Theoretical Parallel Computing

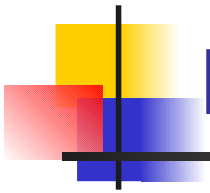
Arnaud Legrand, CNRS, University of Grenoble

LIG laboratory, arnaud.legrand@imag.fr

November 1, 2009

Outline

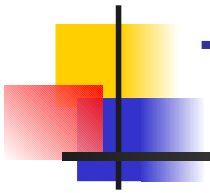
Outline



Models for Parallel Computation

- We have seen how to implement parallel algorithms in practice
  - We have come up with performance analyses
- In traditional algorithm complexity work, the Turing machine makes it possible to precisely compare algorithms, establish precise notions of complexity, etc..
- Can we do something like this for parallel computing?
- Parallel machines are complex with many hardware characteristics that are difficult to take into account for algorithm work (e.g., the network), is it hopeless?
- The most famous theoretical model of parallel computing is the PRAM model
  - We will see that many principles in the model are really at the heart of the more applied things we've seen so far

Courtesy of Henri Casanova

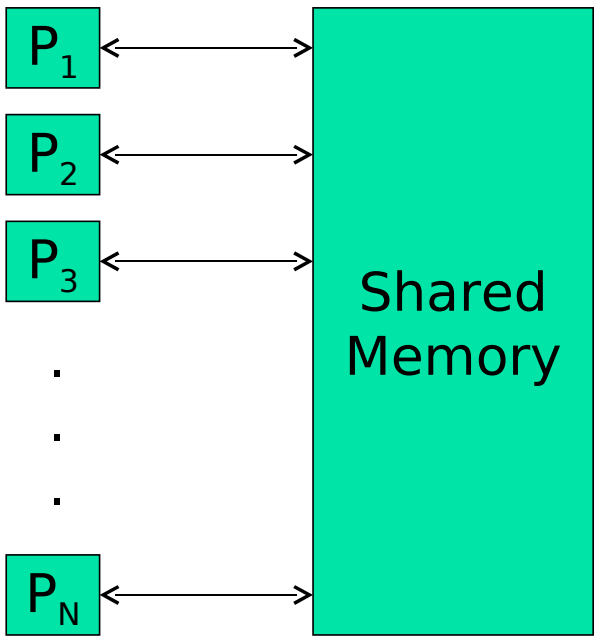


The PRAM Model

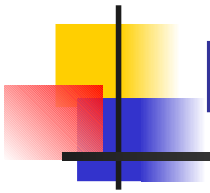
- Parallel Random Access Machine (PRAM)
- An imperfect model that will only *tangentially* relate to the performance on a real parallel machine
  - Just like a Turing Machine tangentially relate to the performance of a real computer
- Goal: Make it possible to reason about and classify parallel algorithms, and to obtain complexity results (optimality, minimal complexity results, etc.)
- One way to look at it: makes it possible to determine the "maximum parallelism" in an algorithm or a problem, and makes it possible to devise new algorithms

The PRAM Model

- Memory size is infinite, number of processors in unbounded
  - But nothing prevents you to "fold" this back to something more realistic
- No direct communication between processors
  - they communicate via the memory
  - they can operate in an asynchronous fashion
- Every processor accesses any memory location in 1 cycle
- Typically all processors execute the same algorithm in a synchronous fashion
  - READ phase
  - COMPUTE phase
  - WRITE phase
- Some subset of the processors can stay idle (e.g., even numbered processors may not work, while odd processors do, and conversely)



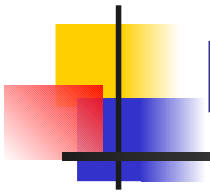
Courtesy of Henri Casanova



Memory Access in PRAM

- Exclusive Read (ER): p processors can simultaneously read the content of p distinct memory locations.
- Concurrent Read (CR): p processors can simultaneously read the content of p' memory locations, where $p' < p$.
- Exclusive Write (EW): p processors can simultaneously write the content of p distinct memory locations.
- Concurrent Write (CW): p processors can simultaneously write the content of p' memory locations, where $p' < p$.

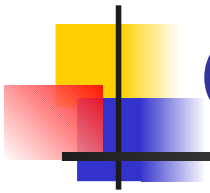
Courtesy of Henri Casanova



PRAM CW?

- What ends up being stored when multiple writes occur?
  - priority CW: processors are assigned priorities and the top priority processor is the one that counts for each group write
  - Fail common CW: if values are not equal, no change
  - Collision common CW: if values not equal, write a “failure value”
  - Fail-safe common CW: if values not equal, then algorithm aborts
  - Random CW: non-deterministic choice of the value written
  - Combining CW: write the sum, average, max, min, etc. of the values
  - etc.
- The above means that when you write an algorithm for a CW PRAM you can do any of the above at any different points in time
- It doesn’t corresponds to any hardware in existence and is just a logical/algorithmic notion that could be implemented in software
- In fact, most algorithms end up not needing CW

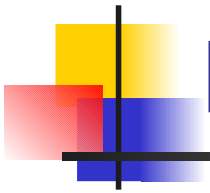
Courtesy of Henri Casanova



Classic PRAM Models

- CREW (concurrent read, exclusive write)
  - most commonly used
- CRCW (concurrent read, concurrent write)
  - most powerful
- EREW (exclusive read, exclusive write)
  - most restrictive
  - unfortunately, probably most realistic

- Theorems exist that prove the relative power of the above models (more on this later)



PRAM Example 1

- Problem:
  - We have a linked list of length n
  - For each element i, compute its distance to the end of the list:

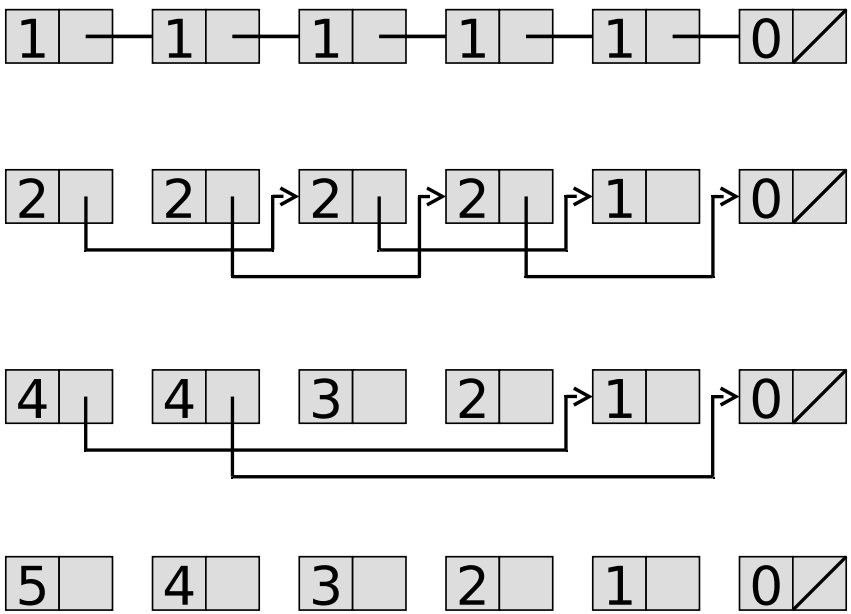
    d[i] = 0 if next[i] = NIL

    d[i] = d[next[i]] + 1 otherwise
- Sequential algorithm in O(n)
- We can define a PRAM algorithm in O(log n)
  - associate one processor to each element of the list
  - at each iteration split the list in two with odd-placed and even-placed elements in different lists

PRAM Example 1

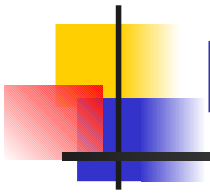
Principle:

Look at the next element
Add its d[i] value to yours
Point to the next element's next element



The size of each list is reduced by 2 at each step, hence the O(log n) complexity

Courtesy of Henri Casanova

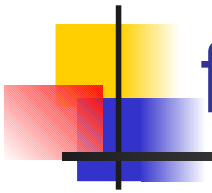


PRAM Example 1

- Algorithm

forall i
  if next[i] == NIL then d[i] $\leftarrow$ 0 else d[i] $\leftarrow$ 1
while there is an i such that next[i] $\neq$ NIL
  forall i
    if next[i] $\neq$ NIL then
      d[i] $\leftarrow$ d[i] + d[next[i]]
      next[i] $\leftarrow$ next[next[i]]

What about the correctness of this algorithm?



forall loop

- At each step, the updates must be synchronized so that pointers point to the right things:

  next[i] ← next[next[i]]

- Ensured by the semantic of forall

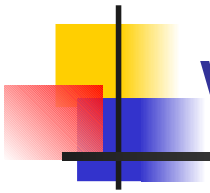
```
forall i
  A[i] = B[i]
```

⟺

```
forall i
  tmp[i] = B[i]
forall i
  A[i] = tmp[i]
```

- Nobody really writes it out, but one mustn't forget that it's really what happens underneath

Courtesy of Henri Casanova

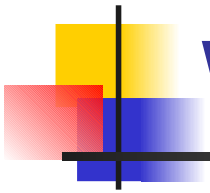


while condition

- while there is an i such that next[i] ≠NULL
- How can one do such a global test on a PRAM?
  - Cannot be done in constant time unless the PRAM is CRCW
    - At the end of each step, each processor could write to a same memory location TRUE or FALSE depending on next[i] being equal to NULL or not, and one can then take the AND of all values (to resolve concurrent writes)
    - On a PRAM CREW, one needs O(log n) steps for doing a global test like the above
- In this case, one can just rewrite the while loop into a for loop, because we have analyzed the way in which the iterations go:

  for step = 1 to $\lceil \log n \rceil$

Courtesy of Henri Casanova



What type of PRAM?

- The previous algorithm does not require a CW machine, but:

  tmp[i] $\leftarrow$ d[i] + d[next[i]]

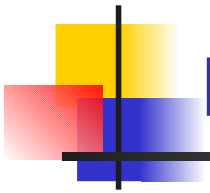
  which requires concurrent reads on proc i and j such that j = next[i].

- Solution:
  - split it into two instructions:

    tmp2[i] $\leftarrow$ d[i]

    tmp[i] $\leftarrow$ tmp2[i] + d[next[i]]

  (note that the above are technically in two different forall loops)
- Now we have an execution that works on a EREW PRAM, which is the most restrictive type



Final Algorithm on a EREW PRAM

```
forall i
    if next[i] == NILL then d[i] ← 0 else d[i] ← 1
for step = 1 to ⌈log n⌉
    forall i
      if next[i] ≠ NIL then
          tmp[i] ← d[i]
          d[i] ← tmp[i] + d[next[i]]
          next[i] ← next[next[i]]
```

O(1)

O(1)

O(log n)

O(log n)

Conclusion: One can compute the length of a list of size n in time O(log n) on any PRAM

Courtesy of Henri Casanova

Cost, Work

- Let P be a problem of size n that we want to solve (e.g., a computation over an n-element list).
- Let $T_{seq}(n)$ be the execution time of the best (known) sequential algorithm for solving P.
- Let us now consider a PRAM algorithm that solves P in time $T_{par}(p, n)$ with p PUs.

Definition: Cost and Work.

The cost of a PRAM algorithm is defined as

$$C_p(n) = p.T_{par}(p, n) \ .$$

The work $W_p(n)$ of a PRAM algorithm is the sum over all PUs of the number of performed operations. The difference between cost and work is that the work does not account for PU idle time.

Intuitively, cost is a rectangle of area $T_{par}(p, n)$.
Therefore, the cost is minimal if, at each step, all PUs are used to perform useful computations, i.e. computations that are part of the sequential algorithm.

Speedup and Efficiency

The speedup of a PRAM algorithm is the factor by which the program's execution time is lower than that of the sequential program.

Definition: Speedup and Efficiency.

The speedup of a PRAM algorithm is defined as

$$S_p(n) = \frac{T_{seq}(n)}{T_{par}(p,n)} .$$

The efficiency of the PRAM algorithm is defined as

$$e_p(n) = \frac{S_p(n)}{p} = \frac{T_{seq}(n)}{p.T_{par}(p,n)} .$$

Some authors use the following definition: $S_p(n) = \frac{T_{par}(1,n)}{T_{par}(p,n)}$, where $T_{par}(1,n)$ is the execution time of the algorithm with a single PU.

This definition quantifies how the algorithm scales: If the speedup is close to p (i.e. is $\Omega(p)$), one says that the algorithm scales well.

However, this definition does not reflect the quality of the parallelization (i.e. how much we can really gain by parallelization), and it is thus better to use $T_{seq}(n)$ than $T_{par}(1,n)$.

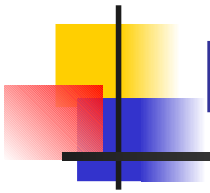
Brent Theorem

Theorem 1: Brent.

Let $\mathcal{A}$ be an algorithm that executes a total number of m operations and that runs in time t on a PRAM (with some unspecified number of PUs). $\mathcal{A}$ can be simulated in time $O\left(\frac{m}{p}+t\right)$ PRAM of the same type that contains p PUs.

Proof.

Say that at step i $\mathcal{A}$ performs $m(i)$ operations (implying that $\sum_{i=1}^{t} m(i) = m$). Step i can be simulated with p PUs in time $\left\lceil \frac{m(i)}{p} \right\rceil \leqslant \frac{m(i)}{p} + 1$. One can simply sum these upper bounds to prove the theorem. □



Brent Theorem

- **Theorem:** Let A be an algorithm with m operations that runs in time t on some PRAM (with some number of processors). It is possible to simulate A in time O(t + m/p) on a PRAM of same type with p processors
- Example: maximum of n elements on an EREW PRAM
  - Clearly can be done in O(log n) with O(n) processors
    - Compute series of pair-wise maxima
    - The first step requires O(n/2) processors
  - What happens if we have fewer processors?
  - By the theorem, with p processors, one can simulate the same algorithm in time O(log n + n / p)
  - If p = n / log n, then we can simulate the same algorithm in O(log n + log n) = O(log n) time, which has the same complexity!
  - This theorem is useful to obtain lower-bounds on number of required processors that can still achieve a given complexity.

Courtesy of Henri Casanova

Another Useful Theorem

Theorem 2.

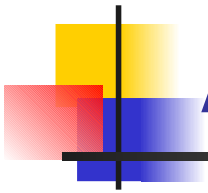
Let $\mathcal{A}$ be an algorithm whose execution time is t on a PRAM with p PUs. $\mathcal{A}$ can be simulated on a RAM of the same type with $p' \leqslant p$ PUs in time $O\left(\frac{t.p}{p'}\right)$. The cost of the algorithm on the smaller PRAM is at most twice the cost on the larger PRAM.

Proof.

Each step of $\mathcal{A}$ can be simulated in at most $\left\lceil \frac{p}{p'} \right\rceil$ time units with $p' \leqslant p$ PUs by simply reducing concurrency and having the p' PUs perform sequences of operations. Since there are at most t steps, the execution time t' of the simulated algorithm is at most $\left\lceil \frac{p}{p'} \right\rceil t$.

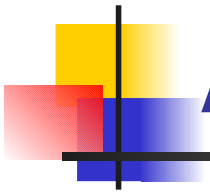
Therefore, $t' = O\left(\frac{p}{p'}.t\right) = O\left(\frac{t.p}{p'}\right)$.

We also have $C_{p'} = t'.p' \leqslant \left\lceil \frac{p}{p'} \right\rceil p'.t \leqslant \left(\frac{p}{p'} + 1\right) p't = p.t\left(1 + \frac{1}{p'}\right) = C_p.(1 + 1/p') \leqslant 2C_p.$ □



An other useful theorem

- **Theorem:** Let A be an algorithm that executes in time t on a PRAM with p processors. One can simulate A on a PRAM with p' processors in time O(t.p/p')
- This makes it possible to think of the "folding" we talked about earlier by which one goes from an unbounded number of processors to a bounded number
  - A.n.log n + B on n processors
  - $A.n^2 + Bn/(\log n)$ on log n processors
  - $A(n^2 \log n)/10 + Bn/10$ on 10 processors



Are all PRAMs equivalent?

- Consider the following problem
  - given an array of n elements, $e_{i=1,n}$, all distinct, find whether some element e is in the array
- On a CREW PRAM, there is an algorithm that works in time O(1) on n processors:
  - initialize a boolean to FALSE
  - Each processor i reads e_i and e and compare them
  - if equal, then write TRUE into the boolean (only one proc will write, so we're ok for CREW)
- One a EREW PRAM, one cannot do better than log n
  - Each processor must read e separately
  - at worst a complexity of O(n), with sequential reads
  - at best a complexity of O(log n), with series of "doubling" of the value at each step so that eventually everybody has a copy (just like a broadcast in a binary tree, or in fact a k-ary tree for some constant k)
  - Generally, "diffusion of information" to n processors on an EREW PRAM takes O(log n)
- Conclusion: CREW PRAMs are more powerful than EREW PRAMs

Courtesy of Henri Casanova

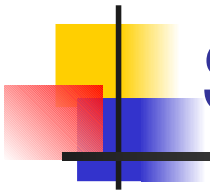
CRCW>CREW


```
1  COMPUTE_MAXIMUM(A, n)
2      forall i ∈ {1, ..., n} in parallel do
3          m[i] ← True
4      forall i, j ∈ {1, ..., n}^2, i ≠ j in parallel do
5          if A[i] < A[j] then m[i] ← False
6      forall i ∈ {1, ..., n} in parallel do
7          if m[i] = True then max ← A[i]
8      return max
```

ALGORITHM 1.3: CRCW algorithm to compute the largest value of an array.

- $O(1)$ on a CRCW
- $\Omega(\log n)$ on a CREW

Therefore, we have CRCW $>$ CREW $>$ EREW (with at least a logarithmic factor each time).

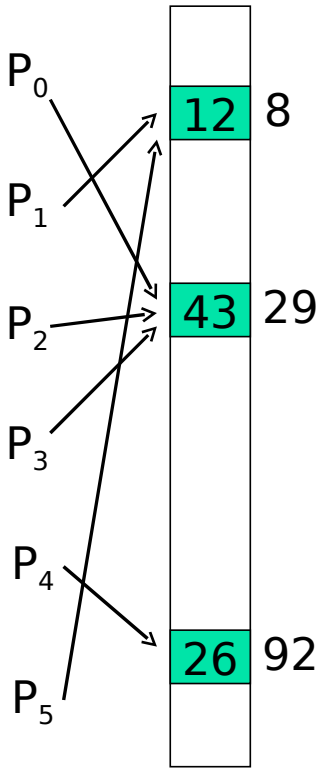


Simulation Theorem

- **Simulation theorem**: Any algorithm running on a CRCW PRAM with p processors cannot be more than O(log p) times faster than the best algorithm on a EREW PRAM with p processors for the same problem
- **Proof:**
  - "Simulate" concurrent writes
    - Each step of the algorithm on the CRCW PRAM is simulated as log(p) steps on a EREW PRAM
    - When Pi writes value x_i to address l_i, one replaces the write by an (exclusive) write of (l_i, x_i) to A[i], where A[i] is some auxiliary array with one slot per processor
    - Then one sorts array A by the first component of its content
    - Processor i of the EREW PRAM looks at A[i] and A[i-1]
      - if the first two components are different **or** if i = 0, write value x_i to address l_i
    - Since A is sorted according to the first component, writing is exclusive

Proof (continued)

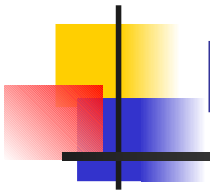
Picking one processor for each competing write



P0 → (29,43)	= A[0]	
P1 → (8,12)	= A[1]	
P2 → (29,43)	= A[2]	sort
P3 → (29,43)	= A[3]	
P4 → (92,26)	= A[4]	
P5 → (8,12)	= A[5]	

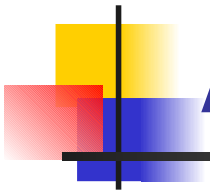
A[0]=(8,12)	P0 writes
A[1]=(8,12)	P1 nothing
A[2]=(29,43)	P2 writes
A[3]=(29,43)	P3 nothing
A[4]=(29,43)	P4 nothing
A[5]=(92,26)	P5 writes

Courtesy of Henri Casanova



Proof (continued)

- Note that we said that we just sort array A
- If we have an algorithm that sorts p elements with O(p) processors in O(log p) time, we're set
- Turns out, there is such an algorithm: Cole's Algorithm.
  - basically a merge-sort in which lists are merged in constant time!
  - It's beautiful, but we don't really have time for it, and it's rather complicated
- Therefore, the proof is complete.



And many, many more things

- J. Reiff (editor), Synthesis of Parallel Algorithms, Morgan Kauffman, 1993
  - Everything you've ever wanted to know about PRAMs
- Every now and then, there are references to PRAMs in the literature
  - "by the way, this can be done in O(x) on a XRXW PRAM"
  - This network can simulate a EREW PRAM, and thus we know a bunch of useful algorithms (and their complexities) that we can instantly implement
  - etc.
- You probably will never care if all you do is hack MPI and OpenMP code

Courtesy of Henri Casanova

Relevance of the PRAM model

- Central question in complexity theory for sequential algorithms: P = NP ?
- Central question in complexity theory for parallel algorithms: P = NC ?
- NC is the class of all problems that, with a polynomial number of PUs, can be solved in polylogarithmic time. An algorithm of size n is polylogarithmic if it can be solved in $O(\log(n)^c)$ time with $O(n^k)$ PUs, where c and k are constants.

Nice theoretical complexity considerations, uh ?

Skeptical readers may question the relevance of the PRAM model for practical implementation purposes:

> *You can't have $n = 100,000,000$ PUs.*
> *One never has an immediately addressable, unbounded shared parallel memory.*

Relevance of the PRAM model

This criticism, similar to the criticism of $O(n^{17})$ "polynomial" time algorithms, often comes from a misunderstanding of the role of theory:

- Theory is not everything: Theoretical results are not to be taken as is and implemented by engineers.
- Theory is also not nothing: The fact that an algorithm cannot in general be implemented as is, does not mean it is meaningless.

Deep understanding

When a $O(n^{17})$ algorithm is designed for a problem, it does not lead to a practical way to solve that problem.
However, it proves something inherent about the problem (namely, that it is in P).
Hopefully, in the process of proving this result, key insights may be developed that can later be used in practical algorithms for this or other problems, or for other theoretical results.
Similarly, the design of PRAM algorithms for a problem proves something inherent to the problem (namely, that it is parallelizable) and can in turn lead to new ideas.

Relevance of the PRAM model

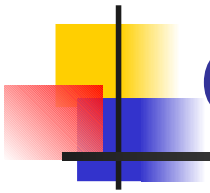
This criticism, similar to the criticism of $O(n^{17})$ "polynomial" time algorithms, often comes from a misunderstanding of the role of theory:

- Theory is not everything: Theoretical results are not to be taken as is and implemented by engineers.
- Theory is also not nothing: The fact that an algorithm cannot in general be implemented as is, does not mean it is meaningless.

Candidate for Practical Implementation

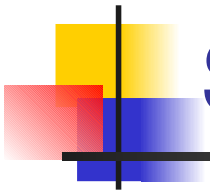
Even if communications are not taken into account in the performance evaluation of PRAM algorithms (a potentially considerable discrepancy between theoretical complexity and practical execution time), trying to design fast PRAM algorithms is not a useless pursuit.
Indeed, PRAM algorithms can be simulated on other models and do not necessarily incur prohibitive communication overheads.
It is commonly admitted that only cost-optimal PRAM algorithms have potential practical relevance and that the most promising are those cost-optimal algorithms with minimal execution time.

Outline



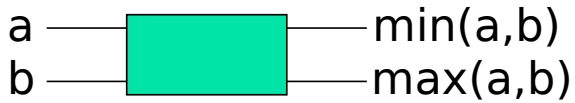
Combinational circuits/networks

- More realistic than PRAMs
- More restricted
- Algorithms for combinational circuits were among the first parallel algorithms developed
- Understanding how they work makes it easier to learn more complex parallel algorithms
- Many combinational circuit algorithms provide the basis for algorithms for other models (they are good building blocks)
- We're going to look at:
  - sorting networks
  - FFT circuit



Sorting Networks

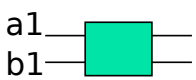
- Goal: sort lists of numbers
- Main principle
  - computing elements take two numbers as input and sort them

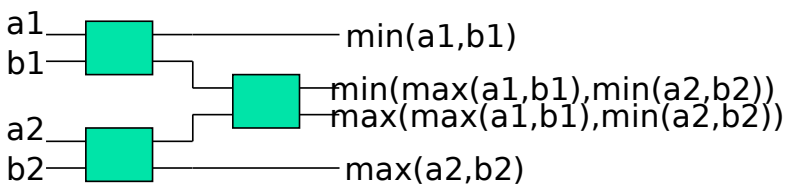


  - we arrange them in a network
  - we look for an architecture that depends only on the size of lists to be sorted, not on the values of the elements

Merge-sort on a sorting network

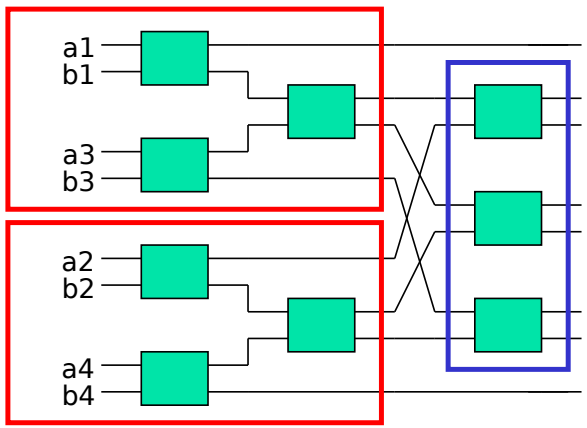
- First, build a network to merge two lists
- Some notations
  - $(c_1,c_2,...,c_n)$ a list of numbers
  - sort$(c_1,c_2,...,c_n)$ the same list, sorted
  - sorted$(x_1,x_2,...,x_n)$ is true if the list is sorted
  - if sorted$(a_1,...,a_n)$ and sorted$(b_1,...,b_n)$ then merge$((a_1,...,a_n),(b_1,...,b_n))$ = sort$(a_1,...,a_n,b_1,...,b_n)$
- We're going to build a network, merge$_m$, that merges two sorted lists with 2^m elements
- m=0 m = 1





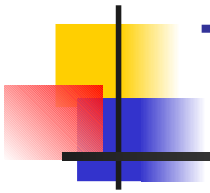
Courtesy of Henri Casanova

What about for m=3?



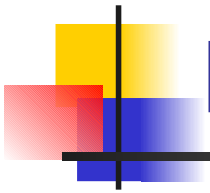
- Why does this work?
  - To build $merge_m$ one uses
    - 2 copies of the $merge_{m-1}$ network
    - 1 row of 2^m-1 comparators
  - The first copy of $merge_{m-1}$ merges the odd-indexed elements, the second copy merges the even-indexed elements
  - The row of comparators completes the global merge, which is quite a miracle really

Courtesy of Henri Casanova



Theorem to build $merge_m$

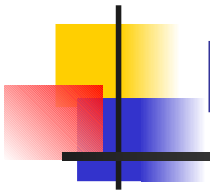
- Given sorted(a_1,...,a_{2n}) and sorted(b_1,...,b_{2n})
- Let
  - (d_1,...,d_{2n}) = merge((a_1,a_3,..,a_{2n-1}), (b_1,b_3,...,b_{2n-1})
  - (e_1,...,e_{2n}) = merge((a_2,a_4,..,a_{2n}), (b_2,b_4,...,b_{2n})
- Then
  - sorted(d_1,min(d_2,e_1),max(d_2,e_1),..., min(d_{2n},e_{2n-1}),max(d_{2n},e_{2n-1}),e_{2n})



Proof

- Assume all elements are distinct
- d_1 is indeed the first element, and e_{2n} is the last element of the global sorted list
- For i>1 and i <=2n, d_i and $e_{i-1,}$ must appear in the final list in position 2i-2 or 2i-1.
- Let's prove that they are at the right place
  - if each is larger than 2i-3 elements
  - if each is smaller than 4n-2i+1 elements
  - therefore each is either in 2i-2 or 2i-1
  - and the comparison between the two makes them each go in the correct place
- So we must show that
  - d_i is larger than 2i-3 elements
  - e_{i-1} is larger than 2i-3 elements
  - d_i is smaller than 4n-2i+1 elements
  - e_{i-1} is smaller than 4n-2i+1 elements

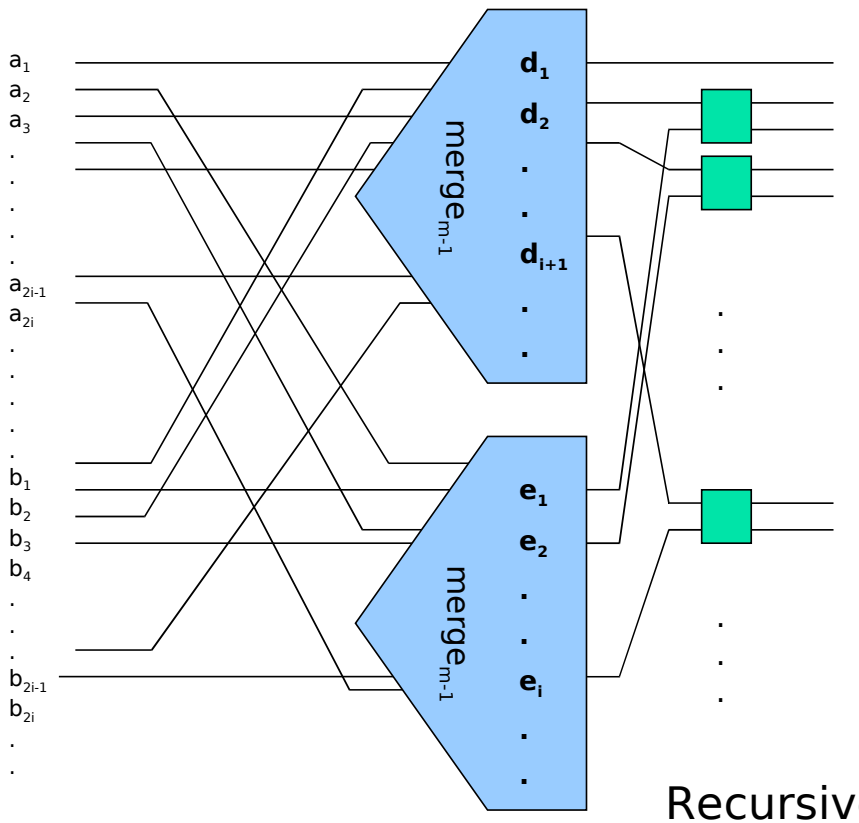
Courtesy of Henri Casanova



Proof (cont'ed)

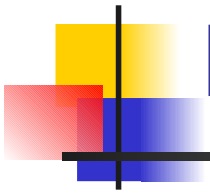
- d_i is larger than 2i-3 elements
  - Assume that d_i belongs to the $(a_j)_{j=1,2n}$ list
  - Let k be the number of elements in $\{d_1,d_2,...,d_i\}$ that belong to the $(a_j)_{j=1,2n}$ list
  - Then $d_i = a_{2k-1}$, and d_i is larger than 2k-2 elements of A
  - There are i-k elements from the $(b_j)_{j=1,2n}$ list in $\{d_1,d_2,...,d_{i-1}\}$, and thus the largest one is $b_{2(i-k)-1}$. Therefore d_i is larger than 2(i-k)-1 elements in list $(b_j)_{j=1,2n}$
  - Therefore, d_i is larger than 2k-2 + 2(i-k)-1 = 2i-3 elements
  - Similar proof if d_i belongs to the $(b_j)_{j=1,2n}$ list
- Similar proofs for the other 3 properties

Construction of merge_m



Recursive construction that implement the result from the theorem

Courtesy of Henri Casanova



Performance of merge_m

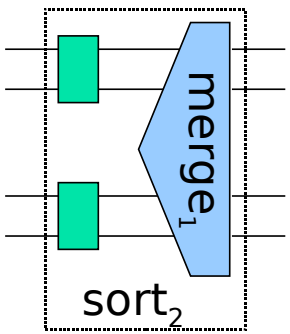
- Execution time is defined as the maximum number of comparators that one input must go through to produce output
  - t_m: time to go through merge_m
  - p_m: number of comparators in merge_m
- Two inductions
  - $t_0=1$, $t_1=2$, $t_m = t_{m-1} + 1$ $\quad (t_m = m+1)$
  - $p_0=1$, $p_1=3$, $p_m = 2p_{m-1} + 2^m - 1$ $\quad (p_m = 2^m m+1)$
  - Easily deduced from the theorem
- In terms of $n=2^m$, $O(\log n)$ and $O(n \log n)$
- Fast execution in $O(\log n)$
- But poor efficiency
  - Sequential time with one comparator: n
  - Efficiency = $n / (n * \log n * \log n) = 1 / (\log n)^2$
  - Comparators are not used efficiently as they are used only once
  - The network could be used in pipelined mode, processing series of lists, with all comparators used at each step, with one result available at each step.

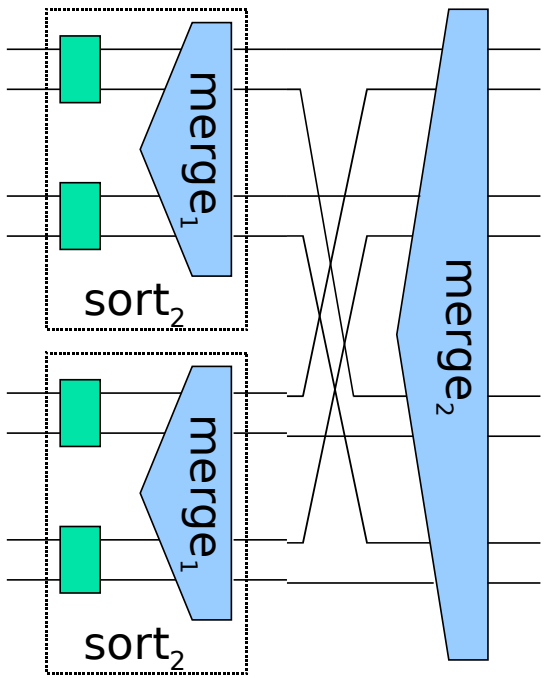
Courtesy of Henri Casanova

Sorting network using merge_m

- Sort_2 network

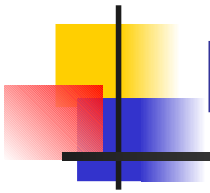
Sort_3 network





Sort 1st half of the list
Sort 2nd half of the list
Merge the results
Recursively

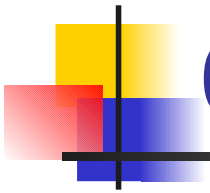
Courtesy of Henri Casanova



Performance

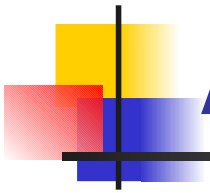
- Execution time t'_m and p'_m number of comparators
  - $t'_1=1 \quad t'_m = t'_{m-1} + t_{m-1} \quad (t'_m = O(m^2))$
  - $p'_1=1 \quad p'_m = 2p'_{m-1} + p_{m-1} \quad (p'_m = O(2^m m^2))$
- In terms of $n = 2^m$
  - Sort time: $O((\log n)^2)$
  - Number of comparators: $O(n(\log n)^2)$
- Poor performance given the number of comparators (unless used in pipeline mode)
  - Efficiency: $T_{seq} / (p * T_{par})$
  - Efficiency = $O(n \log n / (n (\log n)^4)) = O((\log n)^{-3})$
- There was a PRAM algorithm in $O(\log n)$
- Is there a sorting network that achieves this?
  - yes, recent work in 1983
  - $O(\log n)$ time, $O(n \log n)$ comparators
  - But constants are SO large, that it is impractical

Courtesy of Henri Casanova



0-1 Principle

- **Theorem:** A network of comparators implements sorting correctly if and only if it implements it correctly for lists that consist solely of 0's and 1's
- This theorem makes proofs of things like the "merge theorem" much simpler and in general one only works with lists of 0's and 1' when dealing with sorting networks



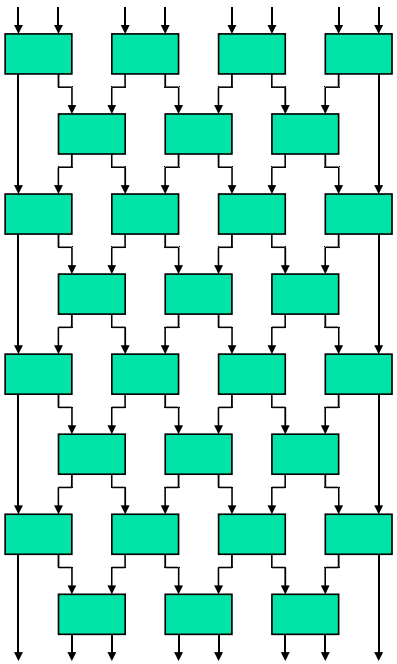
Another (simpler) sorting network

- Sort by even-odd transposition
- The network is built to sort a list of n=2p elements
  - p copies of a 2-row network
  - the first row contains p comparators that take elements 2i-1 and 2i, for i=1,...,p
  - the second row contains p-1 comparators that take elements 2i and 2i+1, for i=1,..,p-1
  - for a total of n(n-1)/2 comparators
  - similar construction for when n is odd

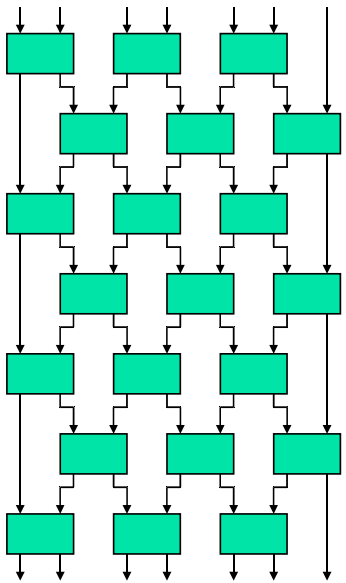
Courtesy of Henri Casanova

Odd-even transposition network

n = 8



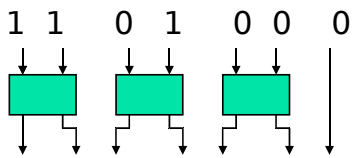
n = 7



Courtesy of Henri Casanova

Proof of correctness

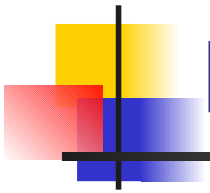
- To prove that the previous network sort correctly
  - rather complex induction
  - use of the 0-1 principle
- Let $(a_i)_{i=1,..,n}$ a list of 0's and 1's to sort
- Let k be the number of 1's in that list; j_0 the position of the last 1



$k = 3$
$j_0 = 4$

- Note that a 1 never "moves" to the left (this is why using the 0-1 principle makes this proof easy)
- Let's follow the last 1: If j_0 is even, no move, but move to the right at the next step. If j_0 is odd, then move to the right in the first step. In all cases, it will move to the right at the 2nd step, and for each step, until it reaches the n^{th} position. Since the last 1 is at least in position 2, it will reach position n in at least n-1 steps.

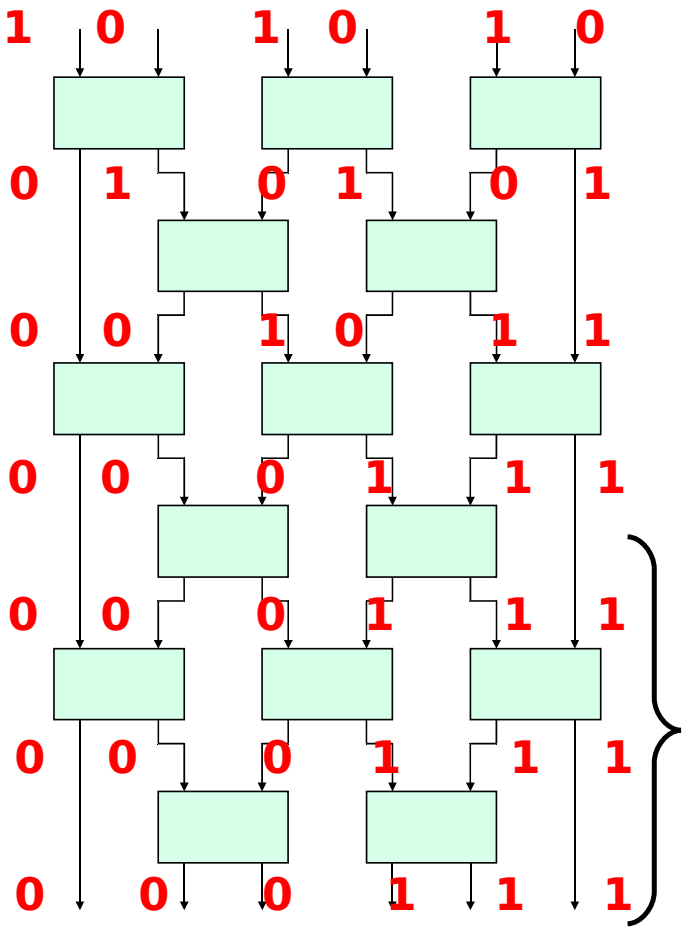
Courtesy of Henri Casanova



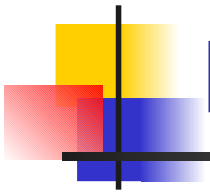
Proof (continued)

- Let's follow the next-to last 1, starting in position j: since the last 1 moves to the right starting in step 2 at the latest, the next-to-last 1 will never be "blocked". At step 3, and at all following steps, the next-to-last 1 will move to the right, to arrive in position n-1.
- Generally, the i^{th} 1, counting from the right, will move right during step i+1 and keep moving until it reaches position n-i+1
- This goes on up to the k^{th} 1, which goes to position n-k+1
- At the end we have the n-k 0's followed by the k 1's
- Therefore we have sorted the list

Example for n=6

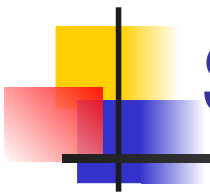


Courtesy of Henri Casanova



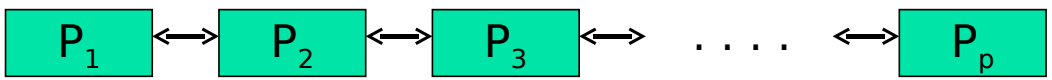
Performance

- Compute time: $t_n = n$
- # of comparators: $p_n = n(n-1)/2$
- Efficiency: $O(n \log n / n * (n-1)/2 * n) = O(\log n / n^2)$
  - Really, really, really poor
  - But at least it's a simple network
- Is there a sorting network with good and practical performance and efficiency?
  - Not really
  - But one can use the principle of a sorting network for coming up with a good algorithm on a linear network of processors

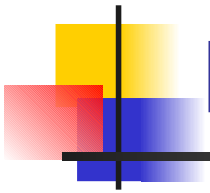


Sorting on a linear array

- Consider a linear array of p general-purpose processors



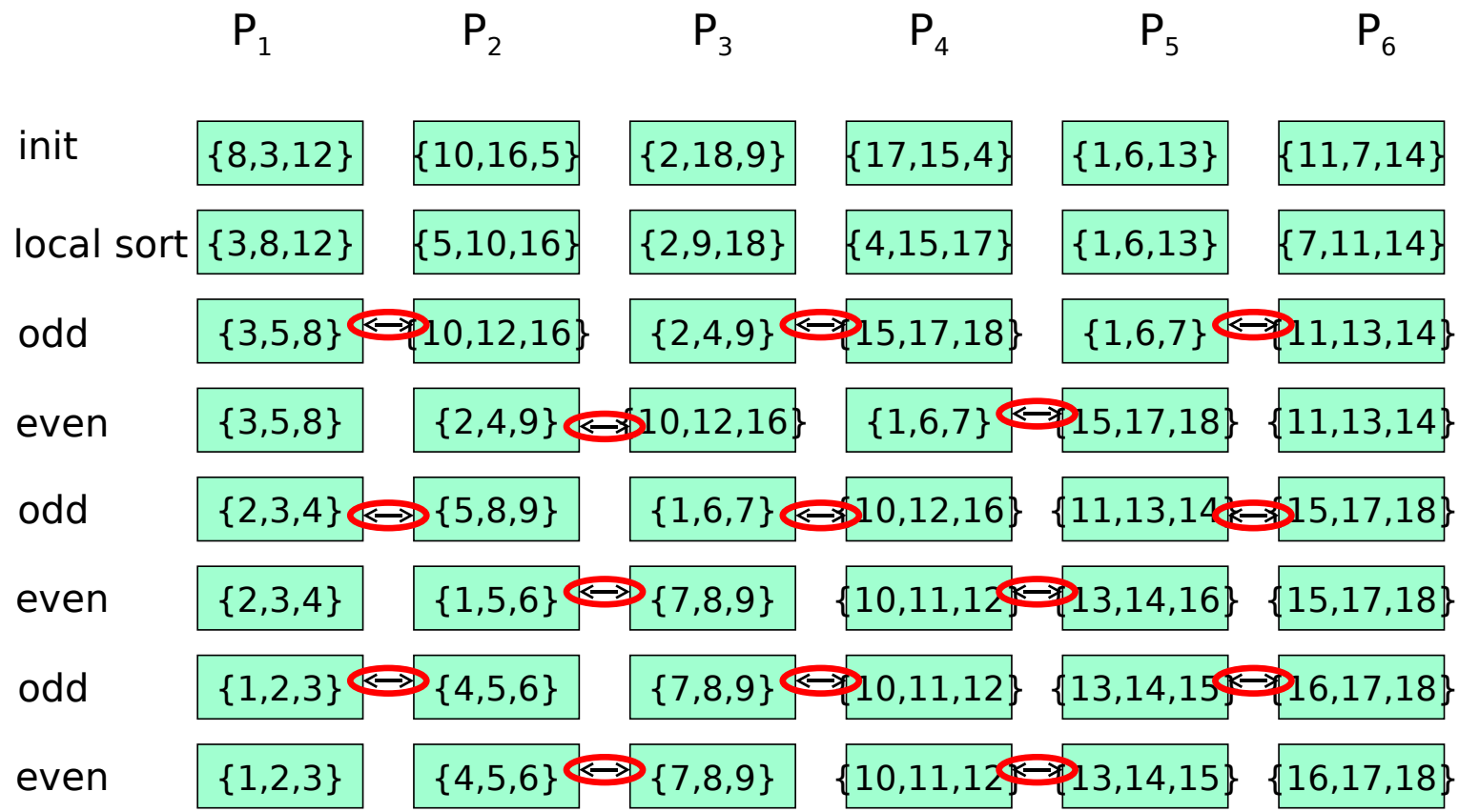
- Consider a list of n elements to sort (such that n is divisible by p for simplicity)
- Idea: use the odd-even transposition network and sort of "fold" it onto the linear array.



Principle

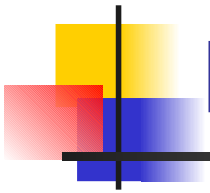
- Each processor receives a sub-part, i.e. n/p elements, of the list to sort
- Each processor sorts this list locally, in parallel.
- There are then p steps of alternating exchanges as in the odd-even transposition sorting network
  - exchanges are for full sub-lists, not just single elements
  - when two processors communicate, their two lists are merged
  - the left processor keeps the left half of the merged list
  - the right processor keeps the right half of the merged list

Example



Same pattern as the sorting network

Courtesy of Henri Casanova

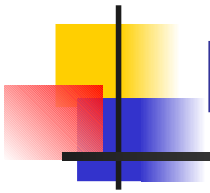


Performance

- Local sort: O(n/p * log n/p) = O(n/p * log n)
- Each step costs one merge of two lists of n/p elements: O(n/p)
- There are p such steps, hence: O(n)
- Total: O(n/p * log n + n)
- If p = log n: O(n)
- The algorithm is optimal for $p \leq \log n$

- More information on sorting networks: D. Knuth, The Art of Computer Programming, volume 3: Sorting and Searching, Addison-Wesley (1973)

Courtesy of Henri Casanova



FFT circuit - what's an FFT?

- Fourier Transform (FT): A "tool" to decompose a function into sinusoids of different frequencies, which sum to the original function
  - Useful is signal processing, linear system analysis, quantum physics, image processing, etc.
- Discrete Fourier Transform (DFT): Works on a discrete sample of function values
  - In many domains, nothing is truly continuous or continuously measured
- Fast Fourier Transform (FFT): an *algorithm* to compute a DFT, proposed initially by Tukey and Cole in 1965, which reduces the number of computation from $O(n^2)$ to $O(n \log n)$

Courtesy of Henri Casanova

How to compute a DFT

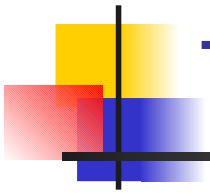
- Given a sequence of numbers $\{a_0, ..., a_{n-1}\}$, its DFT is defined as the sequence $\{b_0, ..., b_{n-1}\}$, where

$$b_j = \sum_{k=0}^{n-1} a_k \times \omega_n^{kj}$$ (polynomia eval)

with ω_n a primitive root of 1, i.e.,

$$\omega_n = e^{i\frac{2\pi}{n}}$$

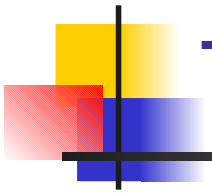
Courtesy of Henri Casanova



The FFT Algorithm

- A naive algorithm would require n^2 complex additions and multiplications, which is not practical as typically n is very large
- Let $n = 2^s$

$$b_j = \underbrace{\sum_{m=0}^{2^{s-1}-1} a_{2m}\omega_n^{2mj}}_{\text{even: } u_j} + \omega_n^j \underbrace{\sum_{m=0}^{2^{s-1}-1} a_{2m+1}\omega_n^{2mj}}_{\text{odd: } v_j}$$



The FFT Algorithm

- Therefore, evaluating the polynomial

$$a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1}$$

at $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$

can be reduced to:

1. Evaluate the two polynomials

$$a_0 + a_2 x + a_4 x^2 + ... + a_{n-2} x^{\frac{n}{2}-1}$$
$$a_1 + a_3 x + a_5 x^2 + ... + a_{n-1} x^{\frac{n}{2}-1}$$

at $(\omega_n^0)^2, (\omega_n^1)^2, ..., (\omega_n^{n-1})^2$

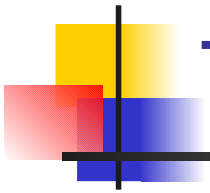
2. Compute

$$b_j = u_j + \omega_n^j v_j$$

BUT: $(\omega_n^0)^2, (\omega_n^1)^2, ..., (\omega_n^{n-1})^2$ contains really n/2 distinct elements!!!

$$(\omega_n^{n-1})^2 = e^{i\frac{4\pi n - 4\pi}{n}} = (e^{\frac{2\pi}{n}})^2 = (\omega_n^1)^2$$

Courtesy of Henri Casanova

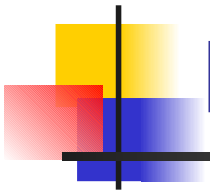


The FFT Algorithm

- As a result, the original problem of size n (that is, n polynomial evaluations), has been reduced to 2 problems of size n/2 (that is, n/2 polynomial evaluations)

```
FFT(in A,out B)
  if n = 1
    b_0 ← a_0
  else
    FFT(a_0, a_2,..., a_{n-2}, u_0, u_1,..., u_{(n/2)-1})
    FFT(a_1, a_3, ..., a_{n-1}, v_0, v_1,..., v_{(n/2)-1})
    for j = 0 to n-1
      b_j ← u_{j mod(n/2)} + ω_n^j . v_{j mod(n/2)}
    end for
  end if
```

Courtesy of Henri Casanova

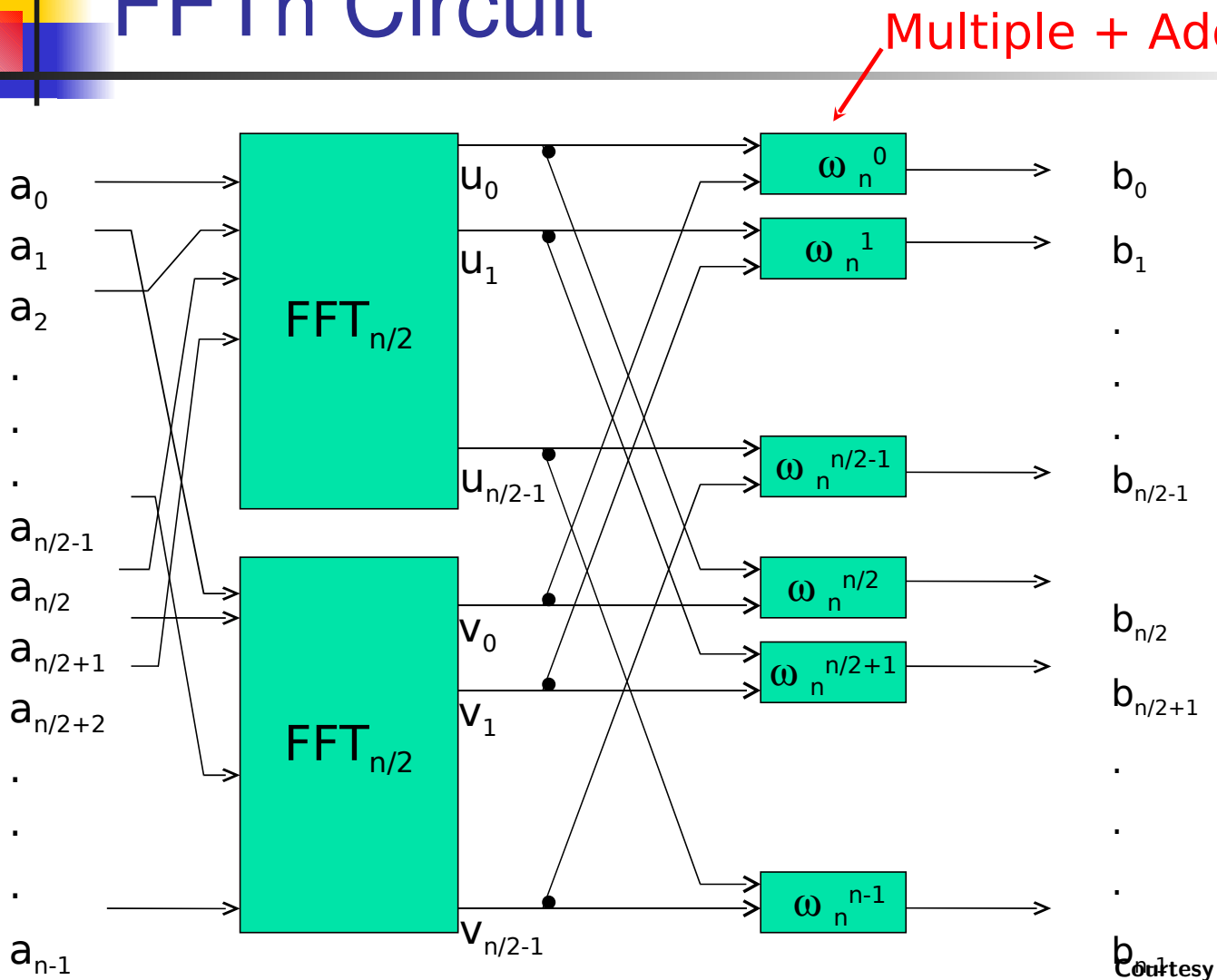


Performance of the FFT

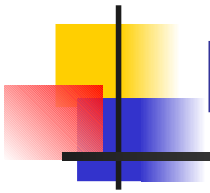
- t(n): running time of the algorithms
- t(n) = d * n + 2 * t(n/2), where d is some constant
- t(n) = O(n log n)
- How to do this in parallel?
  - Both recursive FFT computations can be done independently
  - Then all iterations of the **for** loop are also independent

Courtesy of Henri Casanova

FFTn Circuit



Courtesy of Henri Casanova

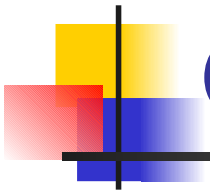


Performance

- Number of elements
  - width of O(n)
  - depth of O(log n)
  - therefore O(n log n)
- Running time
  - t(n) = t(n/2) + 1
  - therefore O(log n)
- Efficiency
  - 1/log n
- You can decide which part of this circuit should be mapped to a real parallel platform, for instance

Courtesy of Henri Casanova

Outline



Conclusion

- We could teach an entire semester of theoretical parallel computing
- Most people just happily ignore it
- But
  - it's the source of most fundamental ideas
  - it's a source of inspiration for algorithms
  - it's a source of inspiration for implementations: DSP

Courtesy of Henri Casanova