PRAM Algorithms

Today´s menu

1. What do you program?
   - Parallel complexity and algorithms
2. The PRAM Model
   - Definition
   - Metrics and notations
   - Brent´s principle
   - A few simple algorithms & concepts
     - » Parallel sum and granularity control,
     - » Prefix computation and Divide & Conquer,
     - » Addition of two n-bits integers and Redundancy

What do you program?

- "Complexity" = metrics to evaluate the quality of your program.
- It depends on:
  - The model of the algorithm and of the program:
    - » Data, computation, memory usage, for instance.
  - The model of the host machine:
    - » Processor(s), memory hierarchy, network...?
- In the sequential case, everything´s fine!
  - Von Neumann model – fetch & run cycles.
  - Turing machine...
  - A very SIMPLE model enables the correct prediction and categorization of any algorithm.

What do you expect from a model?

- A model has to be:
  - extensive:
    - » Many parameters – in general, it ends up in a complx system.
    - » These parameters reflect the program/machine.
  - Abstract
    - » i.e. generic
    - » You do not want to change your model each 18 months (see Morore´s law)
    - » You want the general trend, not the details.
  - Predictive
    - » So it must lead to something you can calculate on.
    - » (It does not mean that it must be analytical)

In the parallel world...

- There is no universal model. ☹
- There are many models
  - For each type of machine, and many types of programs.
- You have no simple "reduction" from a model for distributed memory machine to a model for shared memory machine.
  - Because of the network model...
- Most theoretical models have been obtained with shared memory models.
  - Much simples, less parameters.
  - Scalability limited!

Basic ideas for the machine model

- Disconsider the communication (PRAM)
  - Adapted for shared memory machines, multicore chips...
- Consider a machine as being homegeneous, static, perfectly interconnected, with zero latency, and a fixed time for message passing (delay model).
- Consider a homogeneous, static machine, with a network that has latency and/or a given bandwith (LogP)
  - Okay for a cluster
- Considerar a dynamic, heterogeneous machine (Grid)...
  - No one knows hot to model this.

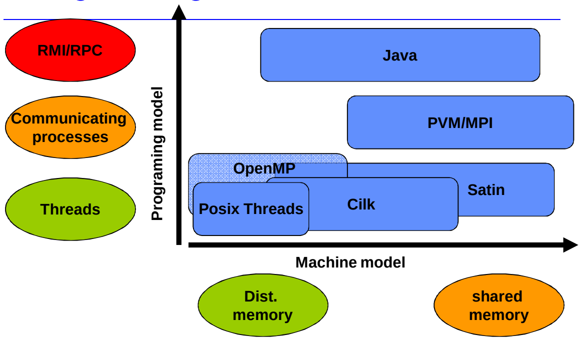
Parallel program model

- How do you describe a parallel program?
- Task parallelism:
  - The program is made of tasks (sequential unit);
  - The tasks are (partially) ordered by dependencies
  - A correct execution if a (possibly parallel) schedule which respects the dependencies.
  - Very close to functionnal programming.
- The dependencies can be explicit (e.g.: depend on messages) or implicit (e.g.: arguments).
- Trivial case: no dependencies ("embarassingly parallel" (EP), "parameter sweeping", "task farm", "master/slave", "client/server", "bag of tasks",...
- More complex case: Divide & Conquer.
  - The dependencies order the tasks in a tree-like way.

Parallel program model

- Data Parallelism
  - Distribute the data, and each process/thread computes on its local data.
    - » Owner Compute Rule
  - *Single Program Multiple Data*
- Loop parallelism
  - Comes from the Compiler world
  - Just tell which iterations of a loop can be performed in parallel.
- *Templates* for parallel programming
  - Provides skeletons (frameworks).

Programming Model vs. Machine Model



Performance evaluation

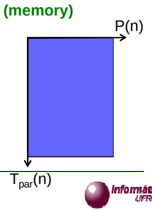
- A parallel program is intrinsically non-deterministic
  - The order of execution of the tass may change from execution to execution
  - The network (if any) adds its part of random.
- You are interested in runtime.
  - The usual argument "I compiled it, therefore the program is okay" does not serve at all!
- It is mandatory to use statistical measurements:
  - At least: x runs (x=10,20, 30...), and mean, min. and max. Runtime (or speedup, or efficiency) indicated.
  - Better: x runs, mean and standard deviation
    - » If the standard dev. is high, run it more – or ask yourself if there is something wrong...
  - Event better: x runs, confidence interval about the mean.

The PRAM model

- A PRAM machine is a set of processorS,
  - All are equal and only distinguished by an id.
  - All can access in constant time whatever address of a global, shared memory.
  - The processors execute their instructions synchronously.
  - You can use as many processor as you want.
- Metrics: executing a parallel program with entry of size n, on a PRAM machine, is characterized by two quantities:
  - The parallel runtime $T_{par}(n)$
  - The number of processors required to this execution P(n)
- The "quality" of the PRAM execution is also measured by $W_p(n)$ (Work), the total number of instructions.
  - $W_p(n) \leq T_{par}(n) * P(n)$

Considerations: time, space and work

- $T_{par}(n)$ is the runtime.
  - Proportional to the runtime of a single instruction.
  - What is important is the order of magnitude:
    - » O(n), O(log n), O(nlog n), θ(n)...
- P(n) is the number of processors.
  - In the PRAM model, 1 processor = 1 process.
  - P(n) can also be considered as a measure of the (memory) space that is required.
- $C(n) = T_{par}(n) \times P(n)$ is the (parallel) cost.
  - Look at it as a rectangular area.
- $W_p(n)$ is the Work
  - A sub-area of the rectangle.
  - As close as possible as the sequential program.

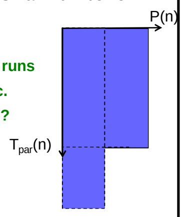


Optimal PRAM algorithm

1. $T_{par}(n)$ as small as possible
   – Maybe you will have to use many processors!
   – What is small?
     » $T_{par}(n) = θ(\log n)$
2. P(n) not too big.
   – Polynomial in n.
3. Do not perform (many) more instructions in parallel than in sequential.
   – i.e. $C(n) = θ(W_p(n)) = θ(W_1(n)) = θ(T_1(n))$

Brent´s Principle

- Na optimal PRAM algorithm will usually require P(n) processors, much more than a fixed, constant number p that is physically available ("p vs. n").
- So how do you map a PRAM algorithm to a small number of processors?
- Easy: just "adjust the rectangle"
  – Each physical processor i=1...p sequentially runs more than one instructions of each PRAM proc.
  – Of course, the runtime increases. How much?
- Brent´s principle (emulation):



$$T_p(n) \leq \frac{W_p(n)}{p} + T_{par}(n)$$

Great, but what does it mean?

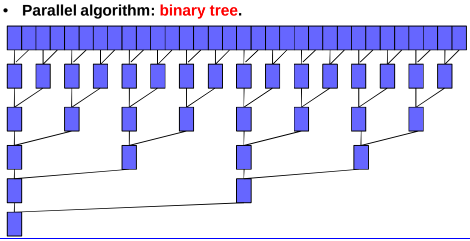
- Take an optimal PRAM algorithm
  – Very parallel, runtime much smaller than seq. and almost as few instr. as in the sequential case.
  – Formally, $T_{par}(n) = θ(\log n)$ and $W_p(n) = θ(T_1(n))$
- Therefore, you can always run it on a fixed, small number of processors p, with runtime:

$$T_p(n) \leq \frac{T_1(n)}{p} + \log(n)$$

- So, when $T_1(n) >> \log(n)$ (which is always the case...), you end up with an almost linear speedup. Nice!

1st PRAM algorithm: sum of n elements

- Input: an array of $n = 2^k$ elements and an associative, commutative operator '+'.
- Output: the "sum" of the n elements.
- $T_1(n) = n-1$ /* I do hope this is obvious for everyone... */
- Parallel algorithm: binary tree.



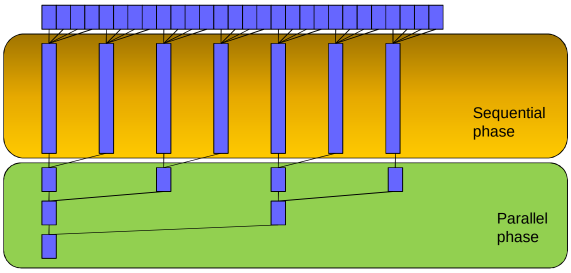
PRAM complexity of the parallel sum

1. $T_{par}(n) = θ(\log n)$
   – Good.
2. P(n) = n/2
   – That is a "small" polynomial in n. Good.
3. W(n) = n/2 + n/4 + n/8 + ... = n-1
   – $= T_1(n)$, great.
4. So what´s wrong ?
   – $C(n) = P(n) * T_{par}(n) = θ(n.\log n) >> θ(T_1(n))$
   – In plain English: the P(n) processors are under-used. The algorithm is inefficient.

The optimal parallel sum algorithm

- The problem comes from a very fine-grained algorithm.
  – The basic operation is a single + operation.
- Other version of the (same) problem: we use a little bit more processors than what we want.
  – If we could use P(n) = n/log(n), with $T_{par}(n) = θ(\log n)$, then the algorithm would be optimal.
- Solution: increase the granularity, or (same thing) use Brent´s principle.
  – Take p = n/log(n) processors, each one running more than one of the basic + instructions of previous algorithm.
  – Each processor will run log(n) '+' instructions.
- This idea can also be seen as a sequential "degeneration" of the parallel algorithm.
  – To be efficient in parallel, run in sequential!
  – A similar technique is used in sequential algorithmics (see quicksort)

The optimal parallel sum algorithm



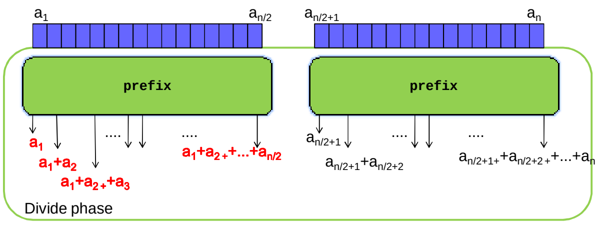
Parallel Prefix

- **Input: an array of n = 2^k elements and an associative, commutative operator '+'.**
- **Output: the n "partial sums" of the i first elements, i=1..n.**
- **$T_1(n)$ = n-1 /* I do hope this is obvious for everyone... */**

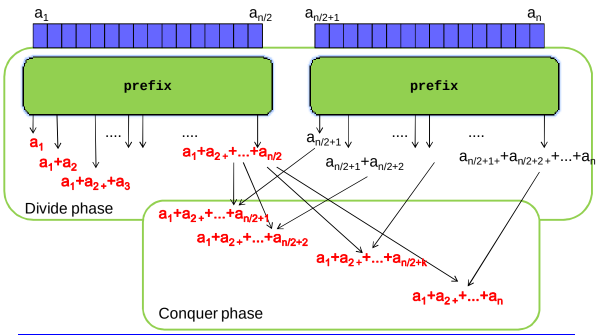
```
result[1] = a[1]   /* the 1st element of the input array */
for (i=2 ; i <= n ; i++)
   result[i] = result[i-1] + a[i]
```

- **This seems highly sequential!**
- **Let us revisit the computation, using Divide & Conquer**
  - This is an IMPORTANT (although simple) concept.

Prefix – the D&C version



Prefix – the D&C version

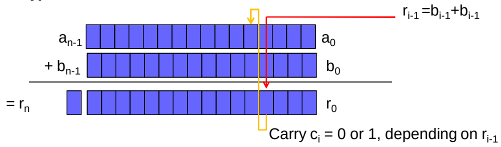


PRAM complexity of the D&C parallel prefix

1. **$T_{par}(n) = T_{par}(n/2) + 1 = ... = \theta(\log n)$**
   - Good.
2. **P(n) = Max{ 2P(n/2) ; n/2 } = ... = n**
   - That is a "small" polynomial in n. Good.
3. **$C(n) = P(n) * T_{par}(n) = \theta(n.\log n) >> \theta(T_1(n))$**
   - In plain English: the P(n) processors are under-used. The algorithm is inefficient.
4. **"Apply Brent" – or increase the granularity – and you will get an optimal version.**
   - $T_{par}(n) = \theta(\log n)$, $P(n) = n/\log(n)$.

Sum of two n-bits numbers.

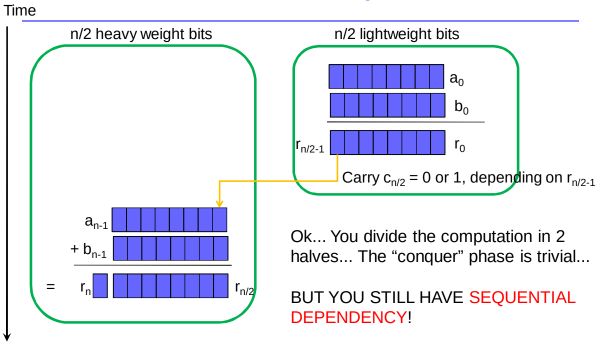
- **Input: 2 binary numbers a and b of n = 2^k bits.**
- **Output: the n+1 bits number equal to a+b.**
- **$T_1(n)$ = n, with the algorithm that is learned at elementary school (sum the digits column by column, from right to left, with the carry).**



Simple and nice, but highly sequential again!

- You have to propagate the carry from right to left, and can not compute the i-th bit without the carry.

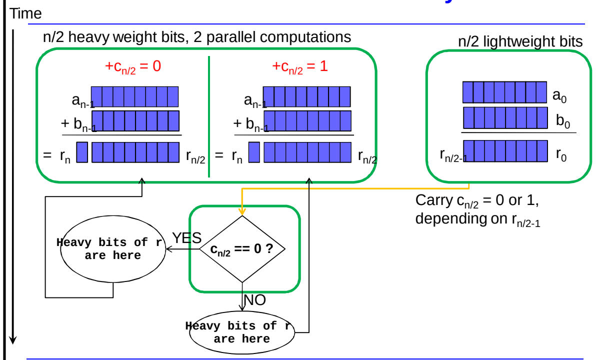
D&C sum of two binary numbers



Ok... You divide the computation in 2 halves... The "conquer" phase is trivial...

BUT YOU STILL HAVE SEQUENTIAL DEPENDENCY!

D&C sum with redundancy



PRAM complexity of D&C sum with redundancy

1. **$T_{par}(n) = T_{par}(n/2) + 1 = ... = \theta(\log n)$**
   - **Good.**
2. **$P(n) = Max\{ 3P(n/2) ; 1 \} = ... = \theta(n^{\log_2(3)}) = \theta(n^{1.58})$**
   - **That is a "small" polynomial in n. Good.**
3. **$C(n) = P(n) * T_{par}(n) = \theta(n^{1.58}.\log n) >> \theta(T_1(n))$**
   - **In plain English: the P(n) processors are under-used. The algorithm is inefficient.**
4. **The optimal algorithm is not obvious to obtain!**

Conclusion about PRAM

- **A simple, but powerful model**
  - **Quantifies the runtime and the processor number.**
  - **Evaluates the parallel number of operations.**
  - **Provides complexity classes (NC).**
- **An unrealistic model?**
  - **Use as many processors as you want**
    - » **This is an aproximation – "Brent resolves the problem".**
  - **Homogeneous architecture**
    - » **Ok...**
  - **Uniform time for all the memory accesses**
    - » **This is a real problem! What if there is some network activity in some place?**
- **Some say that the new area of shamred-memory systems (multicore) give a new force to PRAM.**