# Parallel Scheduling Theory and Practice

Guy Blelloch

Carnegie Mellon University

# Parallel Languages

User Scheduled

- MPI, Pthreads (typical usage)

System Scheduled

Bulk synchronous (data parallel, SPMD)

- HPF, ZPL, OpenMP, UPC, CUDA

General (dynamic)

- ID, Nesl, Cilk, X10, Fortress

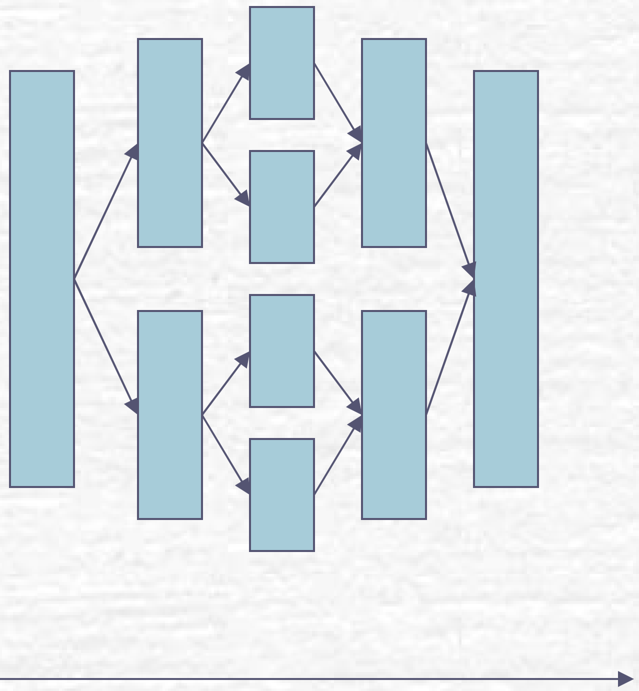
The "general" languages will surely dominate parallel programming in the future.

# Example: Quicksort

```
procedure QUICKSORT(S):
 if S contains at most one element then return S
 else
   begin
     choose an element a randomly from S;
     let S_1, S_2 and S_3 be the sequences of
         elements in S less than, equal to,
         and greater than a, respectively;
     return (QUICKSORT(S_1) followed by S_2
        followed by QUICKSORT(S_3))
end
```

# Parallelism

Parallel Partition and Append



Work = $O(n \log n)$

Span = $O(\lg^2 n)$

# Quicksort in NESL

```
function quicksort(S) =
if (#S <= 1) then S
else let
  a = S[rand(#S)];
  S1 = {e in S | e < a};
  S2 = {e in S | e = a};
  S3 = {e in S | e > a};
  R = {quicksort(v) : v in [S1, S3]};
in R[0] ++ S2 ++ R[1];
```

# Quicksort in X10

```
double[] quicksort(double[] S) {
  if (S.length < 2) return S;
  double a = S[rand(S.length)];
  double[] S1,S2,S3;
  finish {
     async { S1 = quicksort(lessThan(S,a));}
     async { S2 = eqTo(S,a);}
     S3 = quicksort(grThan(S,a));
  }
  append(S1,append(S2,S3));
}
```

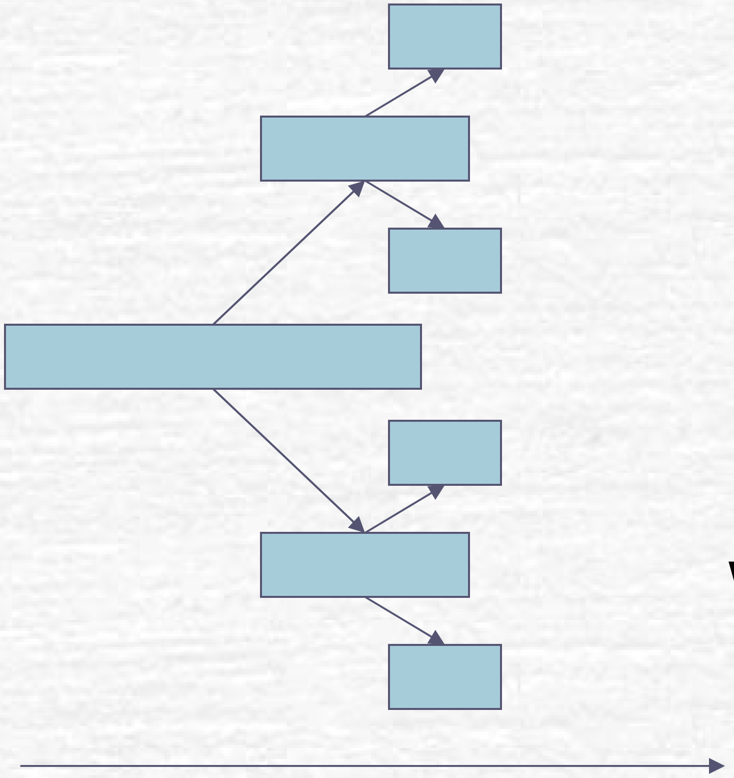
# Quicksort in Multilisp (futures)

```
(defun quicksort (L) (qs L nil))

(defun qs (L rest)
  (if (null L) rest
     (let ((a (car L))
           (L1 (filter (lambda (b) (< b a)) (cdr L)))
           (L3 (filter (lambda (b) (>= b a)) (cdr L))))
       (qs L1 (future (cons a (qs L3 rest)))))))

(defun filter (f L)
  (if (null L) nil
     (if (f (car L))
         (future (cons (car L) (filter f (cdr L))
         (filter f (cdr L))))
```

# Quicksort in Multilisp (futures)

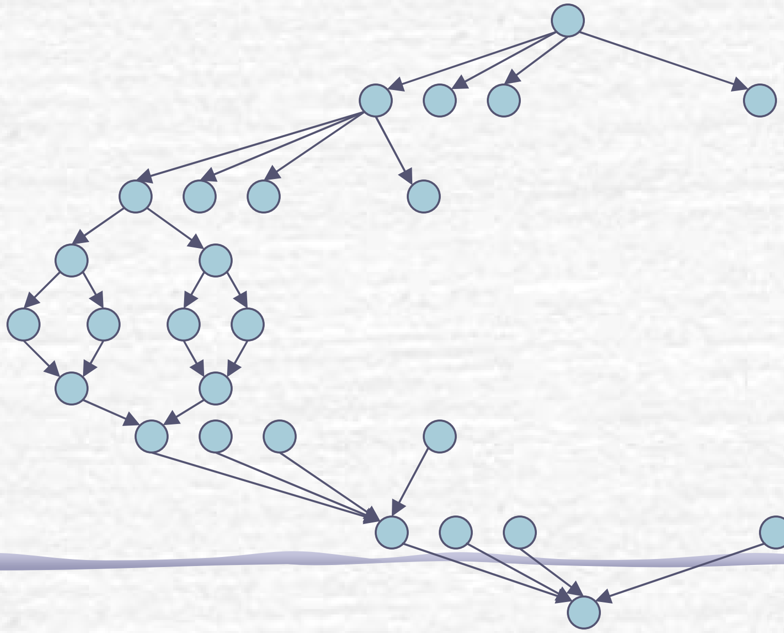


Work = O(n log n)

Span = O(n)

# Example: Matrix Multiply

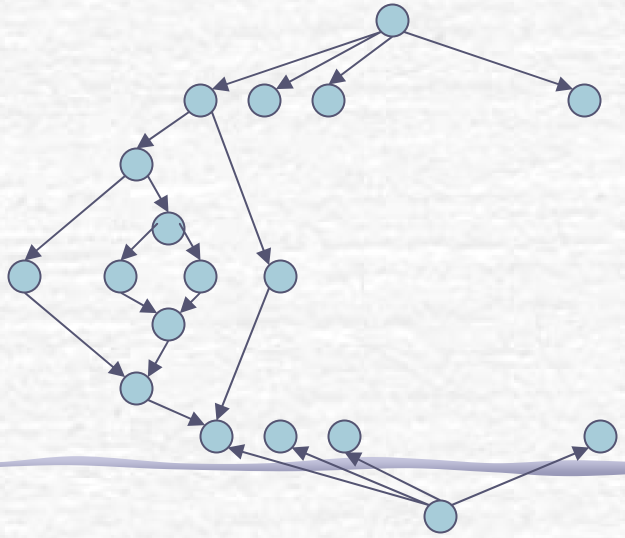
```
for each i in [0:n]
  for each j in [0:n]
    $C[i,j] = \sum_{k=1}^{n} A[i,k] \times B[k,j]$
```



# Example: N-body Tree Code

```
force(p,c)
  if far(p,c) then pointForce(p,center(c))
  else force(p,left(c)) + force(p,right(c))

allForces(P,c)
  foreach p in P, force(p, root)
```



# Generally

- Much more parallelism than processors
- It is all about **scheduling**
  - space usage
  - locality
  - overheads

# Sidebar: Types of Computation

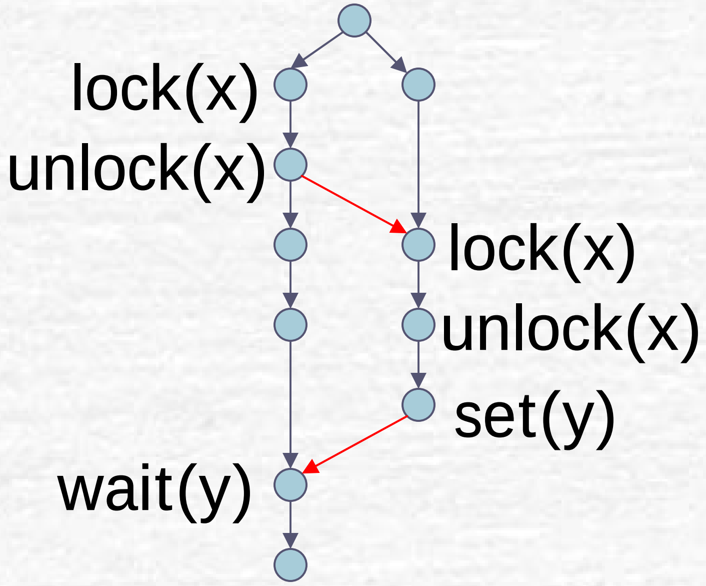
- Assume a way to fork
  - Pairwise or multiway
- What types of synchronization are allowed
  - General
  - Strict and fully strict (fork-join)
  - Futures
  - Clocks
- The last three can be made deterministic
- Can have a large effect on the scheduler and what can be proved about the schedules.

# General

- Locks
- Transactions
- Synch variables

Easy to create deadlock

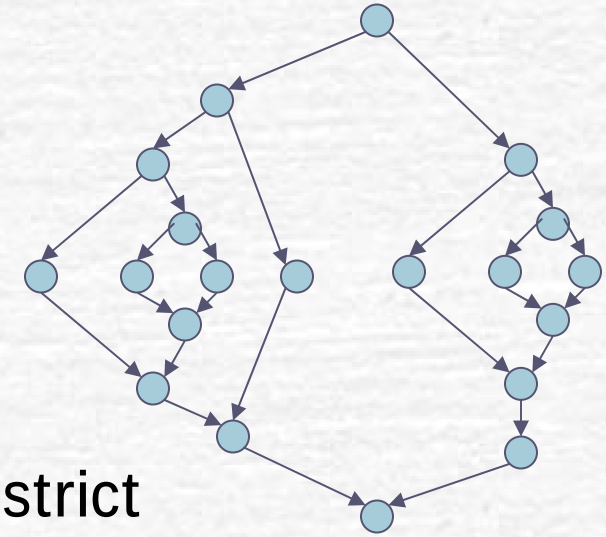
Hard to schedule



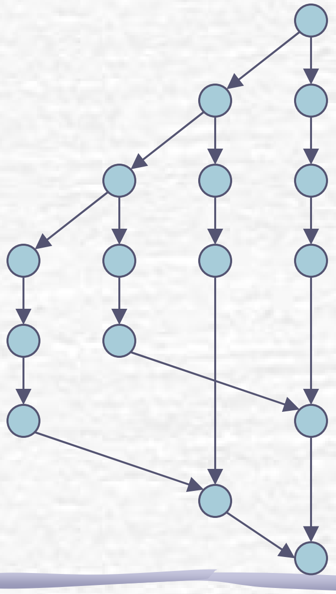
# Strict and Fully Strict

**Fully strict** (fork-join, nested parallel): a task can only synchronize with its parent

**Strict**: a task can only synchronize with an ancestor. (X10 recently extended to support strict computations)



fully strict

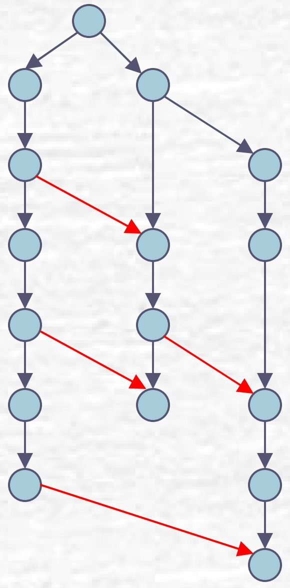


strict

# Futures

Futures or read-write synchronization variables can be used for pipelining of various forms, e.g. **producer consumer pipelines**. This cannot be supported in strict or fully strict computations.

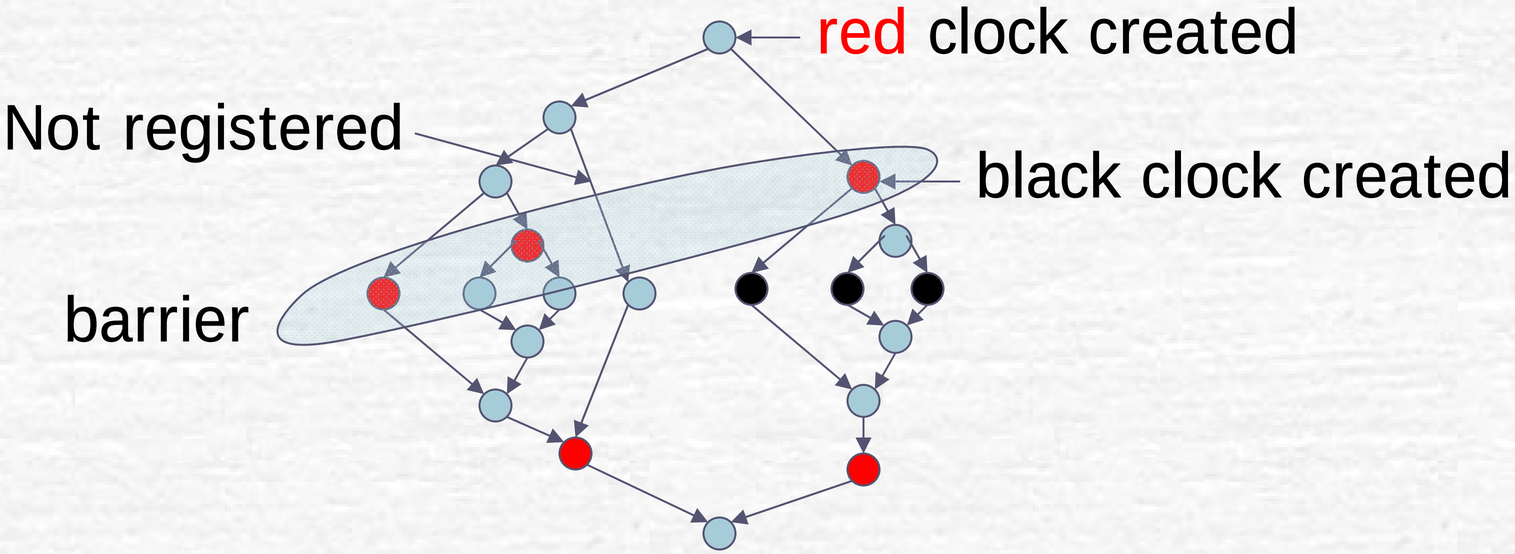
If read always occurs "after" the write in sequential order then there is no deadlock



# Clocks

Clocks generalize barrier synchronizations.

A new idea in X10 and not well understood yet when multiple clocks are used.



# Scheduling Outline

Theoretical results on scheduling

- Graham, 1966
- Eager, Zahorjan, Lazowska, 1989
- Specific schedules
  - Breadth First
  - Work Stealing (Blumofe, Leiserson, 1993)
  - P-DFS (Blelloch, Gibbons, Matias, 1995)
  - Hybrid (Narlikar, 2001)

# Graham

"Bounds on Certain Multiprocessor Anomilies", 1966

Model:

- Processing Units : $P_i$ , $1 \le i \le n$
- Tasks : $T = \{T_i , \ldots , T_m\}$
- Partial order : $<_T$ on T
- Time function : $\mu : T \text{ -> } [0,\infty]$

$(T, <_T , \mu)$ : define a weighted DAG

# Graham: List Scheduling

- Task List L : ($T_{k1}$ , ... , $T_{km}$)
- Task is **ready** when not yet started but all predecessors are finished
- **List scheduling** : when a processor finishes a task it immediately takes the first ready task from L. Ties broken by processor ID.
- Showed that for any L and L'

$$\frac{T(L)}{T(L')} = 1 + \frac{n-1}{n}$$

# Some definitions

- $T_p$ : time on P processors
- W : single processor time
- D : longest path in the DAG

- Lower bound on time : $T_p \geq \max(W/P, D)$

# Greedy Schedules

"Speedup versus Efficiency in Parallel Systems", Eager, Zahorjan and Lazowska, 1989

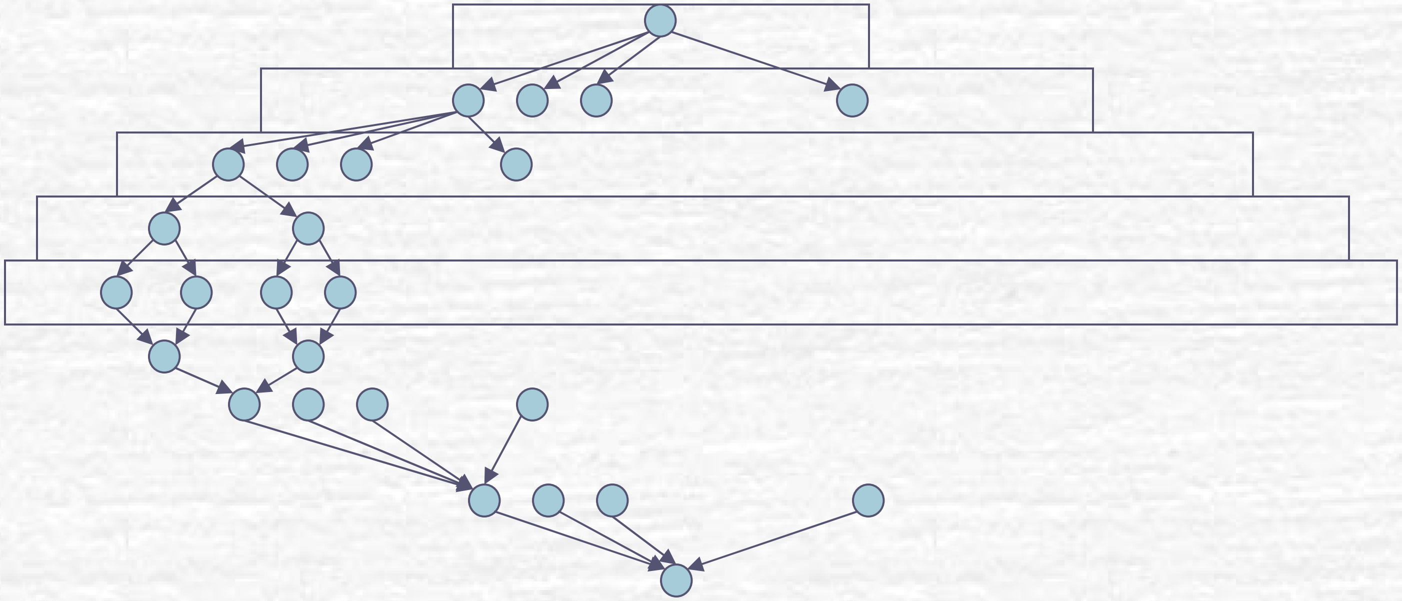
For any greedy schedule:

Efficiency = $\frac{W}{T_P} \geq \frac{PW}{W + D(P-1)}$

Parallel Time = $T_P \leq \frac{W}{P} + D$

# Breadth First Schedules

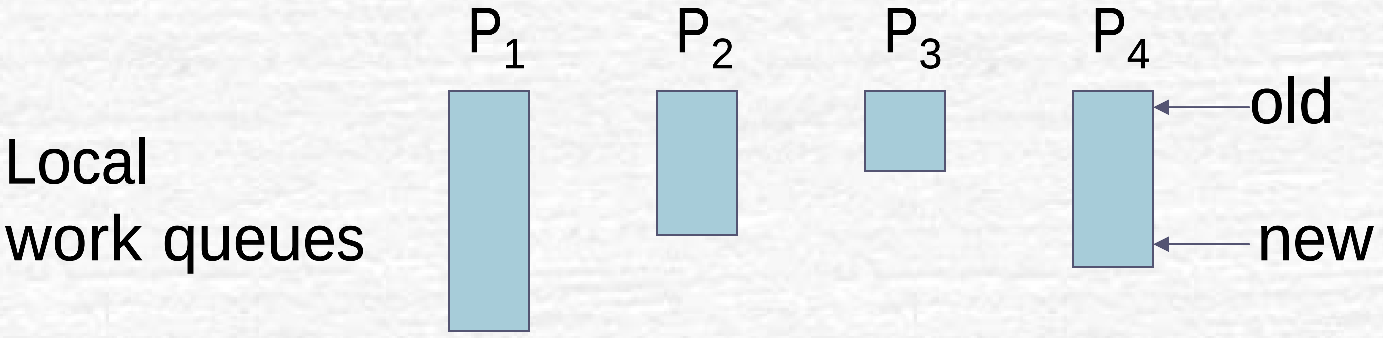
Most naïve schedule. Used by most implementations of P-threads.



$O(n^3)$ tasks

Bad space usage, bad locality

# Work Stealing

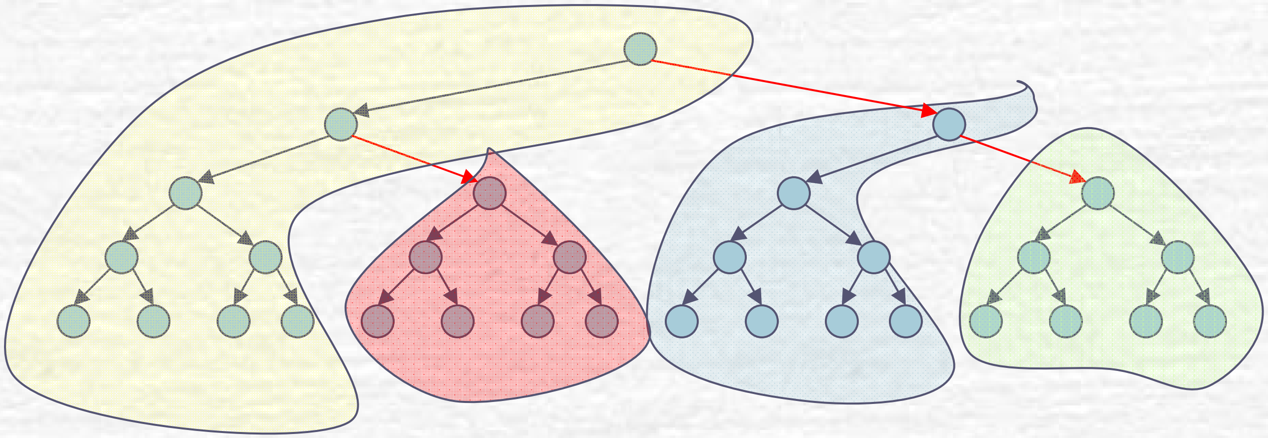


- push new jobs on “new” end
- pop jobs from “new” end
- If processor runs out of work, then “**steal”** from another “old” end

Each processor tends to execute a sequential part of the computation.

# Work Stealing

Tends to schedule “sequential blocks” of tasks



→ = steal

# Work Stealing Theory

For strict computations

Blumofe and Leiserson, 1999

- # of steals = O(PD)
- Space = $O(PS_1)$ $S_1$ is the sequential space

Acar, Blelloch and Blumofe, 2003

- # of cache misses on distributed caches

  $M_1 + O(CPD)$

  $M_1$ = sequential misses, C = cache size

# Work Stealing Practice

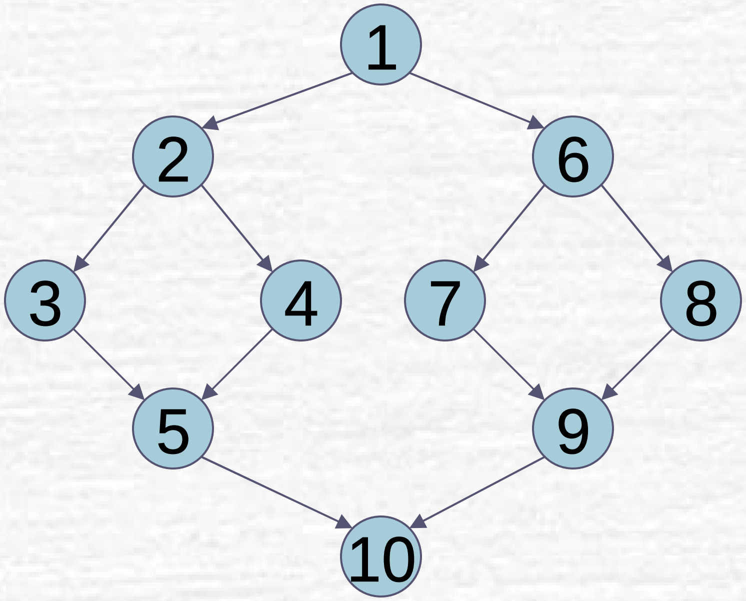
Used in Cilk Scheduler

- Small overheads because common case of pushing/popping from local queue can be made fast (with good data structures and compiler help).
- No contention on a global queue
- Has good distributed cache behavior
- Can indeed require $O(S_1P)$ memory

Used in X10 scheduler, and others

# Parallel Depth First Schedules (P-DFS)

List scheduling based on Depth-First ordering



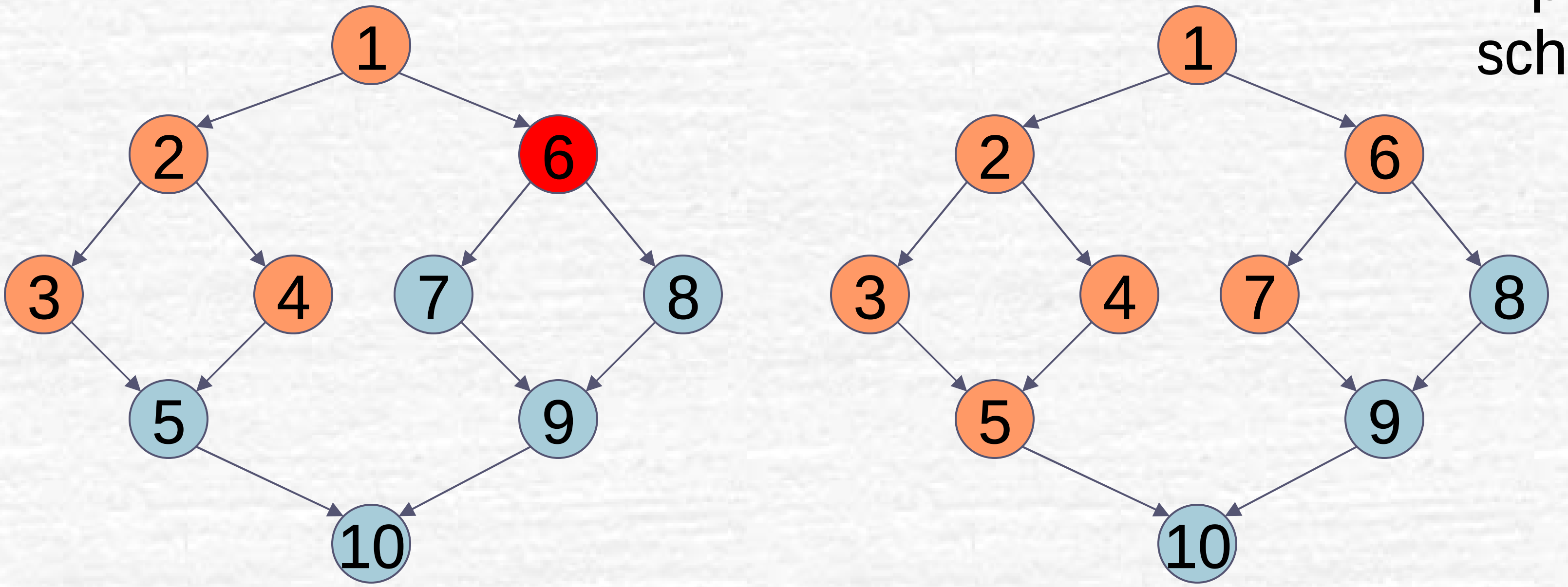
2 processor schedule

1
2, 6
3, 4
5, 7
8
9
10

For strict computations a shared stack implements a P-DFS

# “Premature task” in P-DFS

A running task is premature if there is an earlier sequential task that is not complete



2 processor schedule

1
2, 6
3, 4
5, 7
8
9
10

● = premature

# P-DFS Theory

Blelloch, Gibbons, Matias, 1999

For any computation:

- Premature nodes at any time = O(PD)
- Space = $S_1$ + O(PD)

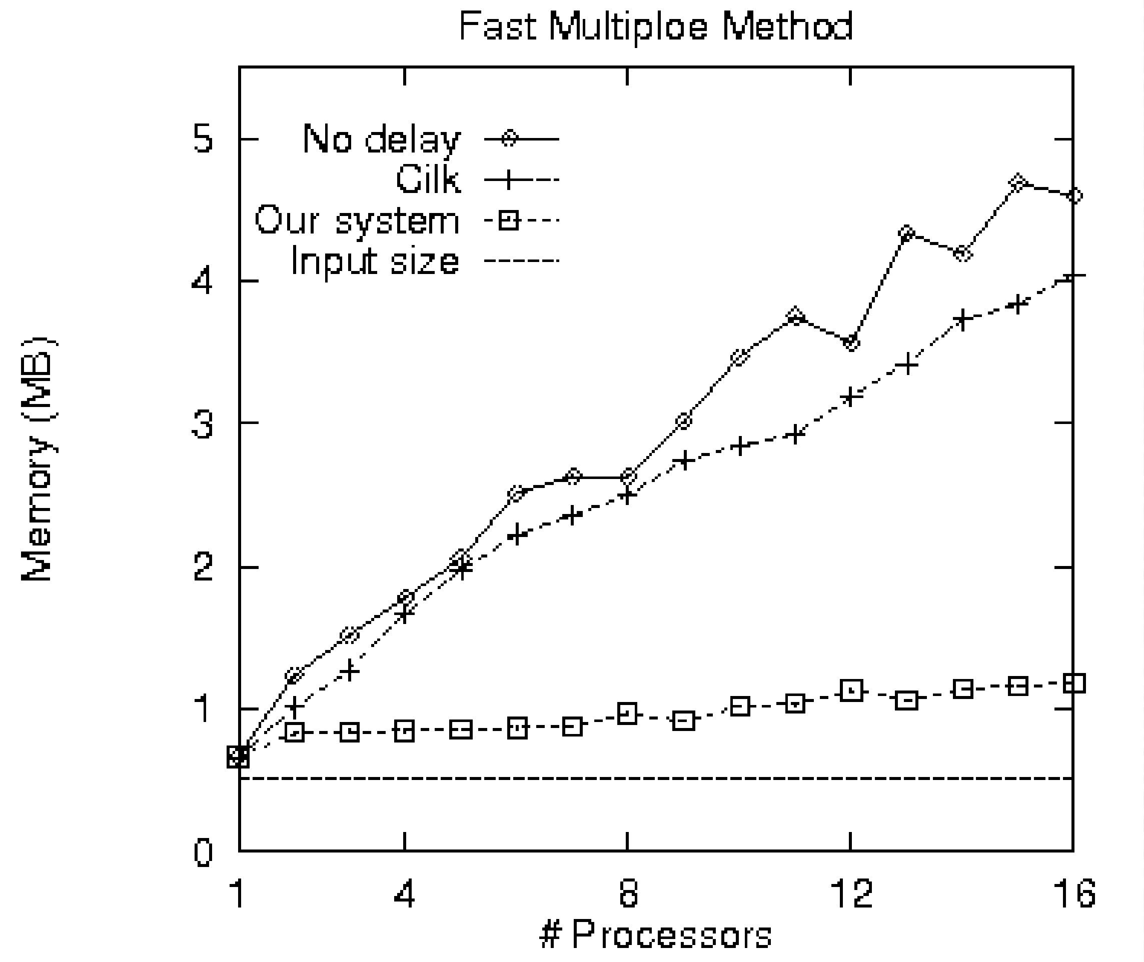
Blelloch and Gibbons, 2004

- With a shared cache of size $C_1$ + O(PD) we have $M_p = M_1$

# P-DFS Practice

- Experimentally uses less memory than work stealing and performs better on a shared cache.
- Requires some “coarsening” to reduce overheads

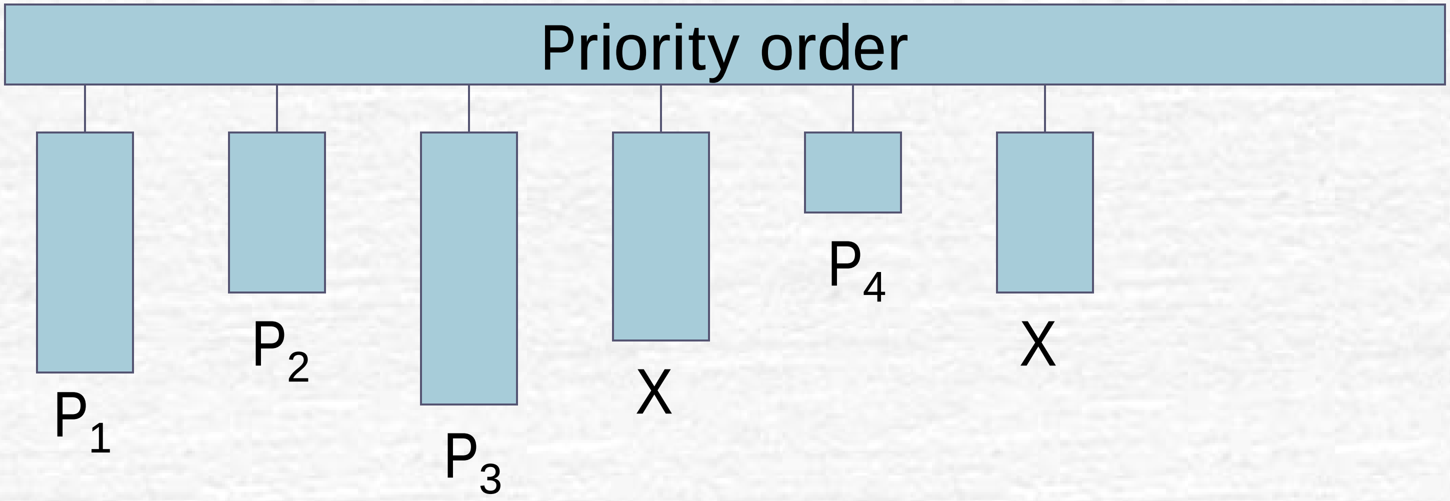
# P-DFS Practice



# Hybrid Scheduling

Can mix Work Stealing and P-DFS

Narlikar, 2002



Gives a way to do automatic coarsening while still getting space benefits of PDF

Also allows suspending a whole Q

# Other Scheduling

Various other techniques, but not much theory

e.g.

- Locality guided work stealing
- Affinity guided self-scheduling

Many techniques are for particular form of parallelism

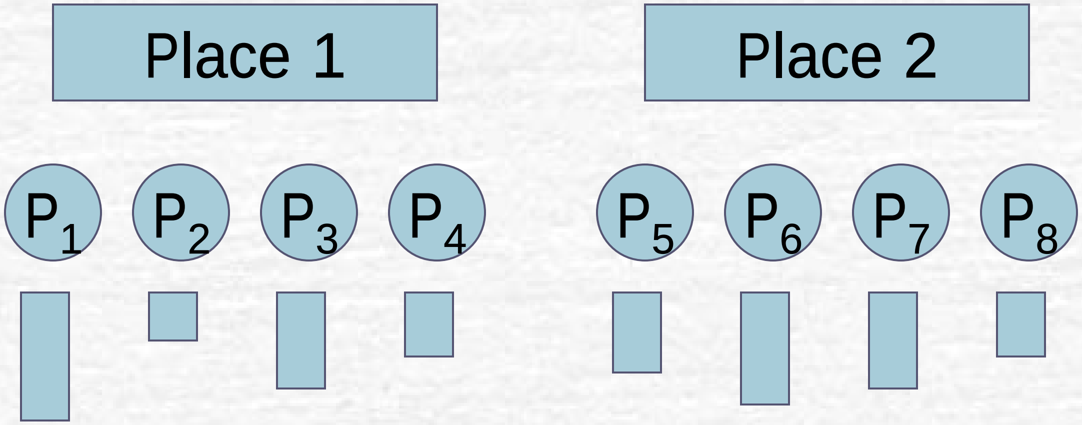
# Where to Now

X10 introduces many interesting new problems in scheduling

- Places
- Asynchronous statements at other places
- Futures (allows blocking of local activities)
- Clocks – generalization of bulk synchronous model
- Atomic sections
- Conditional atomic sections
- Exceptions

Clean design of X10 makes these issues reasonable

# Places

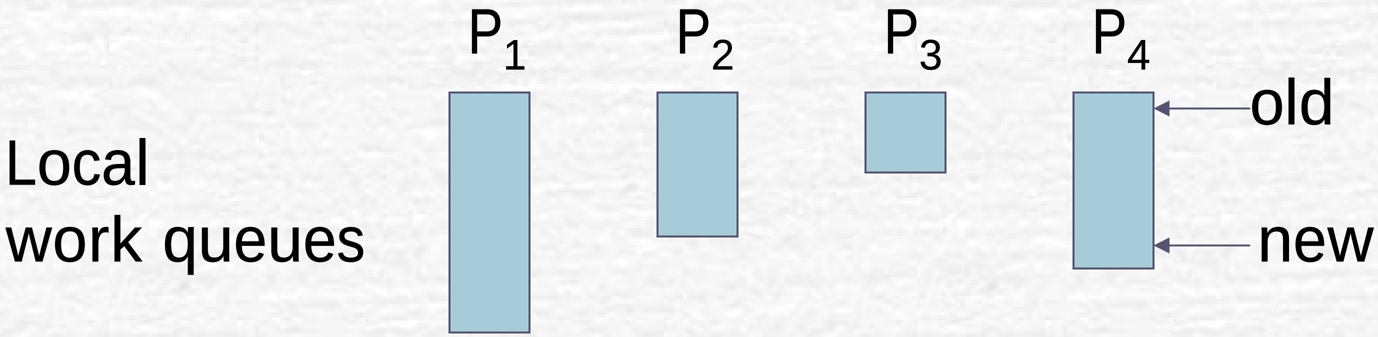


Some issues:

- Could be many more places than nodes
- Can you steal from another place?
- Do places on the same node share the task queues?
- Can one show any interesting theoretical properties

# Suspension

In X10 suspension can be caused by **atomic**, **futures**, **when** and by **clocks**. None of these are present in Cilk.



**NOT WELL STUDIED. e.g.**

When you wake up a suspension, where does it go?

When you suspend, do you continue on your own queue?

# Conclusions

1. Parallel computing is all about scheduling.
2. Theory matches practice reasonably well
3. **Many** open questions in both theory and practice
4. Even existing results in scheduling are not widely understood