The PRAM Model

Alexandre David

1.2.05

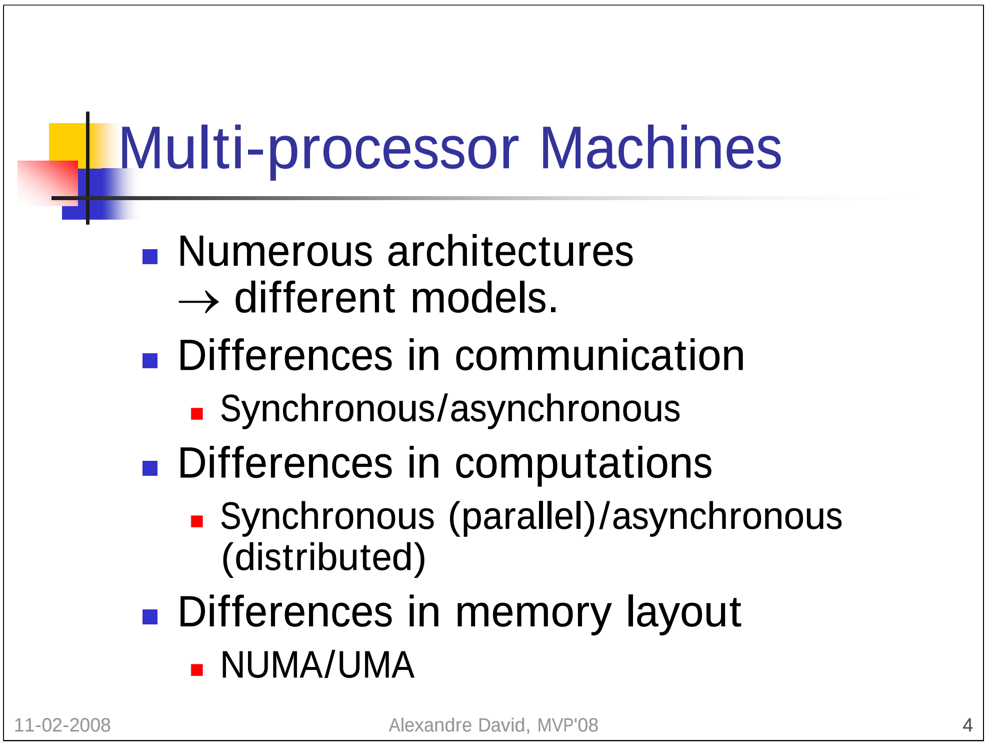
Outline

- Introduction to Parallel Algorithms (Sven Skyum)
  - PRAM model
  - Optimality
  - Examples

Standard RAM Model

- Standard **R**andom **A**ccess **M**achine:
  - Each operation
    load, store, jump, add, etc …
    takes one unit of time.
- Simple, generally one model.

The RAM is the basic machine model behind *sequential* algorithms.



Even if there are different architectures and models, the goal is to *abstract* from the hardware and have a model on which to reason and analyze algorithms. Synchronous vs. asynchronous communication is like *blocking* vs. *non-blocking* communication. NUMA is assumed most often when the model talks about local memory to a given processor.

Clusters of computers correspond to NUMA in practice. They are best suited for message passing type of communication.

Shared memory systems are easier from a programming model point of view but are more expensive.

PRAM Model

- A PRAM consists of
  - a *global* access *memory* (i.e. shared)
  - a set of *processors* running the same program (though not always), with a private *stack*.
- A PRAM is **synchronous.**
  - One global clock.
- Unlimited resources.

PRAM **model** – Parallel Random Access Machine.

In the report the stack is called accumulator.

Synchronous PRAM means that all processors follow a global clock (ideal model!). There is no direct or explicit communication between processors (such as message passing). Two processors communicate if one *reads* what another *writes*.

Unlimited resources means we are not limited by the size of the memory and the *number of processors* varies in function of the size of the problem, i.e., we have access to as many processors as we want. Designing algorithms for many processors is very fruitful in practice even with very few processors in practice whereas the opposite is limiting.

Classes of PRAM

- How to resolve *contention*? ?
  - EREW PRAM – exclusive read, exclusive write
  - CREW PRAM – concurrent read, exclusive write
  - ERCW PRAM – exclusive read, concurrent write
  - CRCW PRAM – concurrent read, concurrent write

? Most realistic?
Most convenient?

The most powerful model is of course CRCW where everything is allowed but that's the most unrealistic in practice too. The weakest model is EREW where concurrency is limited, closer to real architectures although still infeasible in practice (need *m*p* switches to connect *p* processors to *m* memory cells and provide exclusive access).

Exclusive read/write means access is serialized.

Main protocol to **resolve contention** (writing is the problem):

•Common: concurrent write allowed if the values are identical.

•Arbitrary: only an arbitrary processes succeeds.

•Priority: processes are ordered.

•Sum: the result is the sum of the values to be stored.

Exclusive write is exclusive with reads too.

Example: Sequential Max

```
Function smax(A,n)
    m := -∞
    for i := 1 to n do
        m := max{m,A[i]}
    od
    smax := m
end
```

Time $O(n)$

Sequential dependency, difficult to parallelize.

Simple algorithm description, independent from a given language. See your previous course on algorithms. O-notation used, check your previous course on algorithms too.

Highly sequential, difficult to parallelize.

Example: Sequential Max (bis)

```
Function smax2(A,n)
    for i := 1 to n/2 do
        B[i] := max{A[2i-1],A[2i]}
    od
    if n = 2 then
        smax2 := B[1]
    else
        smax2 := smax2(B,n/2)
    fi
end
```

Time *O(n)*

Dependency only between every call.

Remarks:

•Additional memory needed in this description

•B[i] *compresses* the array A[1..n] to B[1..n/2] with every element being the max of two elements from A (all elements are taken).

•The test serves to stop the recursive call – termination!

This is an example of the *compress and iterate* paradigm which leads to natural parallelizations. Here the computations in the for loop are independent and the recursive call tree gives the dependency between tasks to perform.

Example: Parallel Max

Time $O(\log n)$?

```
Function smax2(A,n) [p1,p2,…,pn/2]
    for i := 1 to n/2 pardo
        pi: B[i] := max{A[2i-1],A[2i]}
    od
    if n = 2 then
        p1: smax2 := B[1]
    else
        smax2 := smax2(B,n/2) [p1,p2,…,pn/4]
    fi
end
```

EREW-PRAM ?

EREW-PRAM algorithm.. Why? There is actually no contention and the dependencies are resolved by the recursive calls (when they return).

Here we give in brackets the processors used to solve the current problem.

Time $t(n)$ to execute the algorithms satisfies $t(n)=O(1)$ for $n=2$ and $t(n)=t(n/2)+O(1)$ for $n>2$. Why?

Think parallel and PRAM (all operations synchronized, same speed, p_i: operation in parallel). The loop is done in constant time on n/2 processors in parallel.

How many calls?

Answer: see your course on algorithms. Here simple recursion tree $\log n$ calls with constant time: $t(n)=O(\log n)$. **Note:** log base 2. You are expected to know a minimum about log.

Analysis of the Parallel Max

- Time: $O(\log n)$ for $n/2$ processors.
- *Work done?*
  - $p(n)=n/2$ number of processors.
  - $t(n)$ time to run the algorithm.
  - $w(n)=p(n)*t(n)$ work done.
    Here $w(n)=O(n \log n)$.

?

Is it optimal?

11-02-2008 Alexandre David, MVP'08 10

Work done corresponds to the actual amount of computation done (not exactly though). In general when we parallelize algorithms, the total amount of computations is greater than the original, but by a constant if we want to be optimal.

The work measures the time required to run the parallel algorithm on one processor that would simulate all the others.

Optimality

Definition

If $w(n)$ is of the **same order** as the time for the best known sequential algorithm, then the parallel algorithm is said to be **optimal**.

What about our previous example?

It's **not** optimal. Why? Well, we use only $n/2, n/4, \ldots, 2, 1$ processors, not n all the time!

We do not want to waste time like that right?

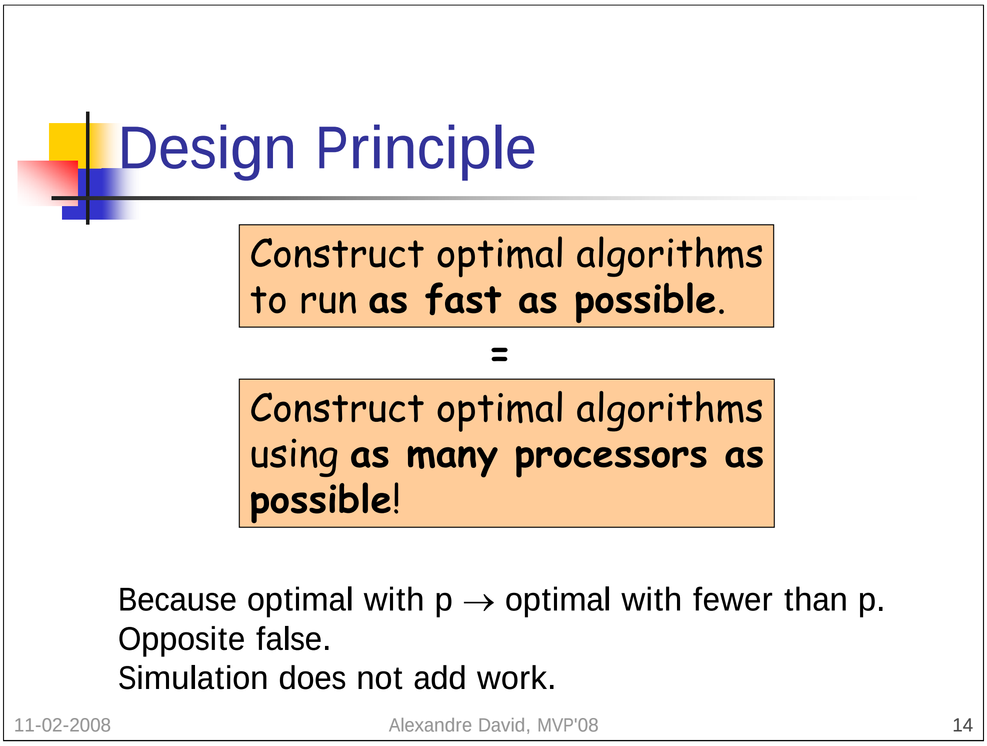
Another way to see it is that you get a speed-up linear to the number of processors (though at a constant factor, which means sub-linear).

Analysis of the Parallel Max

- Time: $O(\log n)$ for $n/2$ processors.
- *Work done?*
  - $p(n)=n/2$ number of processors.
  - $t(n)$ time to run the algorithm.
  - $w(n)=p(n)*t(n)$ work done.
    Here $w(n)=O(n \log n)$.
    Is it optimal? ***NO, O(n) to be optimal.***
    ? *Why?*

But...

Can a parallel algorithm solve a problem with **less** work than the best known sequential solution?



Note that if we have an optimal parallel algorithm running in time *t(n)* using *p(n)* processors then there exist optimal algorithms using *p'(n)<p(n)* processors running in time *O(t(n)*p(n)/p'(n))*. That means that **you can use fewer processors to simulate an optimal algorithm that is using many processors**! The goal is to maximize utilization of our processors. Simulating does not add work with respect to the parallel algorithm.

Brent's Scheduling Principle

Theorem

If a parallel computation consists of
- ***k* phases**
- taking time $t_1, t_2, \ldots, t_k$
- using $a_1, a_2, \ldots, a_k$ processors
- in phases $1, 2, \ldots, k$

then the computation can be done in time ***O(a/p+t)*** using ***p* processors** where $t = \mathrm{sum}(t_i)$, $a = \mathrm{sum}(a_i t_i)$.

What it means: same time as the original plus an overhead. If the number of processors increases then we decrease the overhead. The overhead corresponds to simulating the a_i with p. What it **really** means: It is possible to make algorithms optimal with the right amount of processors (provided that $t*p$ has the same order of magnitude of $t_{sequential}$). That gives you a bound on the number of needed processors.

It's a *scheduling* principle to reduce the number of physical processors needed by the algorithm and increase utilization. It does not do miracles.

Proof: i'th phase, p processors simulate a_i processors. Each of them simulate at most $\mathrm{ceil}(a_i/p) \leq a_i/p+1$, which consumes time t_i at a constant factor for each of them.

Previous Example

- k phases = log$n$.
- t_i = constant time.
- $a_i = n/2, n/4, \ldots, 1$ processors.
- With p processors we can use time $O(n/p + \log n)$.
- Choose $p=O(n/\log n) \rightarrow$ time $O(\log n)$ and this is **optimal**!

There is a "but": You need to know n in advance to schedule the computation.

Note: n is a power of 2 to simplify. Recall the definition of optimality to conclude that it is optimal indeed. This does not gives us an implementation, but almost.

Typo p6 "using $O(n/\log n)$ processors". Divide and conquer same as compress and iterate for the exercise.

Prefix Computations

Input: array A[1..n] of numbers.
Output: array B[1..n] such that B[k] = sum(i:1..k) A[i]
Sequential algorithm:

```
function prefix+(A,n)
    B[1] := A[1]
    for i = 2 to n do
        B[i] := B[i-1]+A[i]
    od
end
```

Time $O(n)$

Problem?

Prefix Computation

```
function prefix+(A,n)
        B[1] := A[1]
    if n > 1 then
        for i = 1 to n/2 pardo
                C[i]:=A[2i-1]+A[2i]
        od
        D:=prefix+(C,n/2)
        for i = 1 to n/2 pardo
                B[2i]:=D[i]
        od
        for i = 2 to n/2 pardo
                B[2i-1]:=D[i-1]+A[2i-1]
        od
    fi
    prefix+:=B
end
```


Correctness: When the recursive call of prefix$^+$ returns then D[k]=sum(i:1..2k) A[i] (for $1 \le k \le n/2$). That comes from the compression algorithm idea.

Parallel Prefix Computation

```
function prefix+(A,n)[p1,…,pn]
        p1: B[1] := A[1]
        if n > 1 then
                for i = 1 to n/2 pardo
                        pi: C[i]:=A[2i-1]+A[2i]
                od
                D:=prefix+(C,n/2)[p1,…,pn/2]
                for i = 1 to n/2 pardo
                        pi: B[2i]:=D[i]
                od
                for i = 2 to n/2 pardo
                        pi: B[2i-1]:=D[i-1]+A[2i-1]
                od
        fi
        prefix+:=B
end
```


Correctness: When the recursive call of $prefix^+$ returns then D[k]=sum(i:1..2k) A[i] (for $1 \le k \le n/2$). That comes from the compression algorithm idea.

Prefix Computations

- The point of this algorithm:
  - It works because + is associative (i.e. the compression works).
  - It will work for *any* other associative operations.
  - Brent's scheduling principle:

> For any associative operator computable in $O(1)$, its prefix is computable in $O(\log n)$ using $O(n/\log n)$ processors, which is optimal!

On a EREW-PRAM of course.

In particular initializing an array to a constant value…

Merging (of Sorted Arrays)

- Rank function:
  - rank(x,A,n) = 0 if x < A[1]
  - rank(x,A,n) = max{i | A[i] ≤ x}
  - Computable in time $O(\log n)$ by binary search.
- Merge A[1..n] and B[1..m] into C[1..n+m].
- Sequential algorithm in time $O(n+m)$.

Parallel Merge

```
function merge1(A,B,n,m)[p1,…,pn+m]
        for i = 1 to n pardo pi:
                IA[i] := rank(A[i]-1,B,m)
                C[i+IA[i]] := A[i]
        od
        for i = 1 to m pardo pi:
                IB[i] := rank(B[i],A,n)
                C[i+IB[i]] := B[i]
        od
        merge1 := C
end
```

?

CREW
Not optimal.

11-02-2008 Alexandre David, MVP'08 22

On CRCW-PRAM.

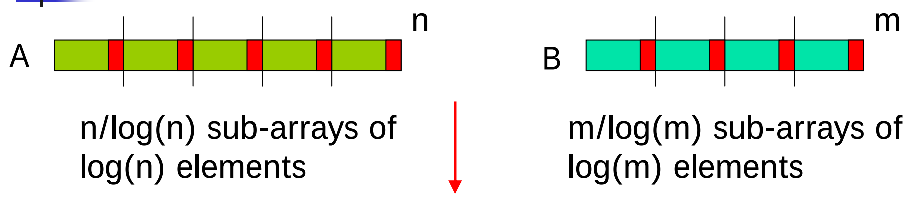
Compute indices for A[i] and compute indices for B[i] in parallel. Indices found by computing the rank of the elements. Dominating factor is the rank so this runs in *O(* log*(n+m))*. Not optimal, you see why?

However we could use processors p_{i+n} for the 2^{nd} loop (and we would have to rewrite this so that we have all processors doing something), which is not suggested by the report but it does not change much (we still have (n+m)*log(n+m)).

The more complicated version proposed in the report is optimal, which means it's possible to merge arrays optimally.

Being more careful here we see that it's actually CREW-PRAM. If it is CRCW then it would write fewer elements than n+m and it would be wrong.

Optimal Merge - Idea



n/log(n) sub-arrays of log(n) elements

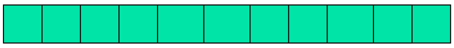
m/log(m) sub-arrays of log(m) elements

previous merge: n/log(n) + m/log(m) elements
costs max(log(n),log(m)) = O(log(n+m)),
(optimal) on (m+n)/log(n+m) processors!

Merge n/log(n)+m/log(m) lists with sequential merge in parallel.
Max length of sub-list is O(log(n+m)).

Example: Max in O(1)

- Max of an array in constant time!

A  *n* elements

1. Use *n* processors to initialize B.
2. Use n^2 processors to compare all A[i] & A[j].
3. Use *n* processors to find the max.

$$B[i]_{1 \le i \le n} = 0$$

$$A[i] > A[j] \Rightarrow B[j] = 1$$

$$B[i] = 0 \Rightarrow A[i]$$

Simulating CRCW on EREW

- Assumption on addressed memory $p(n)^c$ for some constant c.
- Simulation algorithm idea:
  - Sort accesses.
  - Give priority to 1st.
  - Broadcast result for contentious accesses.
- Conclusion: Optimality can be kept with EREW-PRAM when simulating a CRCW algorithm.

Read the details in the report. Remember the idea and the result.