CS855: Performance and scalability of parallel algorithms

Marc Moreno Maza

February 19, 2008

- A parallel program
- Usual Metrics of Parallel Algorithms

```
fib(n)
        if n < 2
        then return n
        x <- spawn fib(n-1)
        y <- spawn fib(n-2)
        sync
        return (x+y)
```

- A `spawn` instruction allows the next instruction of the parent program to be executed at the same time by a child, which is equivalent to `fork` in UNIX programming.
- A `sync` instruction requires that all locally spawned children finish their work before proceeding with the computation (equivalent to UNIX `join`).

Our syntax describes **logical parallelism** only. We do not state which tasks are assigned to which processors, leaving it to **scheduler**.

We model the instruction stream of a program as a **computation directed acyclic graph (DAG)**. Vertices are **threads**, which is a maximal sequence of instructions not containing any parallel control. We have three threads here:

Thread A: the `if` statement up to the first `spawn`.

Thread B: the computation of `n-2` up to the second `spawn`.

Thread C: the `x + y` operation before `return`.

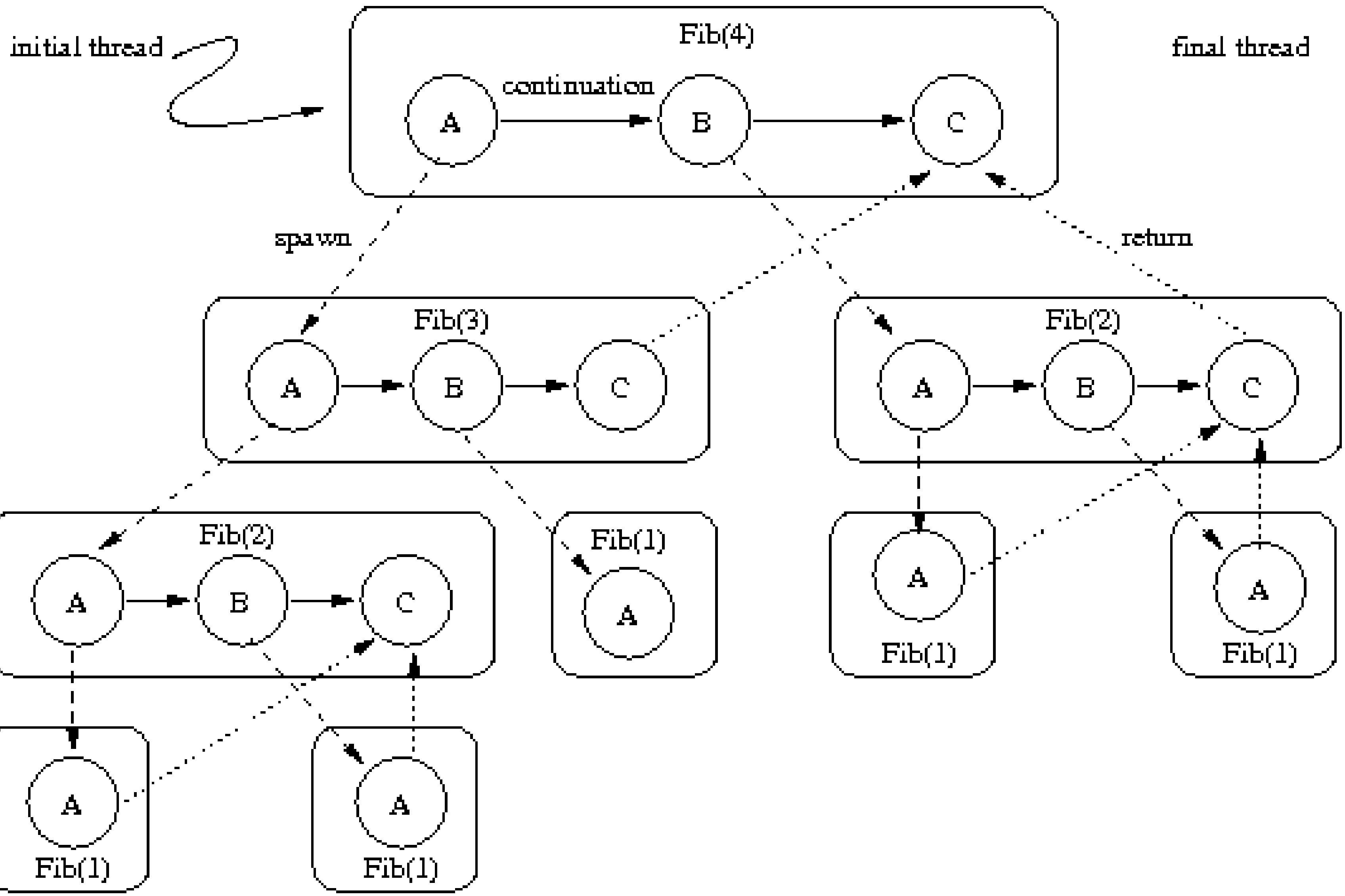
Definition 1 *We define several performance measures to evaluate various scheduler algorithms. We assume an ideal situation: no cache issues, no interprocessor costs:*

T_p *is the minimum running time on* p *processors.*

T_1 *is called the* **work**.

T_∞ *is the minimum running time with infinitely many processors.*

Instructions Streams as a DAG



On the previous figure:

- Each solid arrow represents a `continuation`.
- Each dashed arrow represents a `spawn`.
- Each dotted arrow represents a `return`.

Observe that we have $T_1 = 17$ and $T_\infty = 8$.

Proposition 1 *Assuming that all threads run in unit time, the longest path in the DAG is equal to T_∞. For this reason, T_∞ is referred to as the* critical path length.

When threads do not run in unit time, then the vertices of the DAG need to be weighted by their individual runtimes.

Proposition 2 *We have:* $T_p \geq T_1/p$.

Indeed, in the best case, p processors can do p works per unit of time.

Proposition 3 *We have:* $T_p \geq T_\infty$.

Indeed, $T_p < T_\infty$ contradicts the definitions of T_p and T_∞.

Definition 2 *A program's parallel execution plan can have:*

- linear speedup*:* $T_1/T_P = \Theta(P)$
- superlinear speedup*:* $T_1/T_P = \omega(P)$ *(not possible in this model, though it is possible in others).*
- sublinear speedup*:* $T_1/T_P = o(p)$

Definition 3 *A* scheduler*'s job is to map a computation to particular processors:*

- *If decisions are made at runtime, the scheduler is* online*, otherwise, it is* offline*.*
- *A scheduler is* greedy *if attempts to do as much work as possible at every step.*

In any *greedy schedule*, there are two types of steps:

- **complete step**: There are at least p threads that are ready to run. The greedy scheduler selects any p of them and runs them.
- **incomplete step**: There are strictly less than p threads that are ready to run. The greedy scheduler runs them all.

Theorem 1 (Graham, Brent) *Given p processors, a greedy scheduler executes any computation G with work T_1 and critical path length T_∞ in time:*

$$T_p \quad \leq \quad T_1/p \; + \; T_\infty.$$

PROOF. Observe that

- There can be no more than T_1/p complete steps, by definition of T_1.
- There can be no more than T_∞ incomplete steps. Indeed:
  - (i) Let G' be the subgraph of G that remains to be executed immediately prior to a given incomplete step.
  - (ii) During this incomplete step, all threads that can be run are actually run
  - (iii) Hence removing this incomplete step from G' would reduce its the critical-path length by one.

$\square$

Corollary 1 *A greedy scheduler is always within a factor of 2 of optimal.*

Proof. From Propositions 1 and 2, we have:

$$T_P \geq \max(T_1/P, T_\infty) \tag{1}$$

In addition, we can trivially express:

$$T_1/P \leq \max(T_1/P, T_\infty) \tag{2}$$

$$T_\infty \leq \max(T_1/P, T_\infty) \tag{3}$$

Given Theorem 1, we deduce:

$$\begin{align} T_P &\leq T_1/P + T_\infty \tag{4}\\ &\leq \max(T_1/P, T_\infty) + \max(T_1/P, T_\infty) \tag{5}\\ &\leq 2\max(T_1/P, T_\infty) \tag{6}\end{align}$$

which concludes the proof. □

Every step either brings to progress on the work (complete step), or on the critical path (incomplete step). A greedy scheduler is generally good.

Corollary 2 *The greedy scheduler achieves linear speedup when* $T_\infty = O(T_1/p)$.

Proof.

$$
\begin{aligned}
T_p &\leq T_1/p + T_\infty & (7)\\
&= T_1/p + O(T_1/p) & (8)\\
&= \Theta(T_1/p) & (9)
\end{aligned}
$$

The idea is to operate in the range where T_1/P dominates T_∞. As long as T_1/P dominates T_∞, all processors can be used efficiently.

- Let $T(n, p)$ the time elapsed
  - from the start of a parallel computation to the moment where the last processor finishes,
  - on an input data of size n,
  - and using p processors.
- $T(n, p)$ takes into account
  - computational steps (such as adding, multiplying, swapping variables),
  - routing (or communication) steps (such as transferring and exchanging information between processors).
- **Example:** parallel search of an item x
  - in an unsorted input file with n items,
  - in a shared memory with p processors,
  - where any cell can be accessed by only one processor at a time.

Broadcasting x costs $O(\log(p))$, leading to

$$T(n,p) = O(\log(p)) + O(n/p).$$

- Let $E(n,p)$ be the *parallel efficiency*, or *speedup per processor*, that is

$$E(n,p) = \frac{SU(n)}{pT(n,p)}$$

where $SU(n)$ is a lower bound for a sequential execution. Observe that we have $SU(n) \leq p\,T(n,p)$ and thus $E(n,p) \leq 1$.

- Reasons for inefficiency:
  - large communication latency compared to computational performances (it would be better to calculate locally rather than remotely)
  - too big overhead in synchronization, poor coordination, poor load distribution (processors must wait for dependent data),
  - lack of useful work to do (too many processors for too little work).

- The following metrics measure the performances of a parallel algorithm for given n and p.
  - *Parallel Cost* or *Processor-time product*: $C(n,p) = pT(n,p)$. Not very precise, but easy to measure.
  - *Parallel Work*: $W(n,p) = a_{t_{\mathrm{start}}} + \cdots + a_{t_{\mathrm{end}}}$ where a_t is the number of active processors a time t. Sharp, but hard to measure.
  - *Parallel speedup*: $S(n,p) = \frac{SU(n)}{T(n,p)} \leq p$.
- These other ones measure scalability, that is the ability to utilize increasing number of processors efficiently:
  - A *data-processor iso-efficiency function* is an asymptotically maximal function f_1 such that for all $p_0 > 0$ there exists n_0 such that for $n \geq n_0$ we have $E(n, f_1(n)) \geq E(n_0, p_0)$.
  - A *processor-data iso-efficiency function* is an asymptotically minimal function f_2 such that for all $n_0 > 0$ there exists p_0 such that for $p \geq p_0$ we have $E(f_2(p), p) \geq E(n_0, p_0)$.
  - The iso-efficiency function f_2 quantifies the growth rate of the

problem size, required to keep the efficiency fixed while increasing the number of processors. It reflects the ability of a parallel algorithm to maintain a constant efficiency. A large isoefficiency function indicates poor scalability, whereas a small one indicates that only a small increment in the problem size is sufficient for efficient exploitation of newly added processors.

- Consider the following problem: summing n numbers on a fully connected computer with $p \leq n$ processors. Assuming that every "basic" operation runs in unit time, we have $SU(n) = n$.
  * Each processor adds locally $\lceil \frac{n}{p} \rceil$ numbers.
  * Then the p partial sums are summed using a *parallel binary reduction* on p processors in $\lceil \log(p) \rceil$ iterations.
  * Thus, we have: $T(n, p) \in O(\frac{n}{p} + \log(p))$.
  * Elementary computations give $f_1(n) = \frac{n}{\log(n)}$ and $f_2(p) = p \log(p)$.