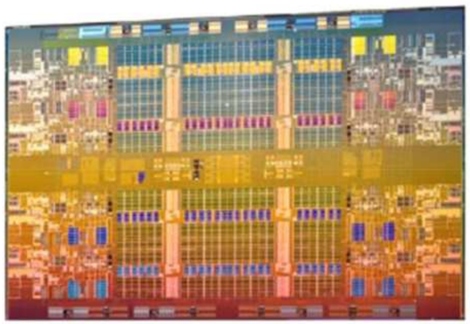
Efficient Primitives for Creating and Scheduling Parallel Computations

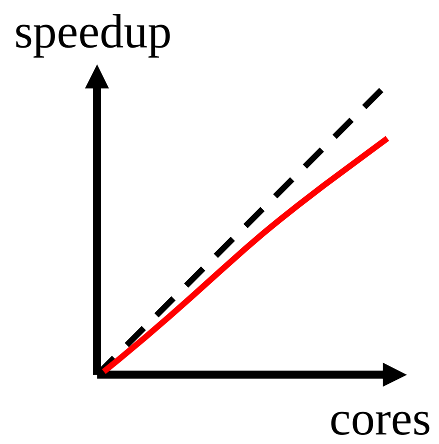
Umut Acar **Arthur Charguéraud** Mike Rainey

Max Planck Institute for Software Systems

DAMP'12 Philadelphia, 2012/01/28

Programming efficient parallel algorithm on multicore architectures





want to leave implicit:

– thread creation

– dynamic load balancing

– synchronization on joins

Constructs for implicit parallelism:

– fork-join

– sync-spawn

– parallel loops

– map-reduce

– graph traversal

– futures

– Invoke, ContinueWith, ContinueWhenAll, WaitAll,
Nested Tasks, Child Tasks (Microsoft's TPL interface)

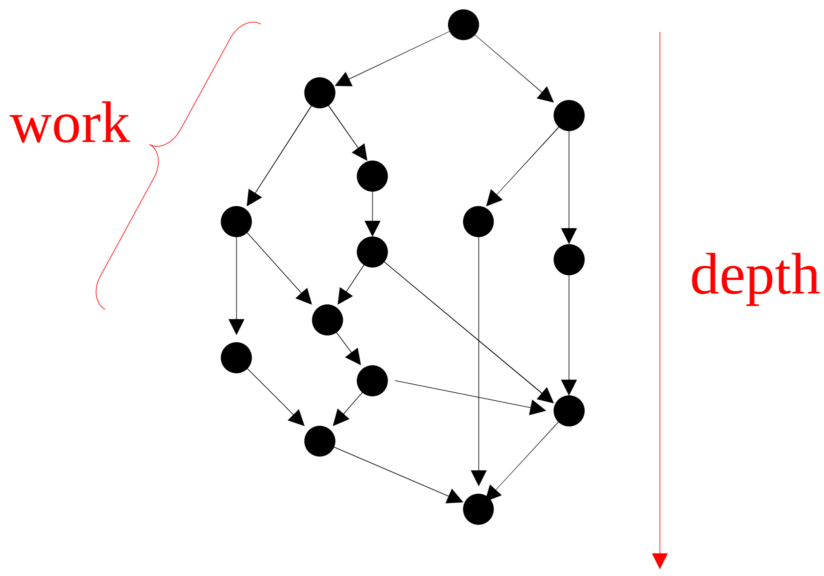
Too many constructs! $\rightarrow$ high cost of entry

Still not enough! $\rightarrow$ doesn't seem complete

What are the fundamental constructs for implicit parallel programming?

Can we find a concise interface that
– generalizes existing constructs,
– lets us express any implicitly parallel program,
– lends itself to efficient implementations?

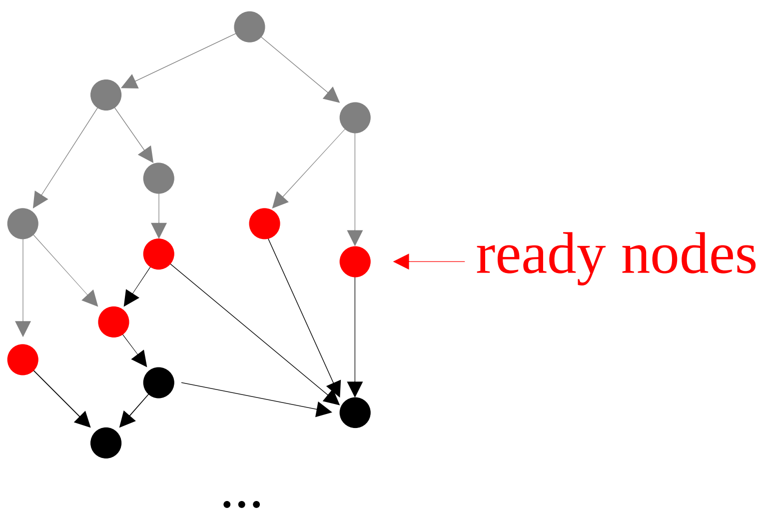
We can view parallel computations as DAGs to analyse their efficiency



Couldn't we <u>program</u> parallel computations directly as dynamic DAGs?

Towards a dynamic DAG programming interface...

```
node*  add_node (closure*)
void   add_edge (node*, node*)
```

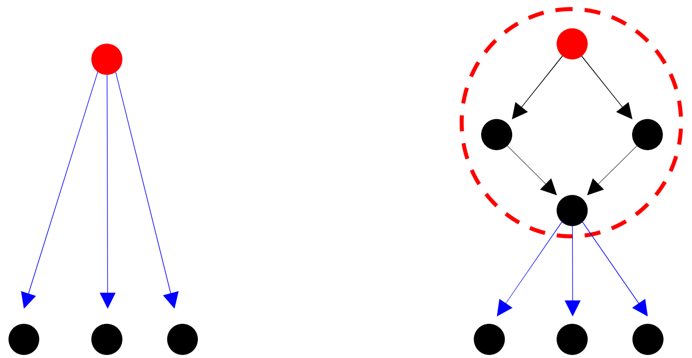


→ we here assume tasks to perform side-effects but not to return a value

→ need to be able to allocate a node before adding it to the DAG

node*	**create_node**	**(closure*)**
void	**add_node**	**(node*)**

→ need to be able to replace a node with a sub-DAG



void transfer_outedges_to (node*)

Our dynamic DAG programming interface:

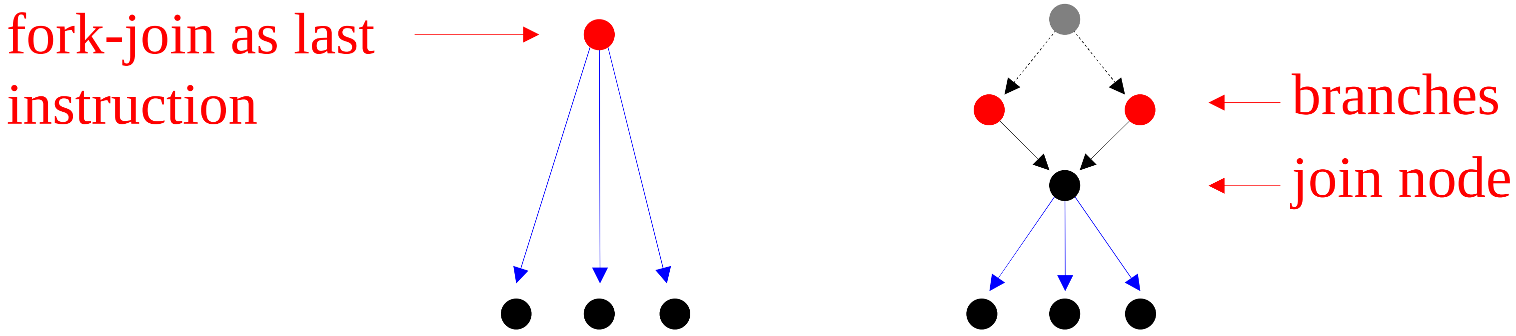
```
node* create_node (closure*)
void  add_node (node*)
void  add_edge (node*, node*)
void  transfer_outedges_to (node*)
```

Rest of the talk:

→ Expressiveness

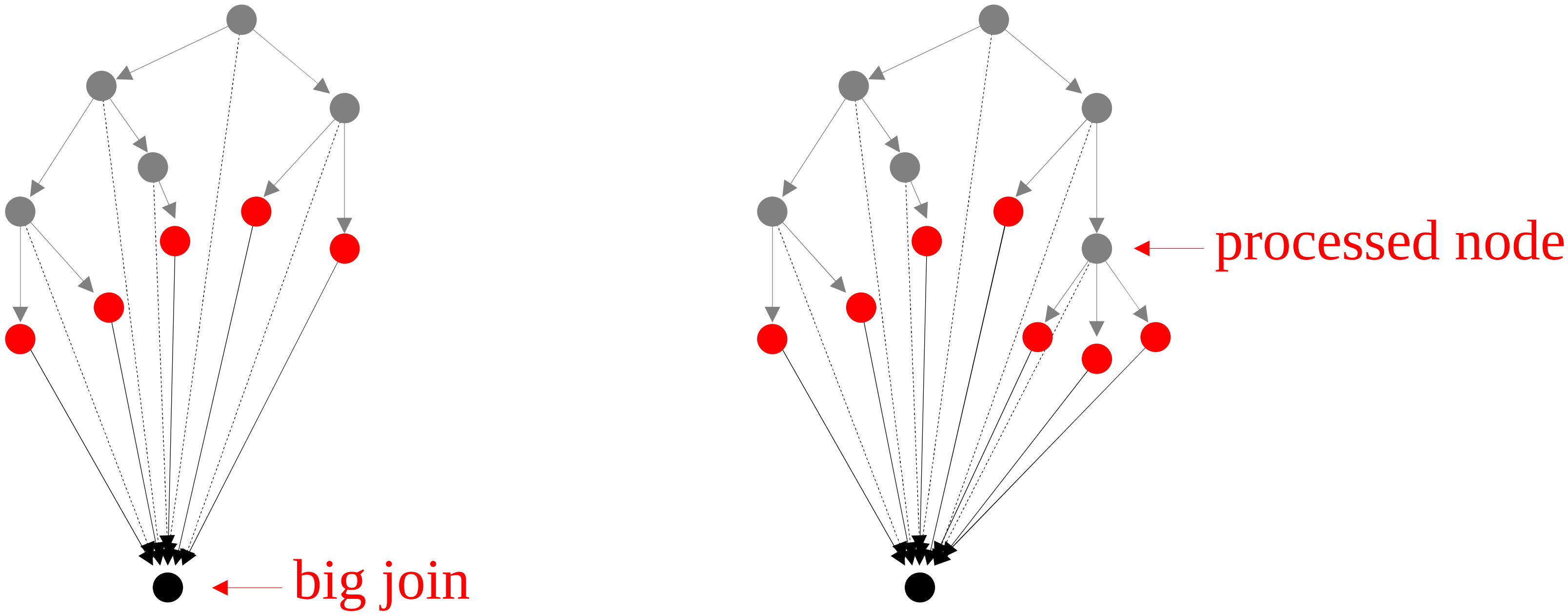
→ Efficiency

Encoding fork-join

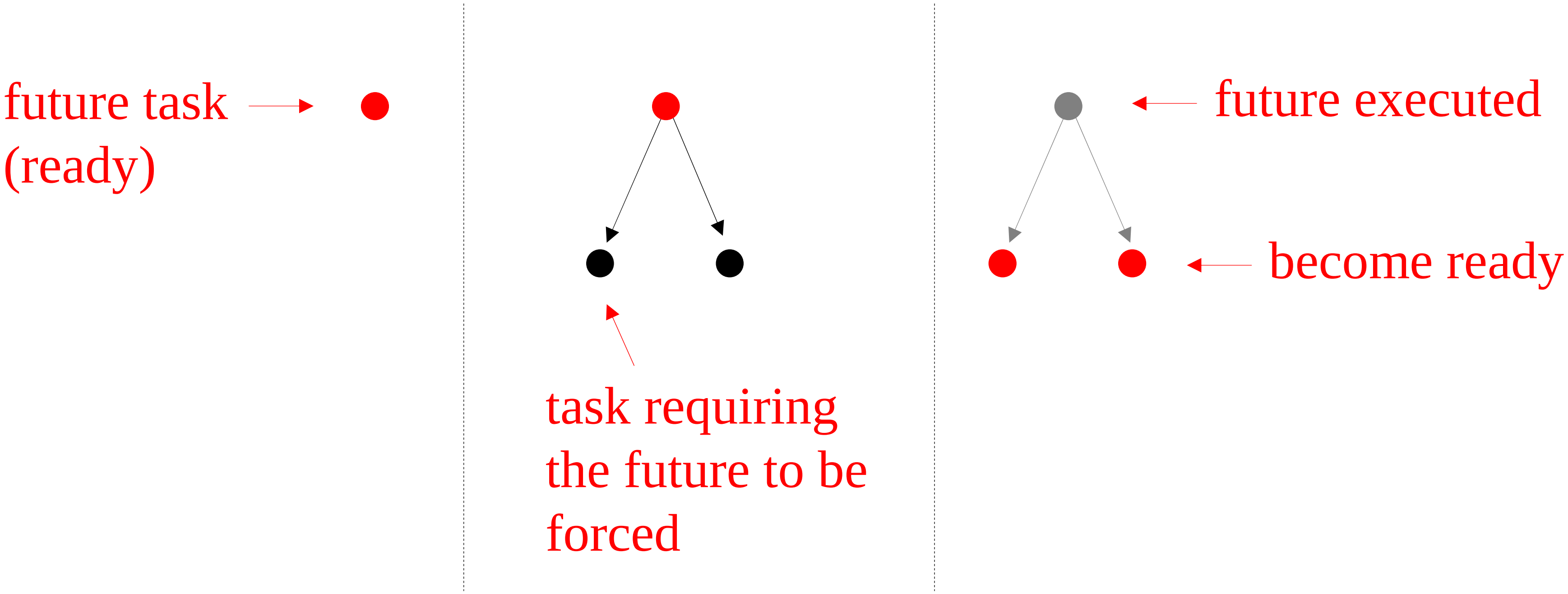



```
void fork_join(closure* c1, closure* c2, closure* cj)
   node* n1 = create_node(c1)
   node* n2 = create_node(c2)
   node* nj = create_node(cj)
   transfer_outedges_to(nj)
   add_edge(n1,nj)
   add_edge(n2,nj)
   add_node(n1)
   add_node(n2)
   add_node(nj)
```

Encoding graph traversal using a big join



Encoding futures



Note: a *lazy* future becomes ready only after first out-edge is added

The dynamic DAG interface is simple and expressive, but...

Can we schedule dynamic DAGs efficiently?

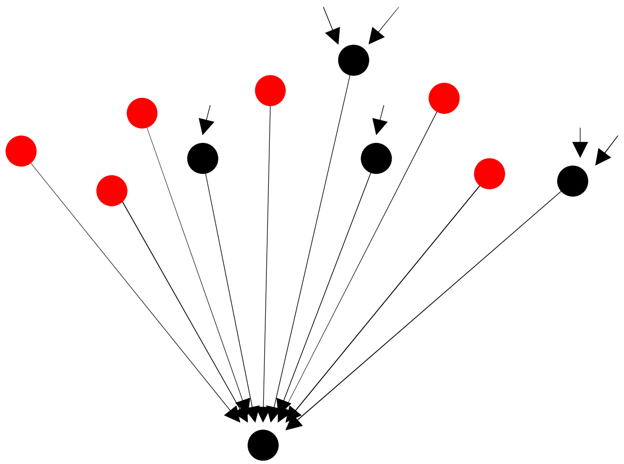
Three key ingredients

1) Load balancing

→ assume some variant of work stealing

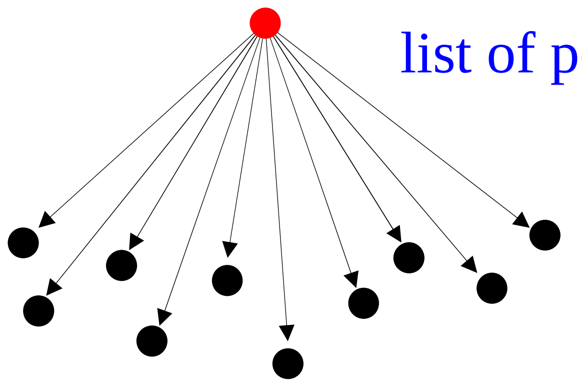
2) Number of incoming edges

→ a.k.a. join counters



3) List of outgoing edges

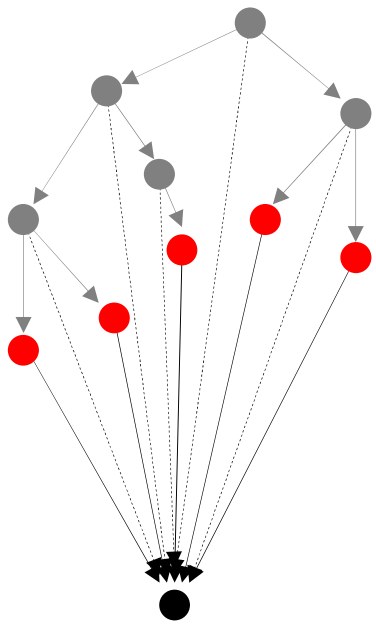
→ dual problem (see paper)



Big-arity joins: distributed counters

– use one counter per processor (# edges added - # edges removed)

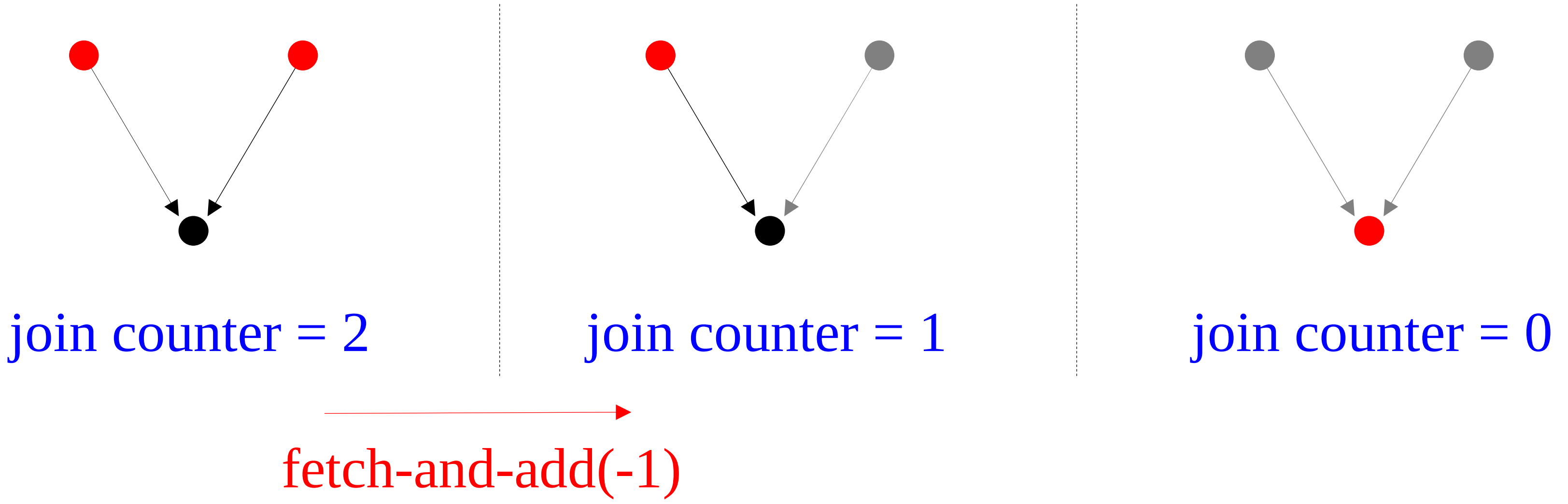
– periodic check by one particular processor to see if the sum is zero



owner = processor #4

counters = [23; -9; 97; 67; 20]

Small arity joins: atomic counters



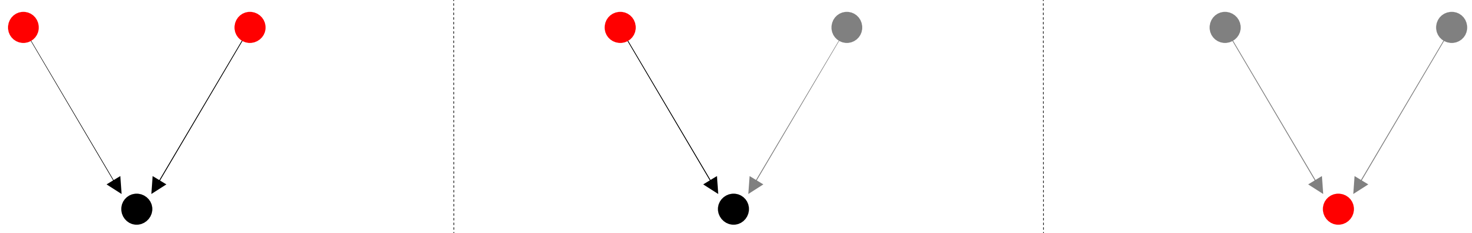
Can we avoid synchronization?

→ clone translation supports fork-join but not arbitrary DAGs

Small arity joins: owner counters

– one owner for each task, in charge of updating the counter

– other processors send messages over producer-consumer buffers



owner = proc X
owner counter = 2

owner = proc X
owner counter = 1

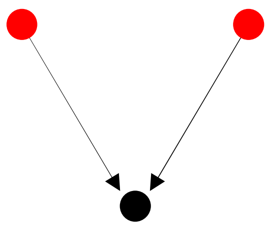
owner = proc X
owner counter = 0

$\rightarrow$ but delays can be incurred

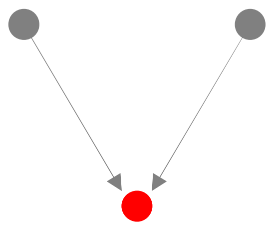
Small arity joins: optimistic counters

→ same as previous slide, plus a shared counter

→ works if no dynamic addition of incoming edges

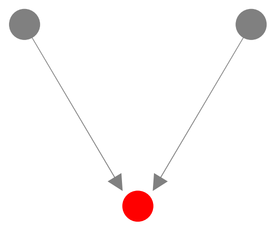


owner = proc X
owner counter = 2
shared counter = 2



owner = proc X
owner counter = 0
shared counter = 0

<u>No race</u>: the shared counter reaches zero



owner = proc X
owner counter = 0
shared counter = 1

<u>Race</u>: messages are used to recover

Representation of edges on a per-node basis

– an instrategy for representing the number of incoming edges

– an outstrategy for representing the list of outgoing edges

node* create_node (closure*, instrategy*, outstrategy*)

Examples of in-strategies:

– distributed

– atomically-updated

– owner-based

– optimistic

Dynamic DAGs, with per-node specification of edges representation

```
node*  create_node (closure*, instrategy*, outstrategy*)
void   add_node (node*)
void   add_edge (node*, node*)
void   transfer_outedges_to (node*)
```

→ concise, expressive, efficient interface

→ define and explain other constructs in terms of this interface

→ implemented in our C++ scheduler *"PASL"*

More details in our paper, available from the DAMP 2012 website