root and relabel(v)
```

where relabel(v) is

```
  for all unvisited neighbors w of v,                          ← parallel loop
       Label[w] = Label[v] and relabel(w)                      ← recursive call
```

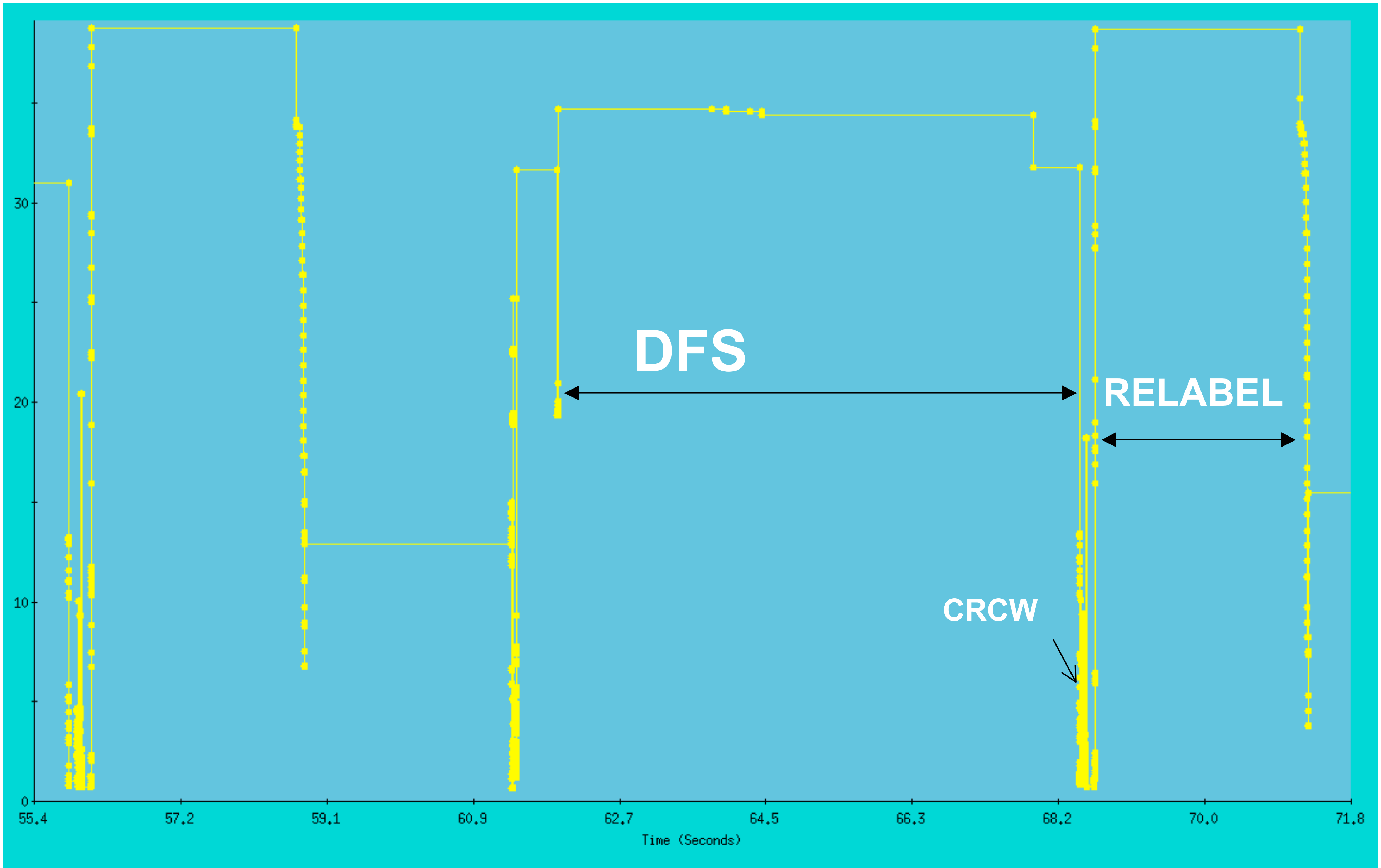
Performance

- Large, pseudo-random graph
  - 67M nodes, 420M edges, 412 components
  - 64 nodes have degree ~2^{20}
  - 64K nodes have degree ~2^{10}
  - the rest have degree ~12

P	Time
20	19.4
30	13.1
40	9.8

That's about 1.25M edges per second per processor !!!

Traceview



DFS
RELABEL
CRCW
0
10
20
30
55.4
57.2
59.1
60.9
62.7
64.5
66.3
68.2
70.0
71.8
Time (Seconds)

Conclusions

- "Roll up and conquer" is a programming technique well-suited for parallel graph algorithms
- We have used the technique to develop high-performance, scalable parallel algorithms for several graph problems
- The MTA's shared-memory, latency-tolerant processors, and full-and-empty bits are critical for good performance