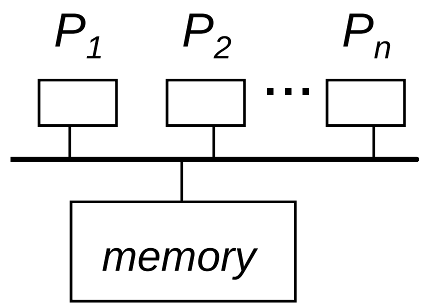
Space-Efficient Implementation of Nested Parallelism

Girija Narlikar and Guy Blelloch
SCANDAL project
School of Computer Science
Carnegie Mellon University

Why space-efficiency?

Machine model



Programming model

Dynamic, fine-grained parallelism (# threads >> # processors)

Too many active threads + "bad" execution order

$\longrightarrow$ *large space requirement* $\longrightarrow$ *smaller problem size*

Space-efficient scheduling algorithms

Previous results

- Heuristic-based [Multilisp, Id, etc]
- Provably efficient

  $S_p = p \times S_1$ [Cilk, Burton-Simpson]

  $S_p = S_1 + O(pD)$ [Blelloch-Gibbons-Matias]

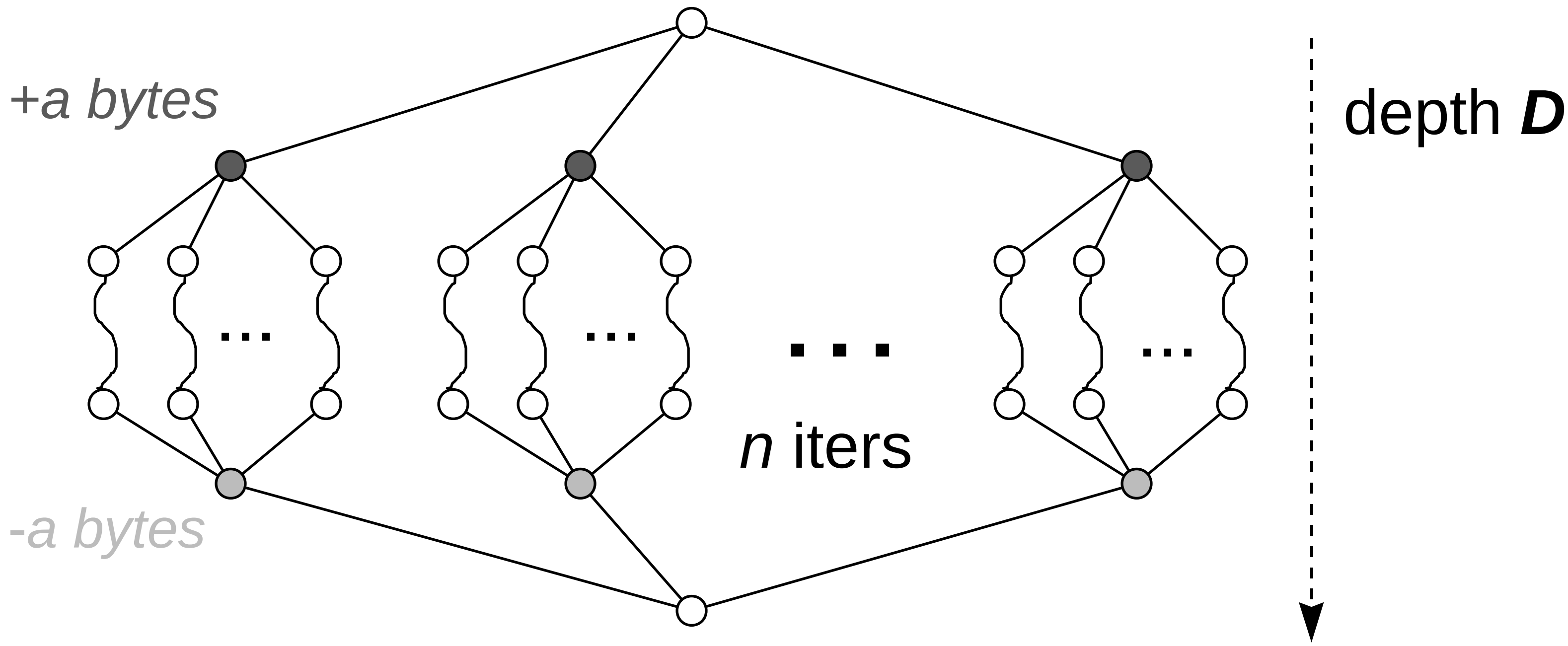
  D = Critical path length (depth)

Current result

- Provably space-efficient, asynchronous

  $S_p = S_1 + O(pD)$

Example



$S_1 = O(a)$

Previous approaches: $S_p = O(na)$ *or* $O(pa)$

Current approach: $S_p = O(a + pD)$

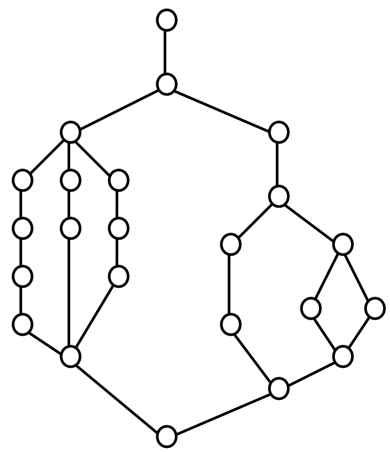
Basic concepts

- **Parallel depth-first order**. Schedule threads in an order as close as possible to the serial order.
- **Delayed allocations**. Delay big allocations to allow other threads to execute instead.
- **Minimal excess parallelism**. Fork only as many threads as required for good performance.
- **Uninterrupted execution**. Keep a thread running on the same processor until it completes, forks or allocates space.
- **Asynchronous scheduling**. Overlap scheduling with computation. Schedule more threads less often.

Programming model

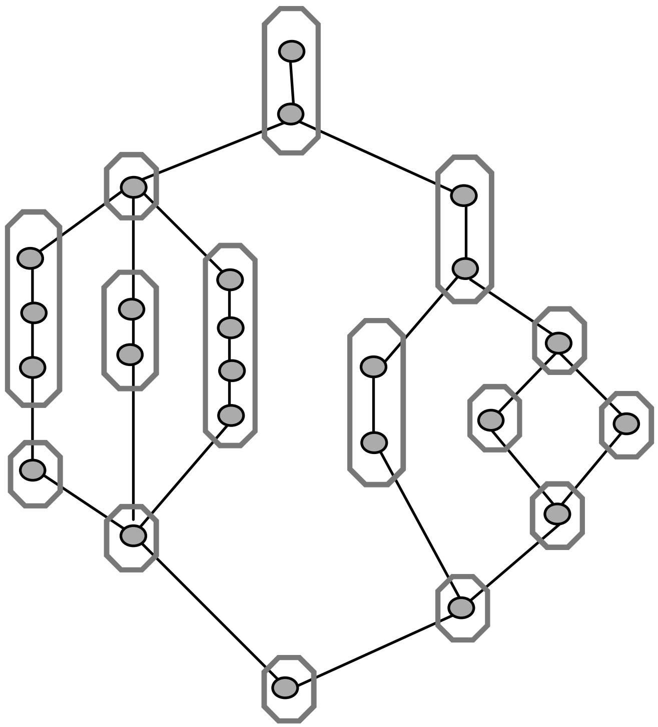
Dynamic, nested parallelism.

Series-parallel DAG.



Each node represents a unit computation that may (de-)allocate any number of bytes.

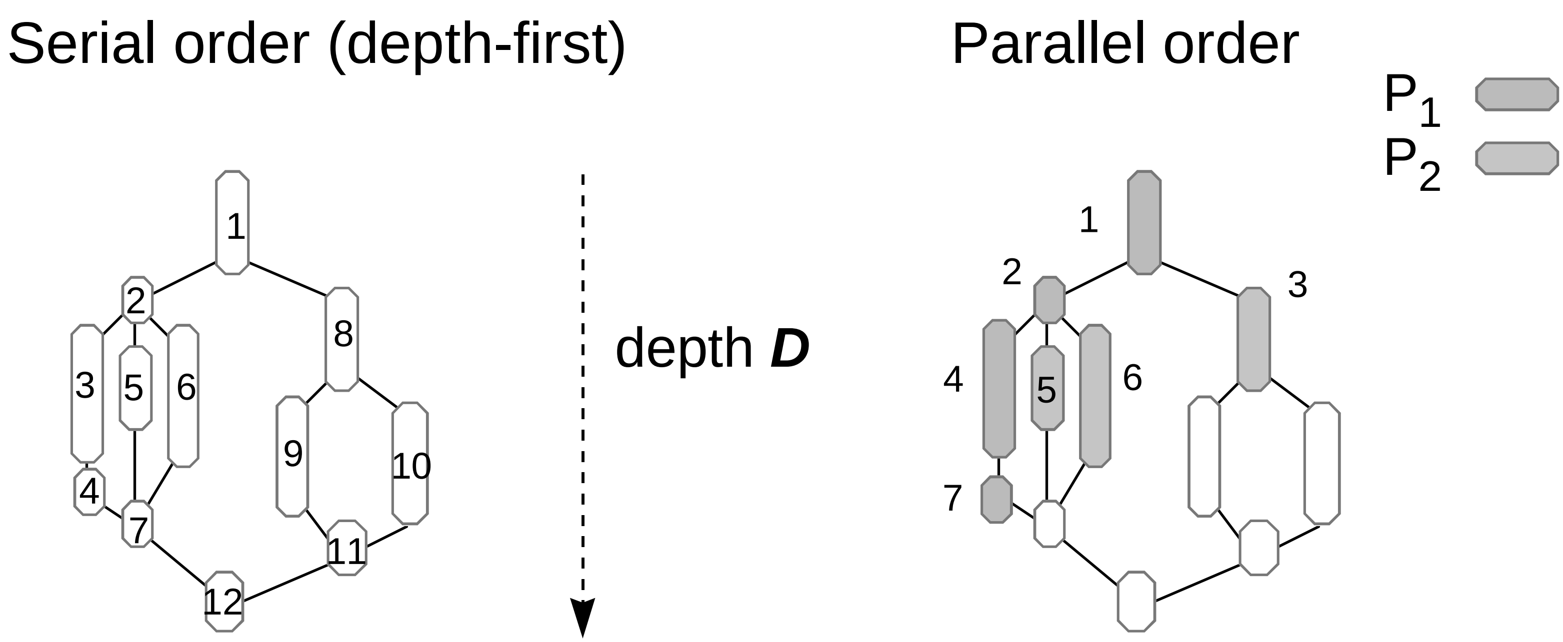
Grouping nodes into non-preemptive jobs



= non-preemptive job, using at most K space.

Computation graph is partitioned into jobs at runtime.

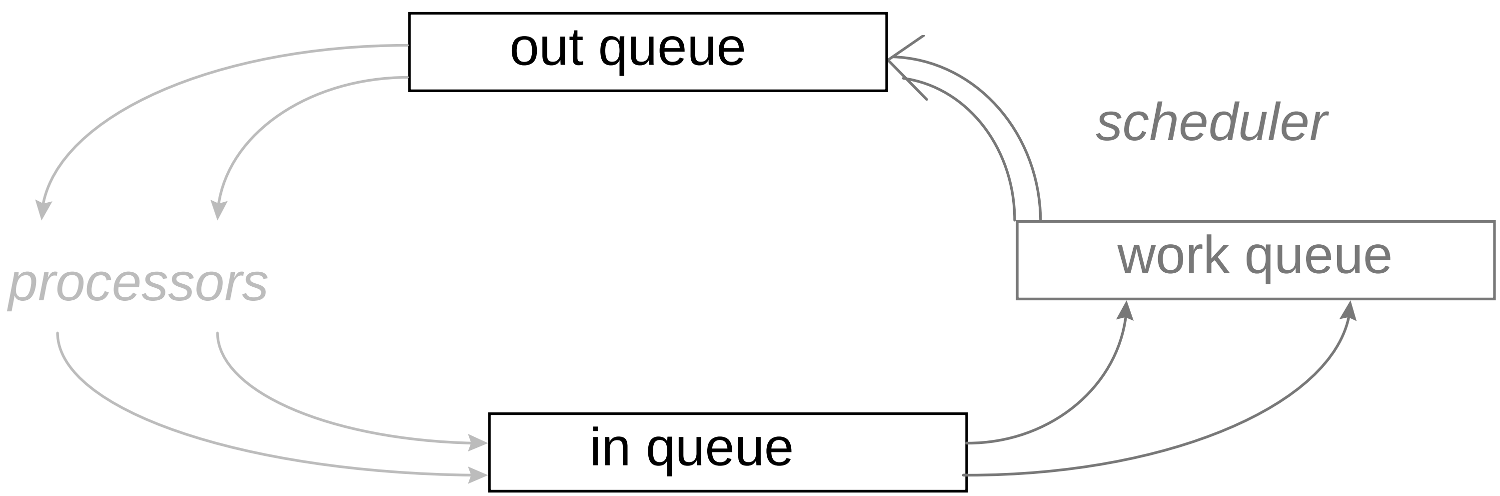
Parallel "depth-first" order



Maximum number of jobs executed "out of order" is *O(pD)*

$\longrightarrow S_p = S_1 + O(pD)$

Implementation



Work queue: priority queue of ready threads.

Platform: SGI Power Challenge SMP (bus based, shared memory).

Programs written in continuation-passing style.

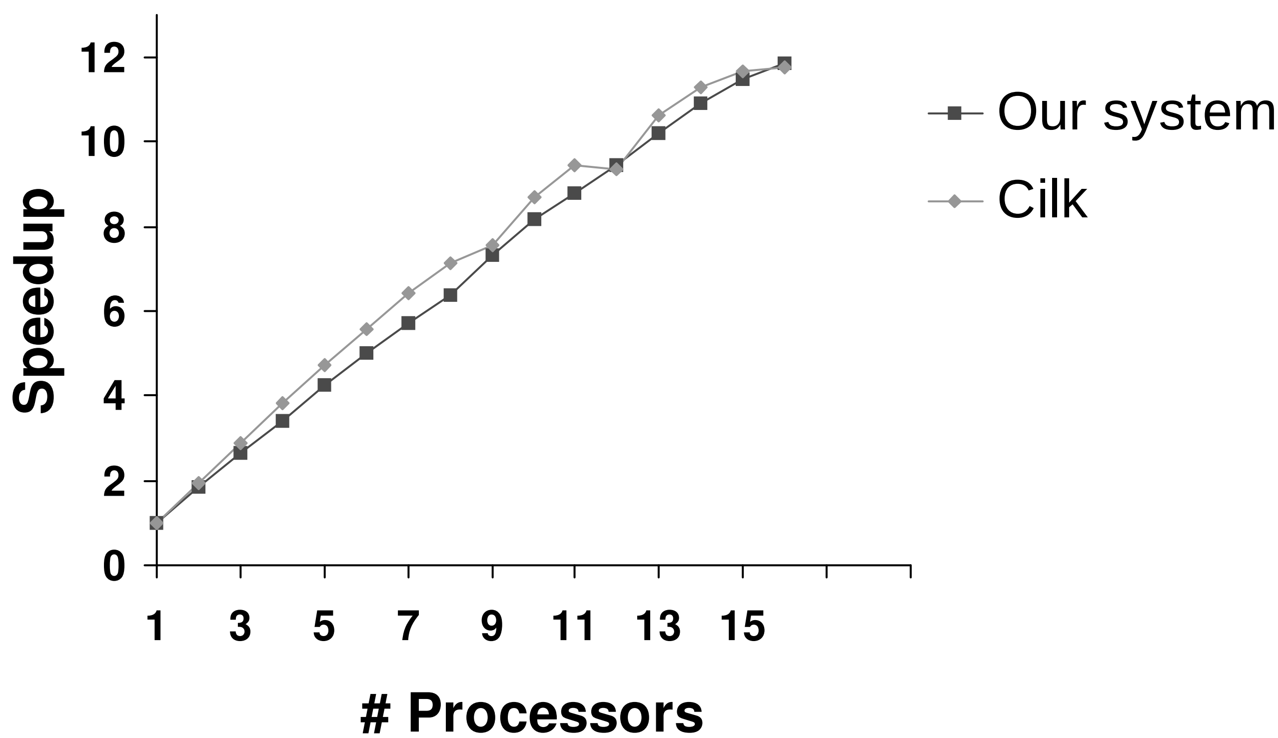
Evaluation

- Parallel benchmarks
  - Blocked recursive matrix multiply (RecMM)
  - Strassen's matrix multiply (StrMM)
  - Fast multipole method (FMM)
  - ID3 decision tree building (ID3)
  - Sparse matrix-vector multiply (MVM)
- Time and space performance compared with Cilk

  ($S_p = p \times S_1$)

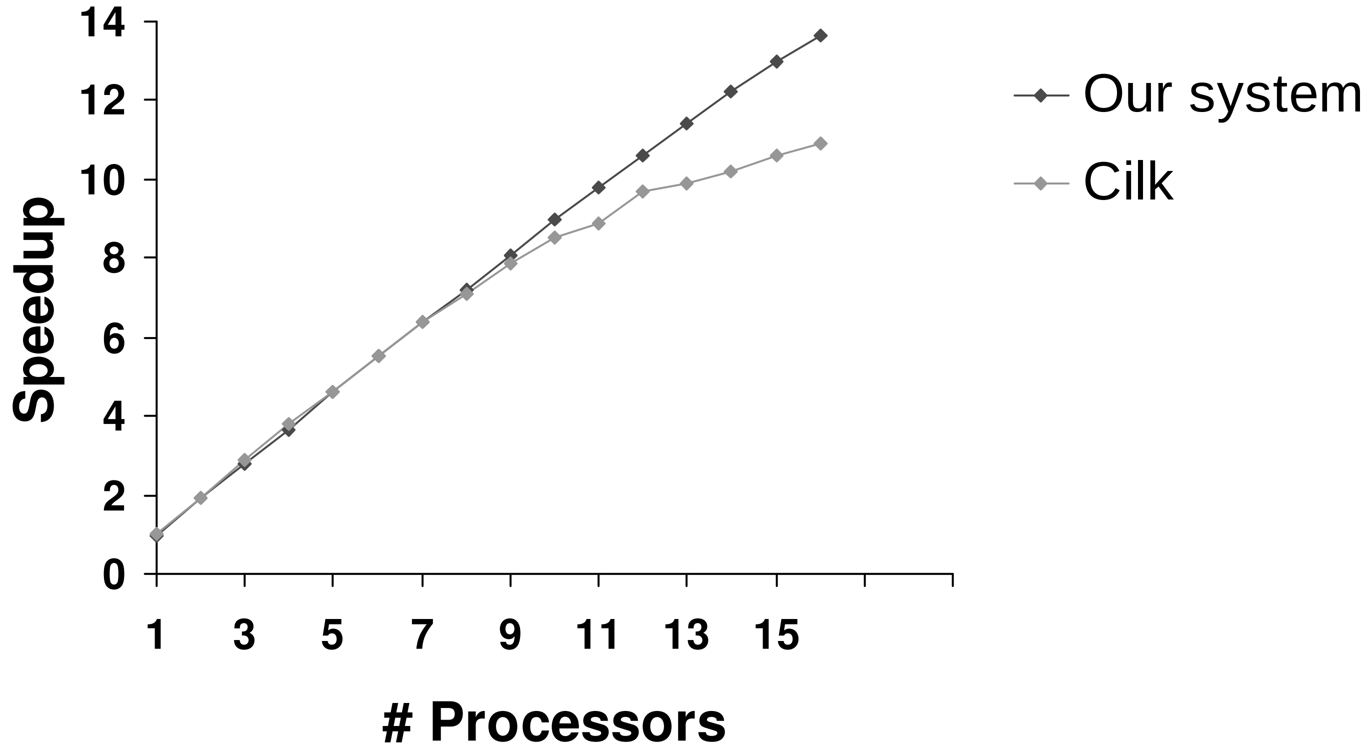
Time performance

Recursive Matrix Multiply *(1024x1024)*

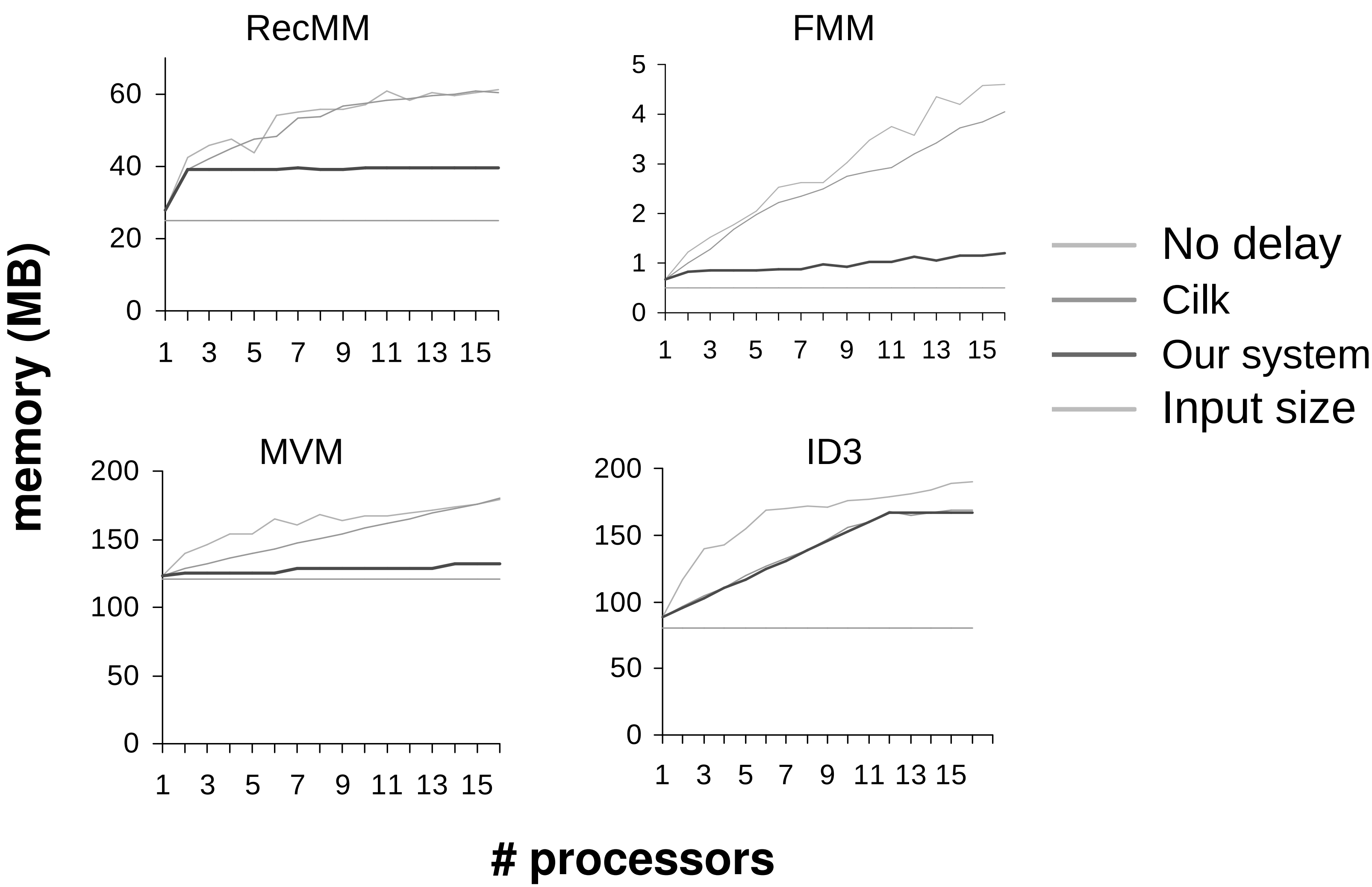


Time performance

Fast Multipole Method *(4 levels, mp = 5)*

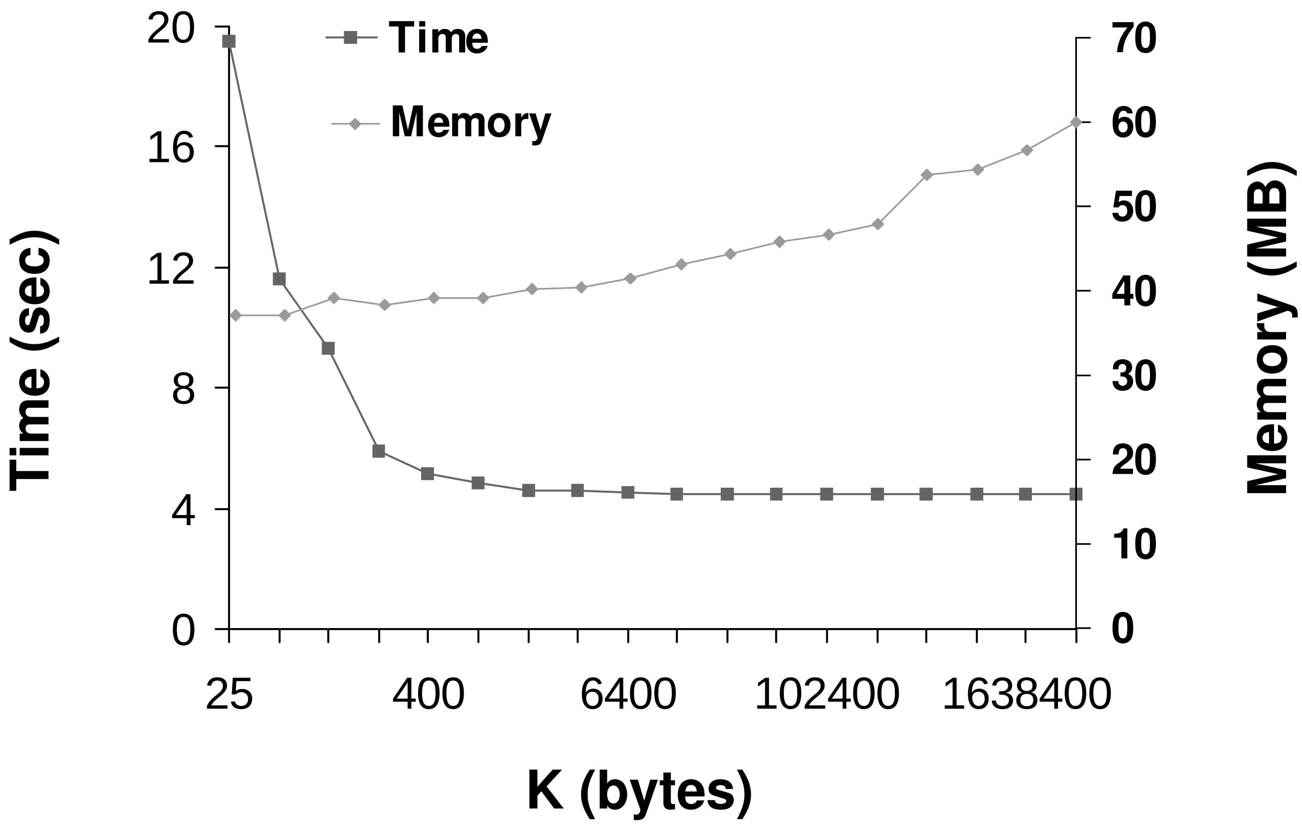


Space performance



Space and time trade-off

Recursive MM *(1024 x 1024, p = 8)*

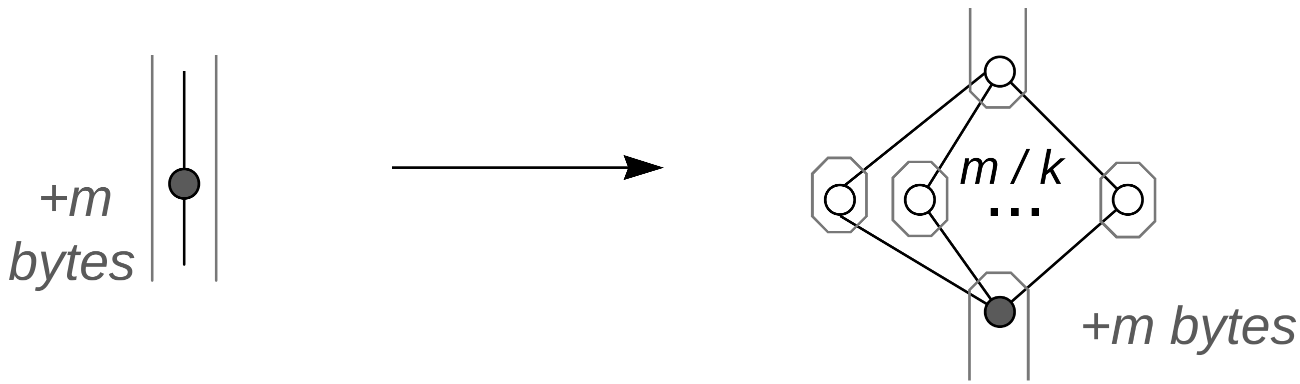


Summary

- Asynchronous scheduling algorithm with provably good space and time bounds.
- Good space and time performance in practice for parallel benchmarks on an SMP.
- Adjustable trade-off between space and time.

Handling big allocations

For every node that allocates *m* space (*m* > *K)*, insert *m/K* parallel "dummy" threads before it.

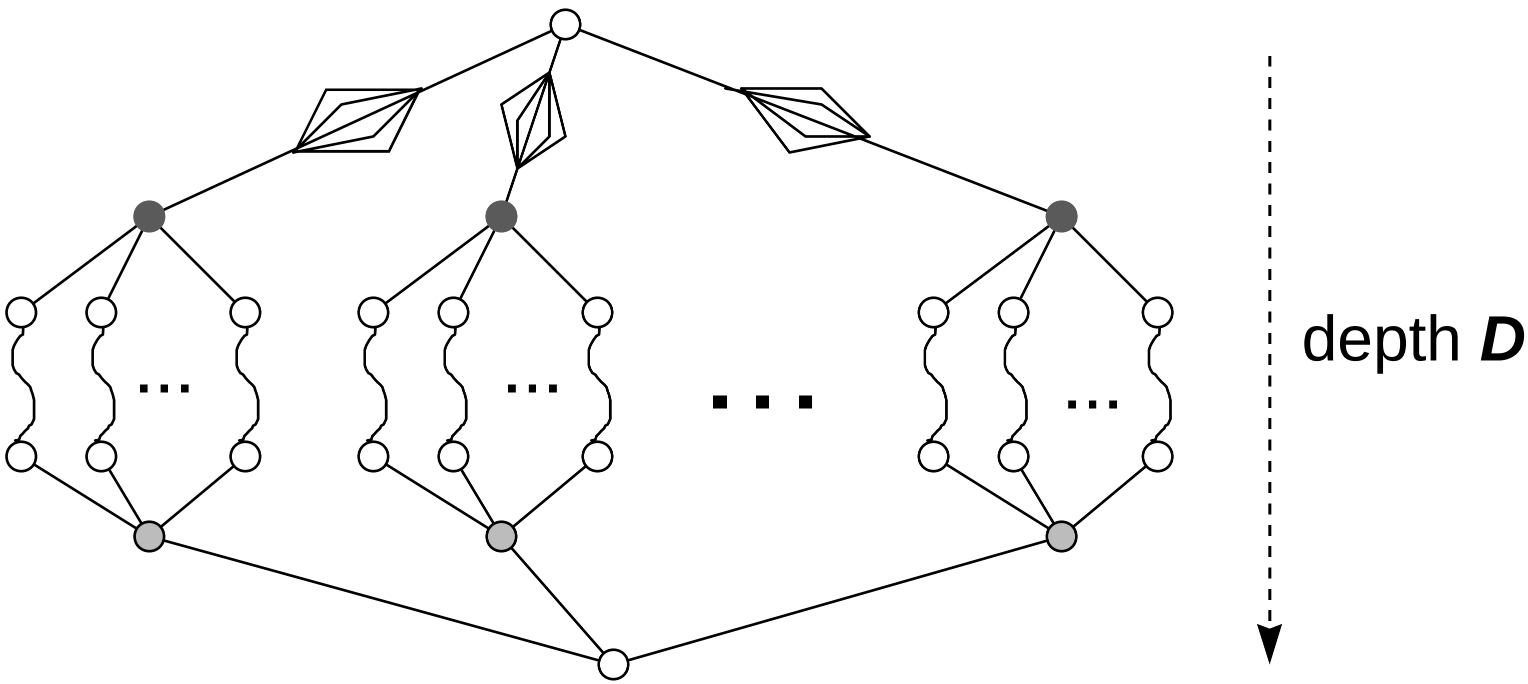


$\#\ nodes = W + S_a / K$ [S_a = sum of allocations > K]

$T_p = O(\#\ nodes / p + D) = \mathrm{O}(W / p + S_a / Kp + D)$

When $S_a = O(W)$, $\boldsymbol{T_p = O(W / p + D)}$

Example



$S_1 = O(m)$

In previous systems, $S_p = O(pm)$

In our system, $S_p = O(m + pD)$