Algorithms for Memory Hierarchies
Lecture 14

Lecturer: Nodari Sitchinava
Scribe: Michael Hamann

1 Parallelism and Cache Obliviousness

The combination of parallelism and cache obliviousness is an ongoing topic of research, in this lecture we will only learn to know a few basics.

In the past lectures we looked at parallel algorithms in which the memory and block size were known while in this lecture we will look at parallel algorithms in which the memory/cache size are unknown. This has many advantages like that the algorithms need to be designed only once and then work in different kinds of setups. Todays processor and system architectures get increasingly complex with several levels of partially shared and partially private caches. Designing cache-aware algorithms for these architectures becomes increasingly difficult while cache-oblivious algorithms also work on systems with several levels of memory.

Dealing with cache hierarchies in parallel algorithms is a lot more complex than dealing with cache hierarchies in sequential algorithms as the caches can be used by any processor that shares the cache. Furthermore while in sequential algorithms it is clear that data that has been loaded at one point in the algorithm and is still in the cache at a later point in time can still be used while in parallel algorithms the data might be used by other processors that don't share that cache. Thus the data could have been modified in other caches and might need to be updated. For the very same reason multiple processors using the same data in a very short period of time can cause problems in parallel algorithms.

Not knowing the block size poses an additional challenge when data that is stored in the same block could be used by different processors. In this case the access would need to be synchronized but as the block size is unknown it is also unknown if there is actually any synchronization needed.

There are two ways to design a parallel algorithm:

Coarse-grain parallelism One thread per physical core. The threads are provided by the operating system.

Fine-grain parallelism Many light-weight user level threads. The program exposes lots of parallelism, the smallest task should be as small as possible. The runtime system then schedules these tasks on physical cores. An example for such a system is CILK++.

1.1 Work-depth framework

The work-depth framework has been initially introduced for PRAM models. In the work-depth framework the algorithm specifies the smallest possible tasks that can be executed concurrently.

The work W is defined as the number of such tasks while the depth D is the number of (parallel) steps.

The computation of an algorithm can be modeled as a DAG (directed acyclic graph) whose nodes represent the tasks and whose edges represent the dependencies between the tasks. The work W is then the running time of the algorithm in sequential and the depth D is the longest path in the graph. The properties of the graph that we need to analyze are thus the number of nodes and the longest path.

With Brent's theorem we get the following inequality for the parallel execution time T_P with P processors:

$$T_P \leq \frac{W}{P} + D$$

When the number of processors is smaller than the number of tasks that can be executed in parallel one physical processors simulates more than one virtual processor.

1.2 Scheduling tasks on physical processors

For just one processor any topological order on the DAG is possible. One possibility that is shown in the graph in Figure 1 is a depth-first scheduler (1-DF - scheduler) that follows each path in the graph as long as possible (i.e. until it encounters a tasks that hasn't all requirements fulfilled).

For two processors, we'll have a look at three possibilities:

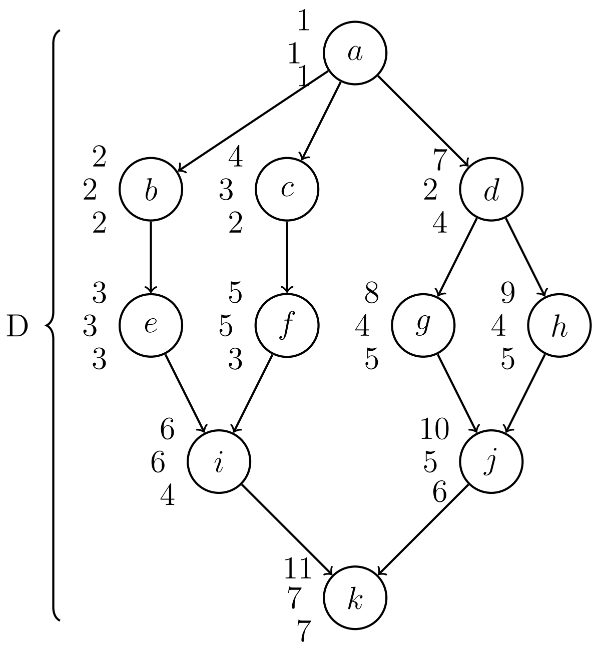


Figure 1: 1-DF-schedule; greedy schedule and PDF schedule for 2 processors

Greedy scheduler The greedy scheduler assigns "free" tasks (tasks that haven't been executed yet but whose predecessors have already been executed) to any available processor, an example is shown in Figure 1. A greedy scheduler will always find a schedule with at maximum twice as many steps as the optimal schedule would need.

Prioritized scheduler A prioritized scheduler assigns the "free" tasks according to a priority, for example the PDF-scheduler (parallel depth-first scheduler) is a greedy scheduler that assigns tasks prioritized by a 1-DF order, an example is shown in Figure 1.

Work stealing scheduler A work-stealing scheduler is for example implemented in CILK++. The algorithm of the work-stealing scheduler can be found in Algorithm 1, an example of its execution in Figure 2.

Algorithm 1: The work stealing scheduler

```
foreach processor do
    if my deque is non-empty then
        pop first task and execute;
    else
        steal task from a random processor's deque's end;
    push newly created "free" tasks on my deque;
```

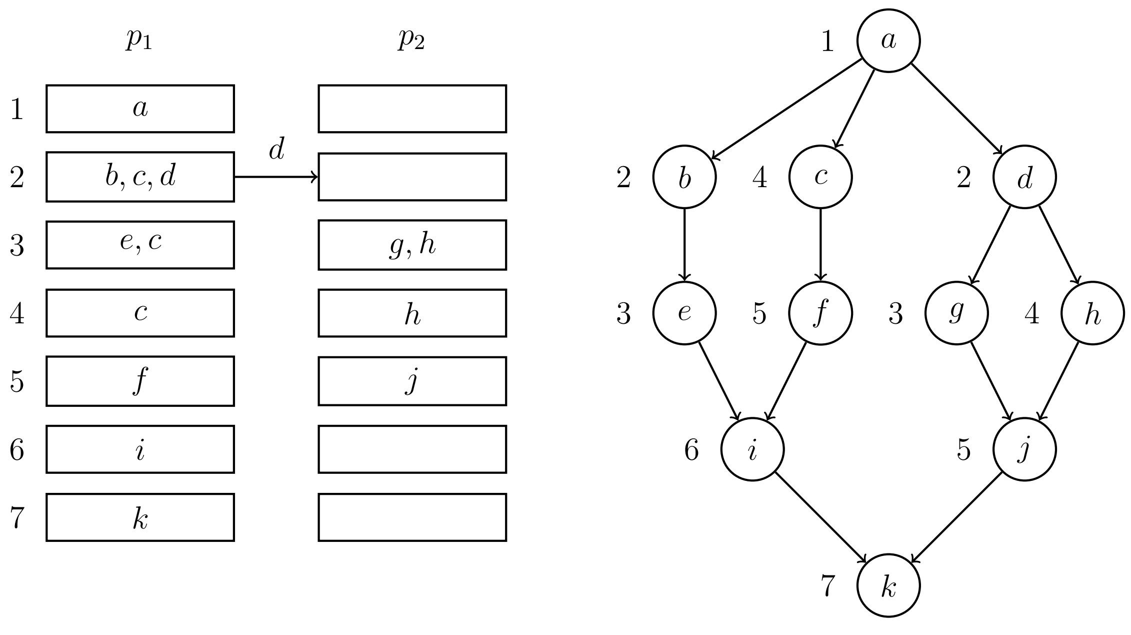


Figure 2: Example execution of the work stealing scheduler

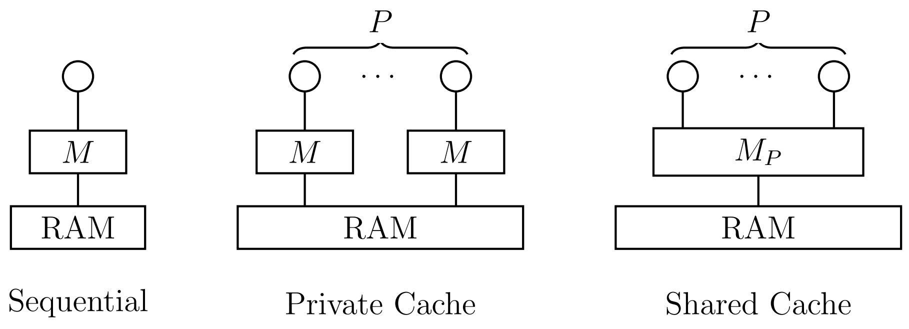


Figure 3: The different cache setups we consider in this lecture

In the sequential cache-oblivious setting it was the paging algorithm that knew the exact parameters of the system. In the parallel setting the scheduler replaces the role of the paging algorithm.

A graph is series-parallel if it can be constructed of series-parallel graphs (with a simple edge with two nodes as the smallest series-parallel graph) by serial or parallel composition and has a source and a sink.

Theorem 1. *If the execution DAG is series-parallel of depth D, then the work stealing scheduler for P processors will perform $O(P \cdot D)$ steals in expectation.*

Proof. Intuition: Each set of $O(P)$ steals reduces the depth of the remaining DAG by at least one level. □

In the following we will look at two simplified scenarios for memory analysis: The I/O-complexity of schedulers for private and shared caches (see Figure 3).

Consider algorithm 2 (visualized in Figure 4).

Algorithm 2: Scheduler example algorithm

```
for parallel $i = 1, 2, \ldots, P$ do
    for $k = 1, \ldots, R$ do
        for $j = 1, \ldots, M$ do
            $x_i \leftarrow x_i + a_i[j]$ ;
```

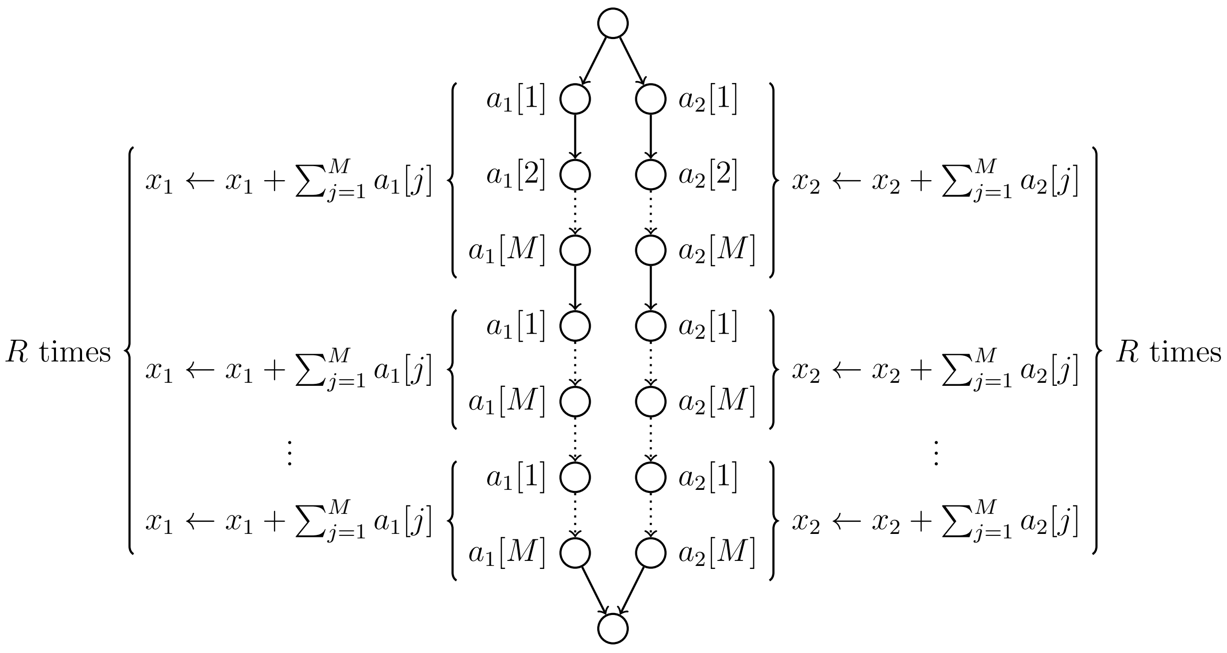
The sequential running time T_{seq} of algorithm 2 is $O(R \cdot M)$. The I/O-complexity when using a 1-DF-scheduler is $2 \cdot \frac{M}{B}$ I/O's.

With a parallel schedule for two processors with the PDF scheduler on a private cache there are $2 \cdot \frac{M}{B}$ I/O's in total. On a shared cache of size $M_P = M$ we get $2 \cdot R\frac{M}{B}$ I/O's while when $M_P = 2M$ there are only $2 \cdot \frac{M}{B}$ I/O's.

Theorem 2. *The PDF scheduler incurs Q_1 I/O's on a shared cache if $M_P = M + P \cdot D$ where Q_1 is the sequential I/O complexity of the solution.*

Theorem 3. *The WS scheduler incurs $O(Q_1)$ I/O's on a shared cache if $M_P = P \cdot M$.*

On a current desktop machine with $P = 8$ processors for $D = O(\log n) \approx 32$ $P \cdot D = 256$ which means that with the PDF scheduler we only need 256 additional words of memory for the parallel solution.



$a_1[1]$
$a_2[1]$
$a_1[2]$
$a_2[2]$
$a_1[M]$
$a_2[M]$
$x_1 \leftarrow x_1 + \sum_{j=1}^{M} a_1[j]$
$x_2 \leftarrow x_2 + \sum_{j=1}^{M} a_2[j]$
R times
R times

Figure 4: Visualization of algorithm 2

This means that we should design cache oblivious algorithms with the smallest depth that's possible.

The cache-oblivious sorting algorithm that we considered in class had depth $\Omega(\sqrt{n})$. Blelloch et al. published a cache-oblivious sorting algorithm with depth $O(\log^2(n))$ in 2011.

For private caches, the WS scheduler incurs $Q_p = Q_1 + M \cdot P \cdot D$ total I/O's. The proof idea is that each of the $O(P \cdot D)$ steals incurs $\Theta(M)$ I/O's.

There are schedulers that work for mixed caches, they are a combination of the WS and PDF schedulers with some additions.