Parallel programming

S.I

// programming

Françoise Baude

Université Côte d'Azur

Polytech Nice Sophia

baude@unice.fr

web site: https://lms.univ-cotedazur.fr/course/view.php?id=14214

(self registration)

Feb 2023

Chapter 2: PRAM related measures, Complexity

Plan

1. **Some indicators to evaluate a PRAM algorithm**
2. Complexity of parallel problems

Work of PRAM algorithms

- SURFACE : number of processors used by a PRAM algorithm $A_{//}(N)$ working on a size N problem:
  - $H(A_{//}(N))$ = the **maximum** amount of **procs required** in a given parallel PRAM instruction, during the algo
- PARALLEL TIME $T_p(A_{//}(N))$ of the algo using P procs:
  - $T_p(A_{//}(N))$ = the number of computation steps when using P procs
- WORK : product of SURFACE by PARALLEL TIME
  - $W = \mathbf{H}(A_{//}(N)) * \mathbf{T}_{H(A_{//}(N))}(A_{//}(N))$

    $= P * T_P$

Speedup, Efficiency

- Consider the (best)Seq time to solve the problem $A_{seq}(N)$ whose time is $T_{seq}(N)$ (using 1 proc!)

- SPEEDUP (acceleration factor) of $A_{//}(N)$ using P procs
  - $S_p(N) = T_{seq}(N) \; / \; T_p(A_{//}(N))$
  - Theoretical goal is to have $S_p(N) = p$

By using p procs, the speed-up is p

- EFFICIENCY of $A_{//}(N)$ using P procs
  - e = Sequential work / Parallel work
  - $e = T_{seq}(N) * 1/ W$

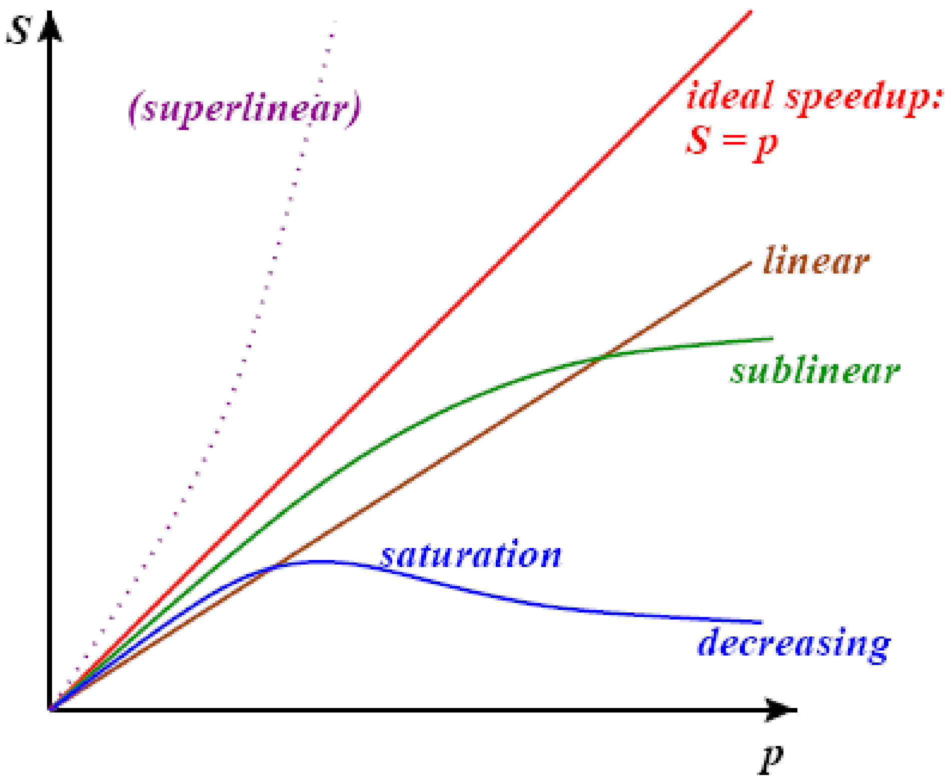
    $= T_{seq}(N) \; / \; p^* T_p(A_{//}(N))$
  - Theoretical goal is to have e=1

Seq runtime has been converted into adding procs!

Speedup curves

Speed-up is a factor, not a speed, nor a duration

Speedup curves measure the utility of parallel computing, not speed.



trivially parallel
(e.g., matrix product, LU decomposition, ray tracing)
$\rightarrow$ close to ideal $S = p$

work-bound algorithms
$\rightarrow$ linear $SU \in \Theta(p)$, work-optimal

tree-like task graphs
(e.g., global sum / max)
$\rightarrow$ sublinear $SU \in \Theta(p/\log p)$

There is a high variation in the number of proc use during the computation

communication-bound
$\rightarrow$ sublinear $SU = 1/fn(p)$

Most papers on parallelization show only relative speedup
(as $SU_{abs} \leq SU_{rel}$, and best seq. algorithm **would be needed for getting *Su*$_{abs}$**)

Speed-up in practice : ways to measure performances

- Theoretical Speed-up (Absolute speedup):
  - Computed using the complexity of the algorithm solving problem in sequential
- Can we always have the sequential time ?
  - Sometimes, no sequential implementation exists
    - Take the parallel version, run it using p = 1
  - Sometimes, not feasible when n is too big
    - Not enough memory to run with input of size n
    - Take the parallel version and measure its time, when p increases
  - ➔In these cases, we measure the RELATIVE speed-up
  - Sometimes, the sequential time gets penalized due to some memory cache effects
    - With more processors, the measured speedup becomes better than the theoretical speedup … ☺ : because mem accesses apply more often in cache
    - Speed-up becomes super-linear !

Example: Cost-optimal parallel sum algorithm on SB-PRAM

Saarbrucken Univ. PRAM: real PRAM machine [1990]

$n = 10,000$

Processors	Clock cycles	Time	SU_{rel}	SU_{abs}	EF_{rel}
Sequential	460118	1.84			
1	1621738	6.49	1.00	0.28	1.00
4	408622	1.63	3.97	1.13	0.99
16	105682	0.42	15.35	4.35	0.96
64	29950	0.12	54.15	15.36	0.85
256	10996	0.04	147.48	41.84	0.58
1024	6460	0.03	251.04	71.23	0.25

$EF_{rel} = \frac{Tpar(1)}{p^* \, Tpar(p)}$

$n = 100,000$

Processors	Clock cycles	Time	SU_{rel}	SU_{abs}	EF_{rel}
Sequential	4600118	18.40			
1	16202152	64.81	1.00	0.28	1.00
4	4054528	16.22	4.00	1.13	1.00
16	1017844	4.07	15.92	4.52	0.99
64	258874	1.04	62.59	17.77	0.98
256	69172	0.28	234.23	66.50	0.91
1024	21868	0.09	740.91	210.36	0.72

Theorem: *Conservation of work by simulation*

- Given an algorithm A running in time t on p procs. of a given PRAM

  It is possible to simulate A on a same sort of PRAM using p' <=p procs in time O(t (p / p'))*

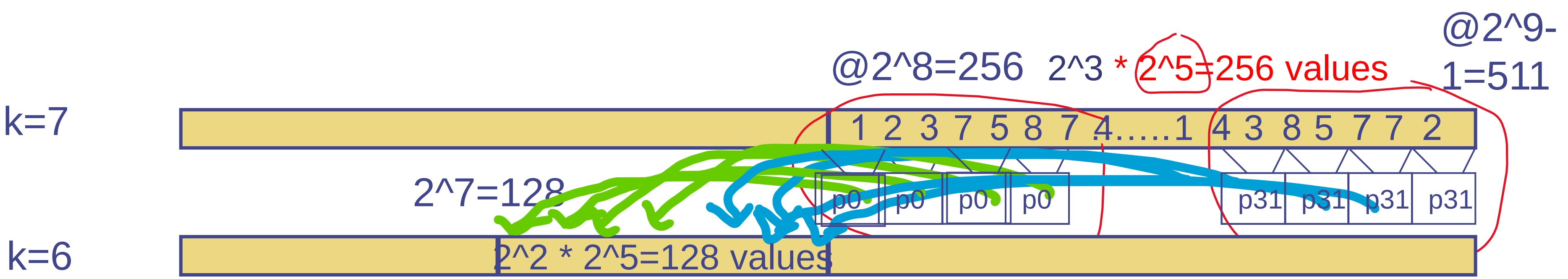
- Proof: intuitive! At each step, each p' proc will have to execute a subset of (p/p') // instructions in sequence
- Work is kept as it is:
  - W = p * t , new W = p' * (t*(p/p'))= t * p

Exemple: EREW maximum computation –v2 => v3

m=k=8, k'=6

N=2^k=2^8 = 256. p=2^(k-1)=2^7=128. p'=2^(k'-1)=2^5=32

each of the O(log(2^8))=O(8) instructions of the $Maxv2_{//}$(2^8) will be simulated by up to 2^5 procs, executing up to 2^(k-k')=2^2=4 max binary operations



Pour (k=m-1; k>=0; k--) ***/*still costs m parallel steps */***

/*enroll up to 2^(k-1) procs per step ? NO, just enroll up to q=2^(k'-1) */

<u>Pour chaque</u> proc q <u>en parallele</u>

Pour (l=0;l<2^(k-k'); l++) /*costs 2^(k-k') seq time*/

A[zz] = max(A[yy],A[yy+1]);

Consequences of Theorem: *Conservation of work by simulation*

- The work of a $A_{//}(N)$ with p proc is **at least** in the order of $T_{seq}(N)$, the best sequential algo

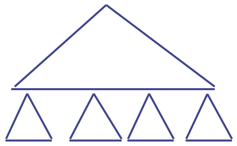
$$H\ (A_{//}(N)) * T_p(A_{//}(N)) \ >= T_{seq}(N)$$

$$T_{seq}(N) <= H\ (A_{//}(N)) * T_p(A_{//}(N))$$

- Proof: by absurd
  - Suppose $H\ (A_{//}(N)) * T_p(A_{//}(N)) \ < T_{seq}(N)$,
  - then, decide p'=1,
    - Each p' proc will have to execute a subset of (p/p') // instructions in sequence
  - If $T_1(A_{//}(N)) \ < T_{seq}(N)$, the seq algo was not the best !!

WORK EFFICIENCY

- A parallel algorithm is said to be WORK EFFICIENT:
  - Its work is of the same amount than the best sequential algorithm
  - i.e. e == 1
- (counter-)Examples:
  - *max-v3* versus O(n) seq max.
    - Using the simulation theorem
      - O(t* (p / p')) ; p=n; choose p'=n/logn ; t = log n
    - max-v3 time = log n * n/n/logn = $\log^2$ n (limiting factor: same t)
    - max-v3 work= n/logn * $\log^2$ n = n*log n => not work efficient
      - Because of the same t=log n on same initial size=n ….
  - *max-v2 with subtrees*, versus O(n) seq max.
    - [log n + O(log (n/logn))] * (n/logn) = O(n), work eff.
    - Here max-v2 with subtrees applies the Brent Principle



Brent Principle

- A general principle to decrease number of used procs (not a method, just a principle!)
- Given $A_{//}(N)$ having a total of m operations, running in $t=T(A_{//}(N))$ with an unbounded number of procs

One can simulate $A_{//}(N)$ in time $O(m/p + t)$ on a similar PRAM using p processors

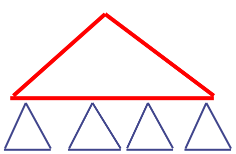
- Proof: at step i, $A_{//}$ runs m(i) ops s.t. $\sum m(i) = m$
  - Simulation with p procs takes $\lceil m(i)/p \rceil \leq m(i)/p+1$

$$\text{In total, the time of simulation} \leq \sum_{i=1 \text{ to } t} (m(i)/p + 1) = m/p + t$$

- Consequence: if $(m/p)=t$ then $T_p(A_{//}(N)) <= 2*t$

(Proof of) Brent has no real incidence!

- The proof is not constructive
  - It is not a method, it is not a receipe in order to go from an unlimited number of procs to a practical & concrete number of p processors
    - It does not tell how to concretely split m operations into m(i)
    - For a given step i, it does not tell how to split the m(i) operations into p tasks (it is not a load sharing method)
      - E.g list ranking on a linked list of size n: it is not easy to split the list into p sublists, each of successive elements, of size n/p
      - Still, in some cases, like working on linear structures such as arrays, quite easy to be split into seq tasks then parallel tasks, eg:
  - Ex: on max-v2, how many max ops ? $\Sigma\ m(i) = m$



    - k=8,2^7=128; + k=7, 2^6=64; + 2^5=32; … + k=0, 2^0
    - Choose p'=p/t=n/log n=256/8=32; t=8 =>t'=log(n/logn)=5
    - New Step1: 256/32=2^8/2^5=8 values to work on, in seq. per proc,tseq=8 => we needed to modify the //algo (8x32 max ops)
    - Next steps: just parallel max-v2 working on 32 values: t'=5

Plan

1. Some indicators to evaluate a PRAM algorithm
2. **Complexity of parallel problems**

The NC complexity class of parallel problems

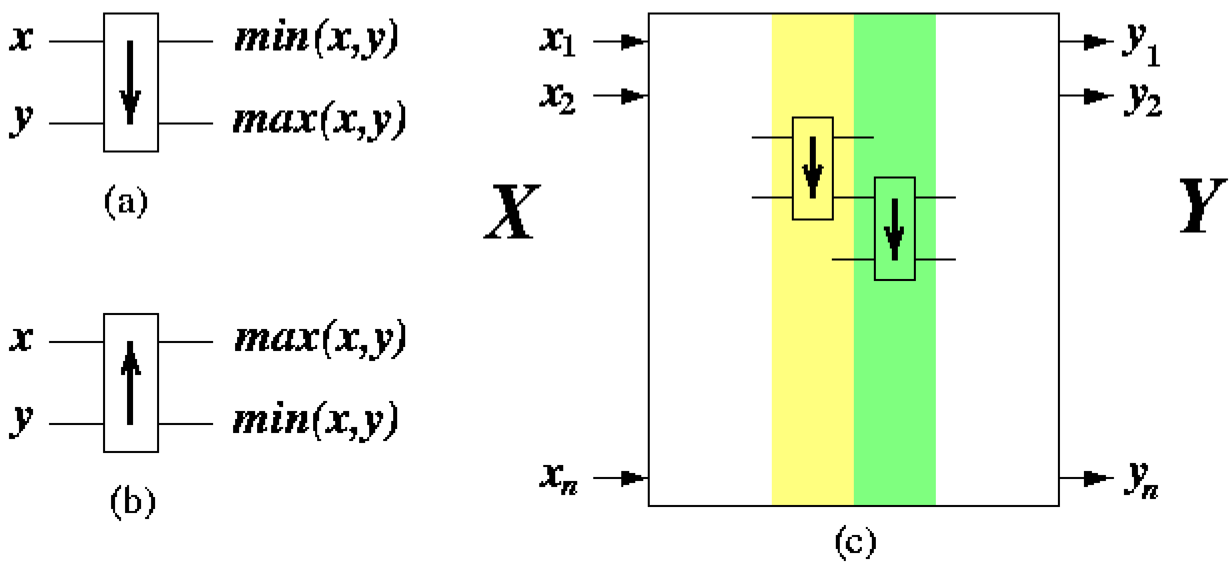
- It tell us what is a « good » parallel algo
- NC complexity class (« Nick's Class »)
  - The set of problems for which there exists a parallel algorithm taking a (poly)logarithmic parallel time, and using a polynomial number of processors
  - $\mathcal{NC}$ in $\mathcal{P}$, $\mathcal{P}$? in $\mathcal{NC}$, probably ≠
- An « Optimal » parallel algorithm :
  - Belongs to NC, and moreover is efficient (in work)
- Be careful in practice with the poly-log time:
  - Ex: A //time = $\log^3 n << n^{1/4}$ only when $n > 10^{12}$
- More: [Parallel complexity theory - NC algorithms (wisc.edu)]() and [Parallel complexity theory - P-completeness (wisc.edu)]()

Sort n values in an optimal way ?

- Complexity to sort n values:
  - Somehow, they all must be compared 2 by 2 (cf max-v1)
  - It is known that the lower bound in sequential is $\Omega(n*\log n)$
  - =>It provides us with a framework for parallel algorithms!
    - **With only O(n) procs. used, goal is to sort in O(logn) //time**
      - It is feasible, but the factor hidden in the O( ) is very high
      - Principle of the merge parallel sort algo on an EREW [due to Cole] :
        - Start from n lists of size 1, merge them two by two in //
        - Start again, to merge all these lists 2 by 2, and so on
        Depth of the tree to traverse from leaves to root: log n, so, the sub goal is to **merge two lists of length O(n) in constant time ….** It is hard but feasible ; make inactive procs of the upper stages in the tree become active in order to contribute to these merge operations in O(1) // time that run at lower stages: pipeline, anticipate
- In pratice: Sort in // in time $O(\log^2 n)$, non optimal
  - On a PRAM
  - Or, on a « sorting network » : it is a topology of *sorting elements* that is always the same whatever be the initial sequence of input data to be sorted
    - Provides an « oblivious » or « regular » algorithm (i.e., just dependant of the problem size, not of the data values)

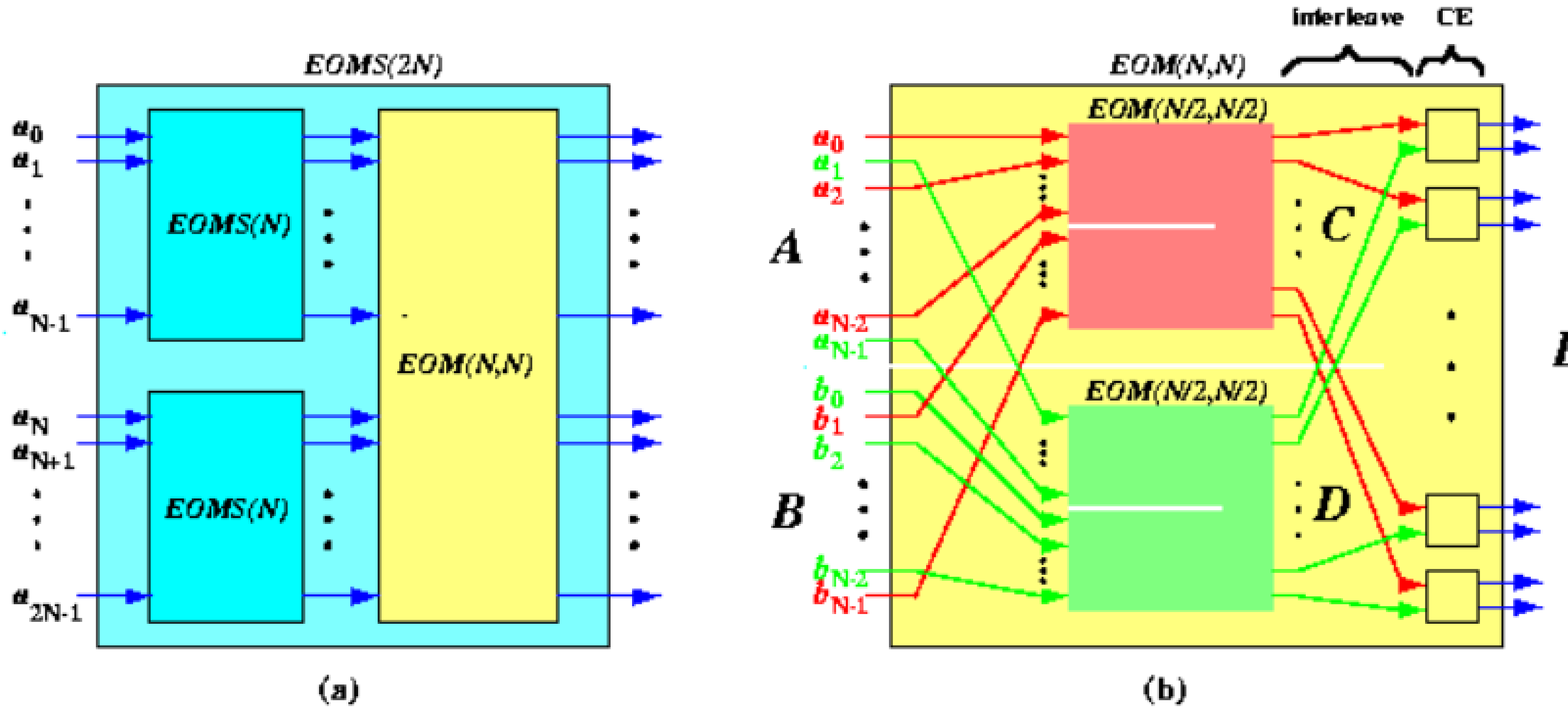
Architecture of a sorting network

- Built using 2x2 comparators/sorting elements
- Architecture of comparison-exchange sorting networks.
  - (a) The default type of comparator
  - (b) The second type of comparator.
  - (c) Sorting network composed from columns of basic comparators.



[Introduction to parallel sorting on mesh-based topologies (wisc.edu)]

Even-Odd Merge Sorting network



Even-Odd MergeSort (a) and Merge (b) network

C={EOMerge}(even(A),odd(B))

D={EOMerge}(odd(A),even(B))

L'={Interleave}(C,D)

L={EOMerge}(A,B)={Pairwise_CE}(L')

[Sorting on hypercubic networks (wisc.edu)]