CS855: Performance and scalability of parallel algorithms

Marc Moreno Maza

February 19, 2008

- Parallelizing algorithms: some examples
- A parallel program
- Usual Metrics of Parallel Algorithms
- Topics covered in CS 855

- Single-processor machine cannot keep evolving following Moore's' law.
- Moreover, some computations may require much more memory resources than we have on an usual PC.
- Therefore, for significant improvements we need to use parallel computations.
- However, we should try
  - to distribute the work evenly among the processors (or machines),
  - to keep communication costs within acceptable limits.
- Indeed, parallel programs may run fast on some architectures and surprisingly slowly on others!

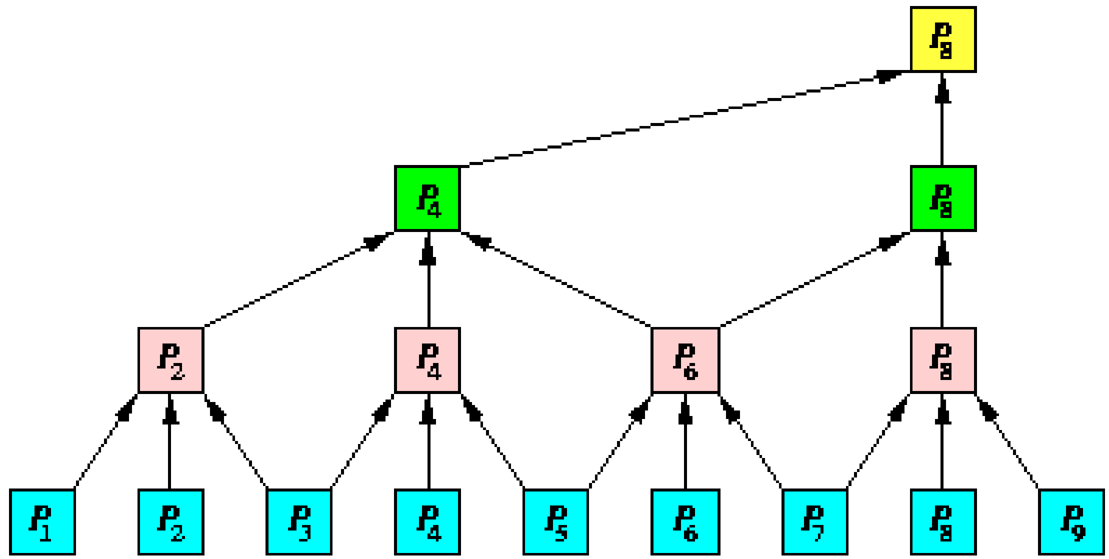
Consider a tridiagonal linear system of order n:

$$\begin{array}{ccccccc}
\dots & \dots & \dots & & & & \dots \\
a_{i-2}x_{i-2} & +\, b_{i-1}x_{i-1} & +\, c_i x_i & & & = & e_{i-1} \\
& a_{i-1}x_{i-1} & +\, b_i x_i & +\, c_{i+1}x_{i+1} & & = & e_i \\
& & a_i x_i & +\, b_{i+1}x_{i+1} & +\, c_{i+2}x_{i+2} & = & e_{i+1} \\
& & & \dots & \dots & \dots & \dots
\end{array}$$

For every even i replacing x_i with $-\frac{e_i - c_{i+1}x_{i+1} - a_{i-1}x_{i-1}}{b_i}$ leads to another tridiagonal system of order $n/2$:

$$\begin{array}{ccccccc}
\dots & \dots & \dots & & & & \dots \\
A_{i-3}x_{i-3} & +\, B_{i-1}x_{i-1} & +\, C_{i+1}x_{i+1} & & & = & E_{i-1} \\
& A_{i-1}x_{i-1} & +\, B_{i+1}x_{i+1} & +\, C_{i+3}x_{i+3} & & = & E_{i+1} \\
& & \dots & \dots & \dots & & \dots
\end{array}$$

An example of efficient parallelization



Observe that, on this example:

- the number of processors, here $p = n$, can be set such that
- the number of parallel steps, here $O(\log n)$, is known and small,
- processors activity (scheduling) is easy to organize,
- data-communication is not intensive.

Here's a second example of an algorithm that is *naturally parallel*. As for the *Karatsuba* algorithm, the trick relies on a smart use of distributivity:

(1) If $n = 1$ the matrices A et B are of the form (a) et (b) where $a, b \in \mathbf{Z}$, such that $A\,B$ equals $(a\,b)$.

(2) If $n > 1$ the matrices A and B can be decomposed as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \; B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

where A_{ij} and B_{ij} are square matrices of order $n/2$. Them, one computes the following 8 sums.

$$\begin{aligned} S_1 &= A_{21} + A_{22} & T_1 &= B_{12} - B_{11} \\ S_2 &= S_1 - A_{11} & T_2 &= B_{22} - T_1 \\ S_3 &= A_{11} - A_{21} & T_3 &= B_{22} - B_{12} \\ S_4 &= A_{12} - S_2 & T_4 &= T_2 - B_{21} \end{aligned}$$

(3) Next, one computes the following 7 matrix products by recursive calls to the algorithm.

$$\begin{aligned} P_1 &= A_{11}\, B_{11} & P_5 &= S_1\, T_1 \\ P_2 &= A_{12}\, B_{21} & P_6 &= S_2\, T_2 \\ P_3 &= S_4\, B_{22} & P_7 &= S_3\, T_3 \\ P_4 &= A_{22}\, T_4 & & \end{aligned}$$

(4) Finally, one computes the following 7 sums:

$$
\begin{aligned}
U_1 &= P_1 + P_2 & \qquad U_5 &= U_4 + P_3 \\
U_2 &= P_1 + P_6 & \qquad U_6 &= U_3 - P_4 \\
U_3 &= U_2 + P_7 & \qquad U_7 &= U_3 + P_5 \\
U_4 &= U_2 + P_5 & &
\end{aligned}
$$

and one returns:

$$
\begin{pmatrix} U_1 & U_5 \\ U_6 & U_7 \end{pmatrix}.
$$

Observe that, on this second example:

- one may need to duplicate data
- one has to be careful with data dependency
- how many processors can be busy at the same time?

$$\begin{cases} f_1 = x - 2 + (y-1)^2 \\ f_2 = (x-1)(y-1) + (x-2)y \\ f_3 = (x-1)z \end{cases}$$

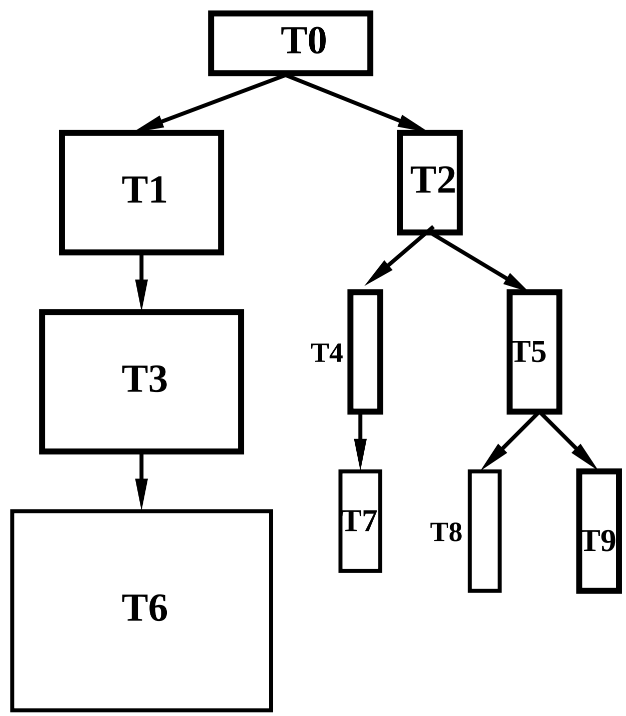
Factorizing f_3 leads to two sub-systems:

$$S_1 : \begin{cases} y = 0 \\ x = 1 \end{cases} \quad \text{and} \quad S_2 : \begin{cases} x - 1 + y^2 - 2y = 0 \\ (2y-1)x + 1 - 3y = 0 \\ z = 0 \end{cases}$$

The sub-system S_1 is solved. Continuing with S_2 leads finally to

$$\begin{cases} z = 0 \\ y = 0 \\ x = 1 \end{cases}, \begin{cases} z = 0 \\ y = 1 \\ x = 2 \end{cases} \quad \text{and} \quad \begin{cases} z = 0 \\ 2y = 3 \\ 4x = 7 \end{cases}$$

- Very irregular tasks (CPU time, memory, data-communication)
- For most systems with rational number coefficients, in theory and in practice, the solution set consists of one main piece and possibly a few tiny pieces.



```
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`.

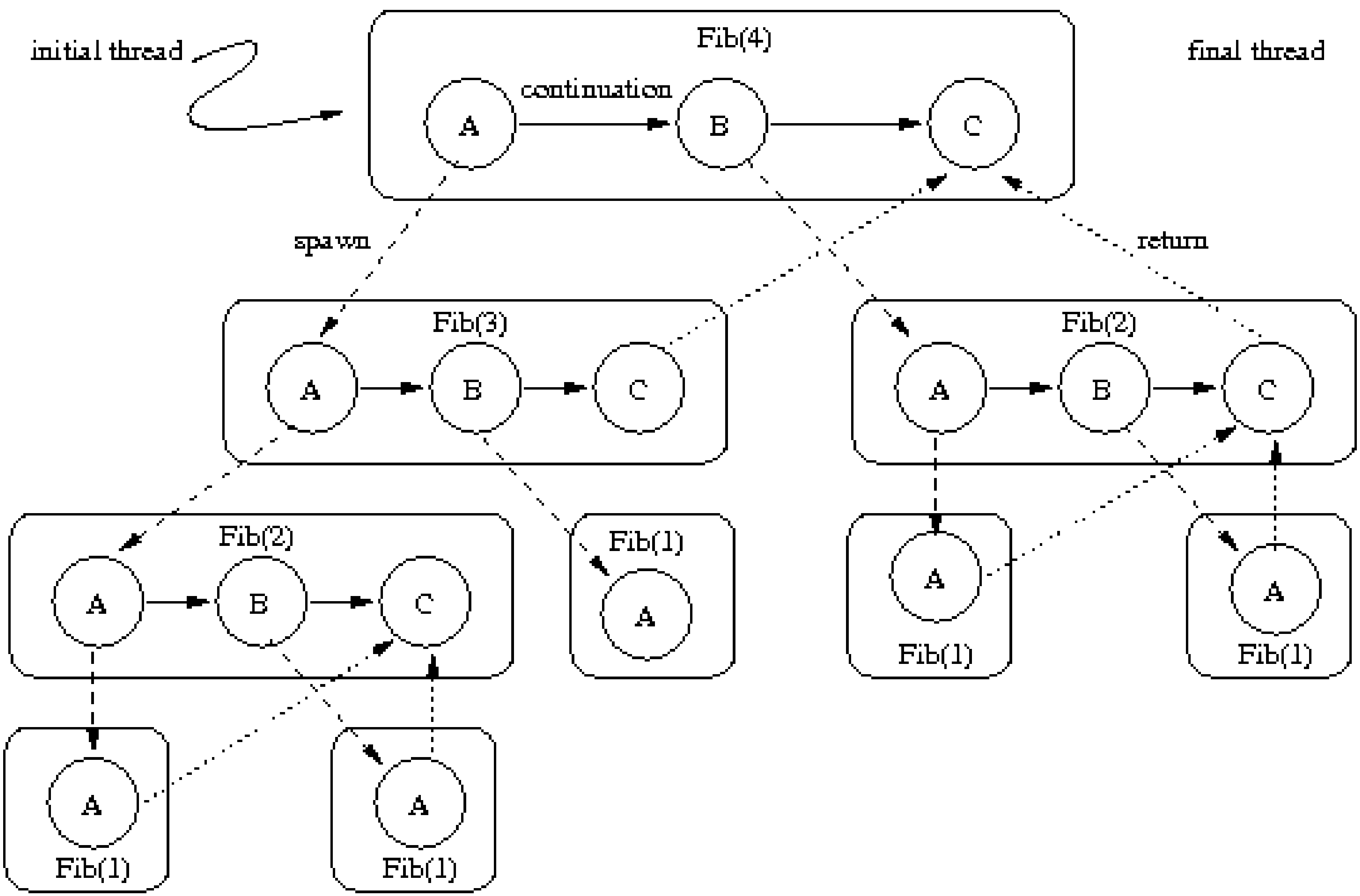
Thread C: the `x + y` operation before `return`.

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 on an infinite number of processors.


initial thread
Fib(4)
continuation
A
B
C
final thread
spawn
return
Fib(3)
A
B
C
Fib(2)
A
B
C
Fib(2)
A
B
C
Fib(1)
A
A
Fib(1)
A
Fib(1)
A
Fib(1)
A
Fib(1)

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 si 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_∞.

- 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 its 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 $f_2(p) = p \log(p)$.

- Performance and scalability of parallel algorithms,
- PRAM and BSP models.
- Parallel complexity theory: circuit complexity, communication complexity.
- Scheduling and load balancing
- Routing and broadcast algorithms.
- Parallel dense matrix operations.
- Solving systems of linear equations in parallel.
- Parallel computations of GCDs.
- Solving systems of non-linear linear equations in parallel.
- Implementation issues.
- Project presentations.

- Familiarity with *linear algebra*.
- Familiarity with *complexity analysis of algorithms*.
- No background in scientific computations or parallel computations is required.

One research project based on articles and leading to a

- a report and
- a presentation in class.