Introduzione programmazione // strutturata

CCP ‘09
Dip. Informatica di Pisa

M. Danelutto
Febbraio 2009

Patterns in parallel computations

- observe a large number of parallel/distributed applications
  - abstract the “features” related to parallelism exploitation
- a few “forms” can be recognized
  - at the proper abstraction level
  - sample forms:
    - pipelining: do one single thing in steps/stages
    - replication: do the same thing onto different data

Motivations

- the bad one ...
  - it’s easy to do things that way
    - we know how to implement them, there are existing implementations that we can cut&paste, ... it works
- the real one
  - there are a few ways of parallelising applications
    - much in the sense of what happens in seq world
      - there are a few constructs needed to program

The goal

- relieve programmers of:
  - programming the mechanisms needed to implement the pattern
  - deciding parameters used to fine tune the mechanisms
  - handle fault tolerance, security, load balancing, ...

- leaving in the hands of programmers:
  - functional parameters

Relieve programmers of mechanism handling

- “compile” patterns
  - to the middleware available
  - portability
  - correctness
  - tuning
- very important but yet not fundamental

Defining parameters needed to tune mechanisms

- much more important
  - parameter tuning requires
    - mechanism specific knowledge
    - architecture specific knowledge
  - it is not a monotonic process
    - try, eval, backtrack procedure

  - it is not really in the possibilities of normal application programmers

Even more important ...

- automatic / compiled implementation of
  - much more complicated (w.r.t. plain mechanism usage) policies
    - load balancing
      - requires specific hw & mw knowledge
    - security
      - requires specific knowledge on security techniques & application
    - fault tolerance
      - requires specific knowledge even on the programming system
- if in charge to programmers it boosts the amount of knowledge required

Sample “programmer view” :: Ocaml binding

- *let pipeline f g = function x -> (g (f x));;*
  val pipeline : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>

- *let rec farm f = function*
  [] -> [] | x :: rx -> (f x) :: (farm f rx);;
  val farm : ('a -> 'b) -> 'a list -> 'b list = <fun>

- let program =
  let firstStage = pipeline (farm blur) (farm smooth) in
  farm (pipeline firstStage sharpen);;
  val program : '_a -> '_a = <fun>

- let firstStage = pipeline (map blur) (map smooth) ;;
  val firstStage : '_a array -> '_a array = <fun>
  let program = farm (pipeline firstStage (farm sharpen));;
  val program : '_a array -> '_a array = <fun>

Let'us instantiate the concept: task farm

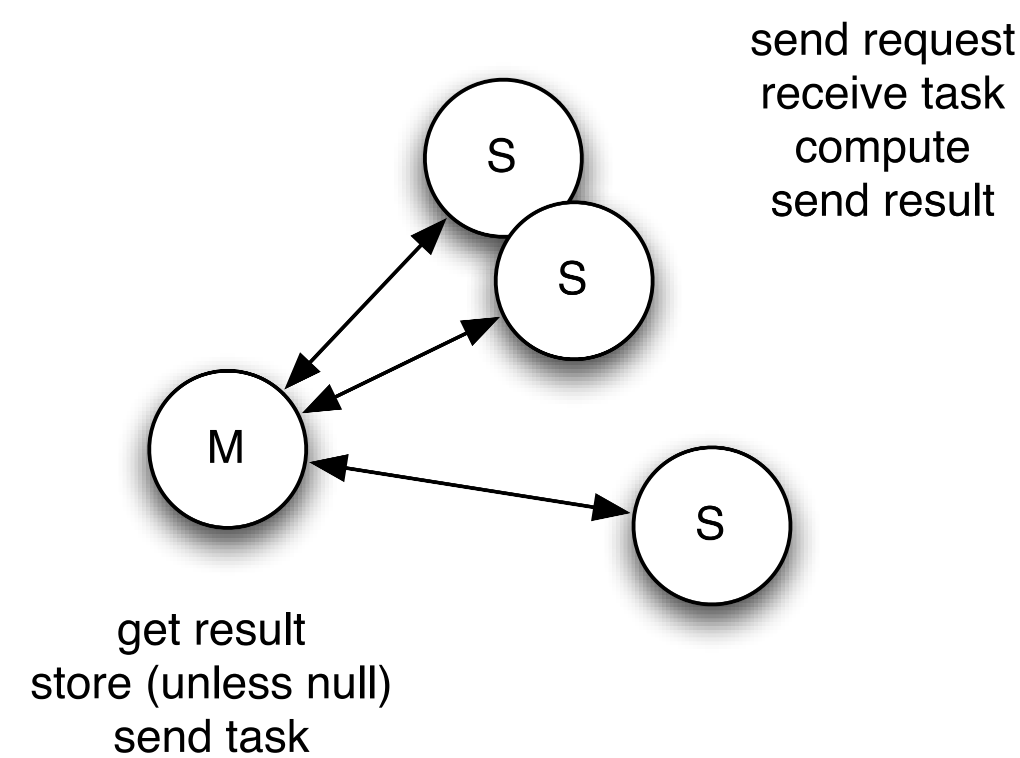
- task farm, master/slave, master/worker, embarrassingly parallel, independent task stream processing, ...
  - basically take x1 ... xn and compute f(x1) ... f(xn)
  - computation of xi is independent of the computation of any xj (j!=i)
- actually:
  - *embarrassingly parallel*, *task farm*
    - qualitative (kind of parallelism)
  - *master/slave*
    - implementation (kind of implementation schema)

Task farm: fundamental parameter(s)

- taskFarm: (‘a -> ‘b) -> ‘a stream -> ‘b stream
  - fundamental parameter is the function to be computed
- but also (in literature)
  - scheduling function: (‘a -> int)
    - violates the concept: pattern => indistinguishable workers
  - result reordering
    - violates the concept: pattern semantics
  - parallelism degree
    - violates the concept: architecture independence

Task farm: mechanisms

- e.g. exploit MPI
  - task request
    MPI_Send(...)
  - task receive
    MPI_Recv(...)
  - ...



  - communicator setup, id handling, message tag handling, ...
    can be completely hidden to the programmer

Task farm: policies

- Previous slide:
    - self scheduling
        - => load balancing
        - but also
            - fault tolerance: reschedule tasks failed on the (previous) remote node
    - reordering
        - enforced on the master (if any)

Task farm: policies

- Previous slide:
  - self scheduling
    - => load balancing
    - but also
      - fault tolerance: reschedule tasks failed on the (previous) remote node
  - reordering
    - enforced on the master (if any)

Alternative schedulings:
1) round robin (static/dynamic)
2) random
3) static partitioning proportional to BOGOMIPS
4) ...

“kinds of parallelism” (1)

- Sima, Fountain, Kacsuk “Advanced Computer Architecture: a design space approach” Addison Wesley, 1997:
  - “problem solutions may contain two different kinds of available parallelism, called functional parallelism and data parallelism. ... Data parallelism is regular, whereas functional parallelism, with the exception of loop-level paralllеism, is usually irregular. ... Regular parallelism is often massive, offering several orders of magnitude in speed up.
  - functional parallelism arises from the logic of problem a problem solution. It occurs in all formal descriptions of problem solutions, such as program flow diagrams, dataflow graphs, programs and so on to a greater or lesser extend
  - data parallelism comes from using data structures that allow parallel operations on their elements, such as vector and matrixes, in problem solutions

“kinds of parallelism” (2)

- Wilkinson, Allen “Parallel programming: techniques and applications using networked workstations and parallel computers” Prentice Hall 2005
  - embarrassingly parallel computations
    - a computation that can be diveded into a number of completely independent parts
  - partitioning and divide-and-conqer strategies
    - the problem is separeted into separate parts, and each part is computed separately
  - pipelined computations
    - wide range of problems that are partially sequential in nature; that is a sequence of steps must be undertaken
  - synchronous computations
    - all the process are synchronized at regular points, Generally, the same computation or operation is applied to a set of data points

“kind of parallelism” (3)

- each research area has its own classification
  - design patterns
  - algorithmical skeletons
  - coordination languages
  - grid computing
  - OO/agent/... distributed computing world

- “raise the level of abstraction” ...
  - and get the previous forms (more or less)

“kind of parallelism” (4, the Pisa way)

- pipeline, farm (stream parallelism)
- map, reduce, scan, parmod, ... (data parallelism)
- “embarrassingly” is orthogonal
- “data parallel” is orthogonal
  - data parallel :: unpack->farm->pack :: pipeline(seq,farm,seq)
- several different ways of implementing
  - templates, highly parametric templates, macro data flow, normal form
  - not significant (w.r.t. classification) but often directly related to ...

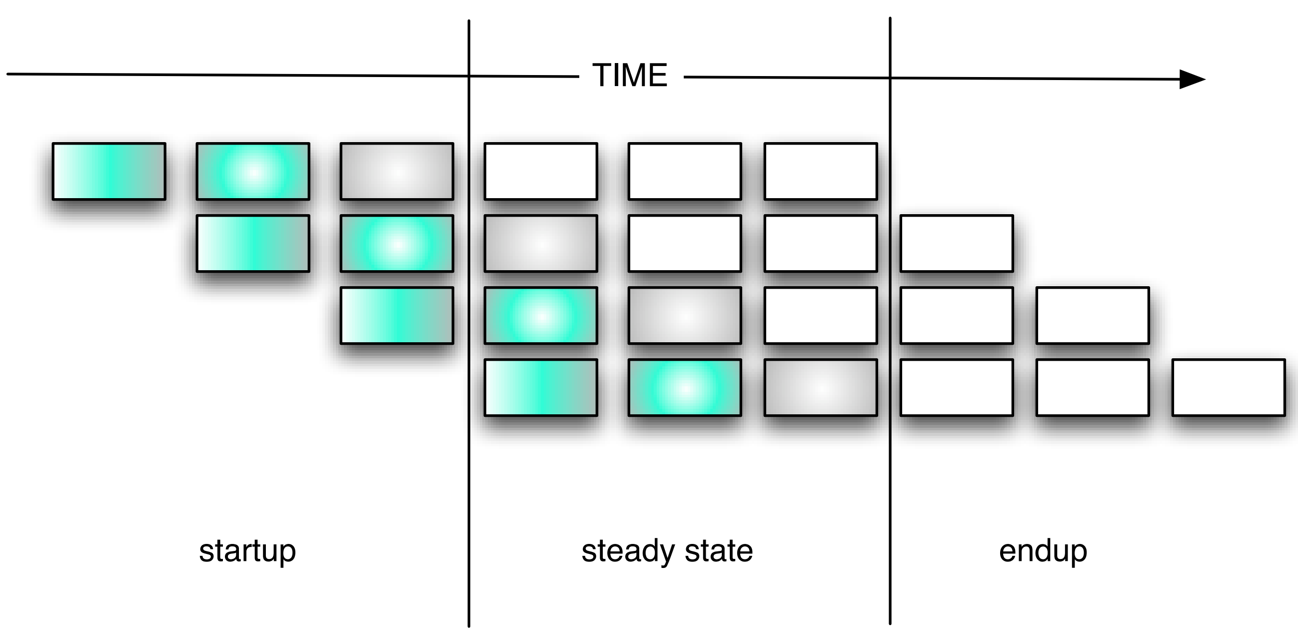
Common patterns

- pipeline
  - stages
- task farm (map)
  - embarrassingly stream (data) parallel computations
- divide & conquer (branch&bound)
  - dynamic data parallel

- plus arbitrary combinations of these form
  - with proper (code) parameters

Pipeline

- fn(... f1(f0(x)) ...)
  - computed on a stream of input data x1 ... xm
- THEN
  - compute in parallel (stages)

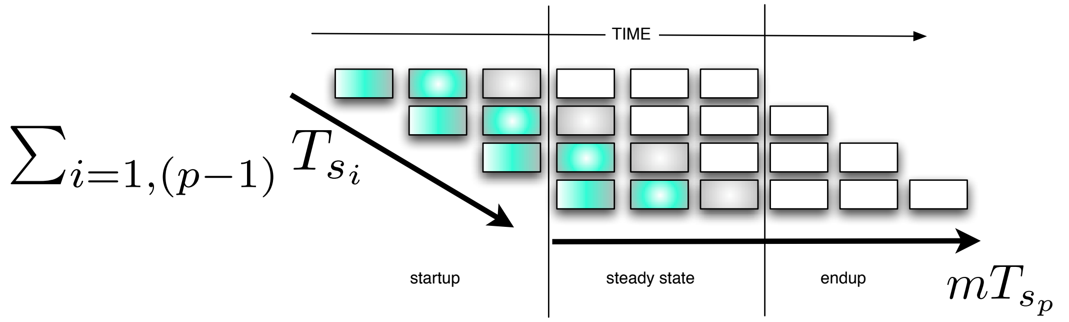


Performance (expected)

- Service time
  - $T_s = max\{T_{s_i}\}$
- Completion time
  - $T_c = \sum_{i=1,(p-1)} T_{s_i} + mT_{s_p}$
  - $T_c \approx mT_s$

Performance (expected)

- Service time
  - $T_s = max\{T_{s_i}\}$
- Completion time
  - $T_c = \sum_{i=1,(p-1)} T_{s_i} + mT_{s_p}$
  - $T_c \approx mT_s$



Speedup

- perfect speedup in theory

$$T_{seq} \approx m(pT_s)$$
$$T_{par} \approx m(T_s)$$
$$speedup(p) = \frac{mpT_s}{mT_s} = p$$

- without taking in too much precise account the comms ... :

$$\frac{mpT_s}{(p+m)T_s} = \frac{mp}{p+m} = \frac{p}{\frac{p}{m}+1} = \frac{p}{1+\frac{p}{m}}$$

overhead

- the more the tasks the smaller the overhead

Taking into account communications ...

- the upper term actually takes into account *grain* (it is computational, it depends on the kind of (code) parameters to the pattern)

$$\frac{m p t_p}{(p+m)(t_p + t_{comm})} = \dots = \frac{p\left(\frac{1}{1 + t_{comm}/t_p}\right)}{1 + p/m}$$

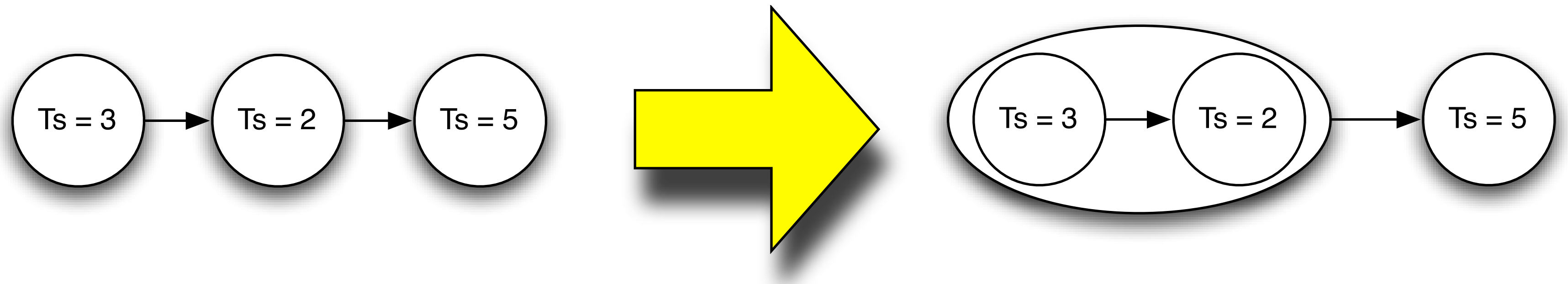
← lowers upper term

↖ makes higher the lower one!

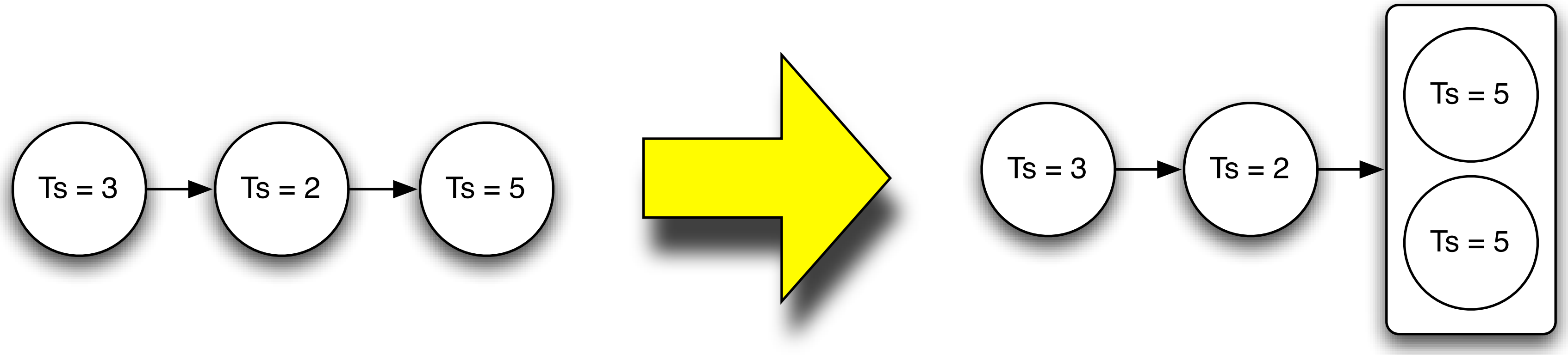
- the lower one takes into account startup overhead (it is structural, it depends on the parallelism exploitation

Optimizations

- unbalanced pipeline
  - group stages to match service times



  - parallelize stages to match service times

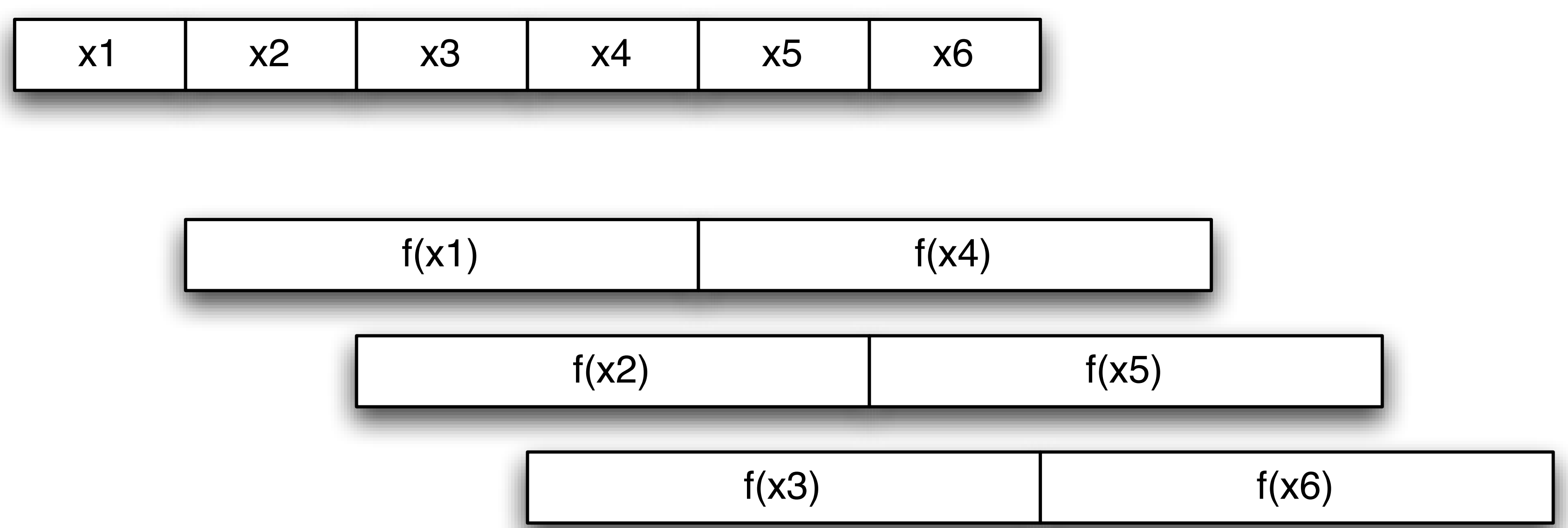


Optimizations (2)

- Possible only because
  - pipeline is a “declared” pattern
  - what if it was an MPI program ?
    - remember: mix functional and non functional code, primitives, etc.
- very simple in the pipeline case
  - due to simple performance model
  - using other patterns it will be much more hard !

Task farm

- stream of tasks x1 ... xn
    - possibly available at different times
        - Tin : interarrival time
- x1 ... xn -> f(x1) ... f(xn)
-



Performance (expected)

- provided we have enough resources
- service time

$$T_s = \frac{T_w}{p}$$

$$T_s = \max\left\{T_{in}, \frac{T_w}{p}\right\}$$

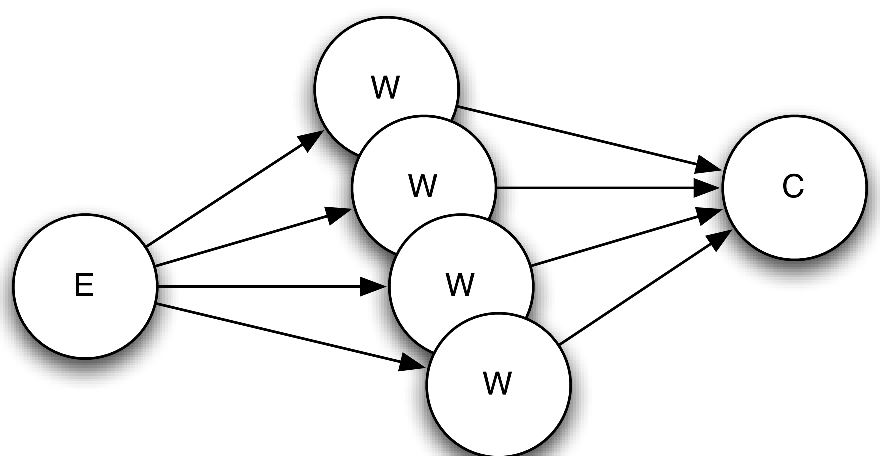
- completion time

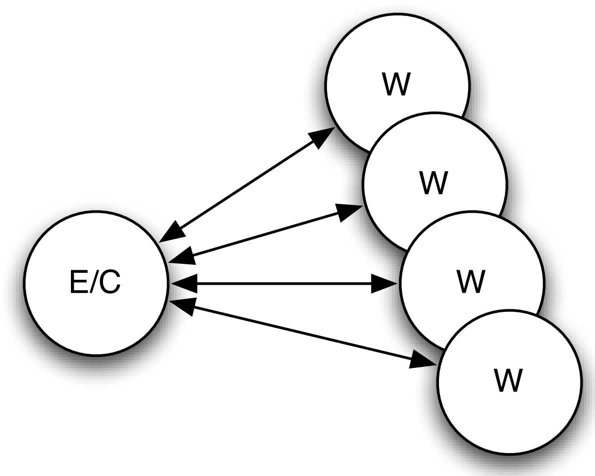
$$T_c = p\,T_{in} + \frac{m}{p} T_w \approx$$

$$\approx \frac{m}{p} T_w$$

Inter-arrival time and Master/Emitter/Manager ...

- task farm usually implemented with emitter and collector processes
  - sending tasks to workers, gathering (reordering) results
  - possibly implemented on the very same node



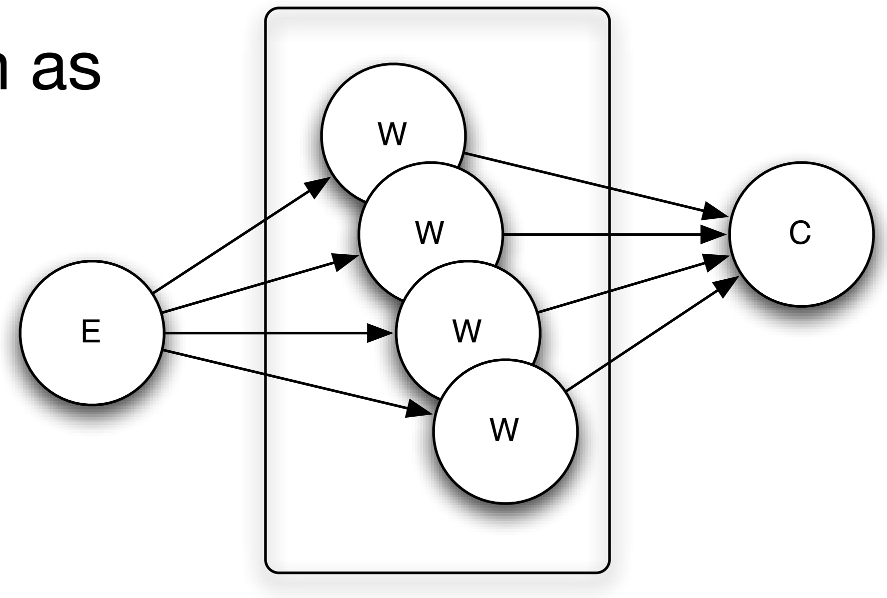


- Emitter, collector time is the bottleneck, usually

$$T_S = \max\left\{T_{in}, T_e, T_c, \frac{T_w}{p}\right\}$$

Farm analysis as a pipeline

- in case we implement farm as



- this is basically a three stage pipeline
  - must match service times of emitter, string of workers, collector
  - in case
    - parallelize the emitter and/or the collector
    - (tree structured E/C ...)

Speedup

- Speedup
  - linear unless overhead term
- overhead
  - decreases with the amount of tasks
  - decreases with the grain
  - increases with p

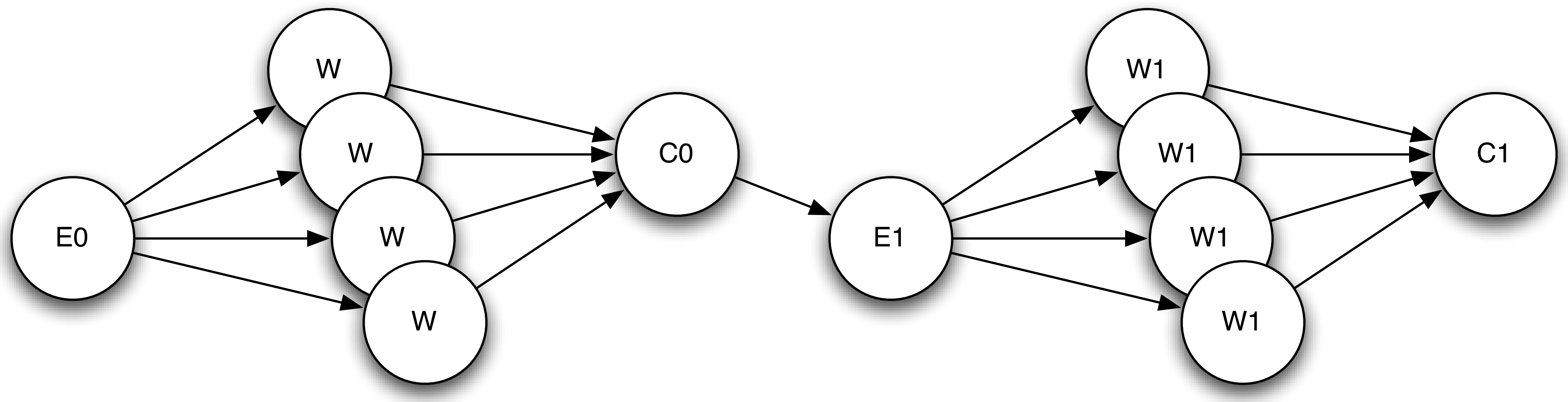
$$p \cdot t_e + \frac{m}{p} t_w \quad \text{vs.} \quad m t_w$$

$$\frac{m t_w}{p \cdot t_e + \frac{m}{p} t_w} = \frac{p t_w}{\frac{p^2}{m} t_e + t_w} =$$

$$= \frac{p}{1 + \frac{p^2}{m} \frac{t_e}{t_w}} = \frac{p}{1 + \frac{p^2}{m g}}$$

Optimizations (1)

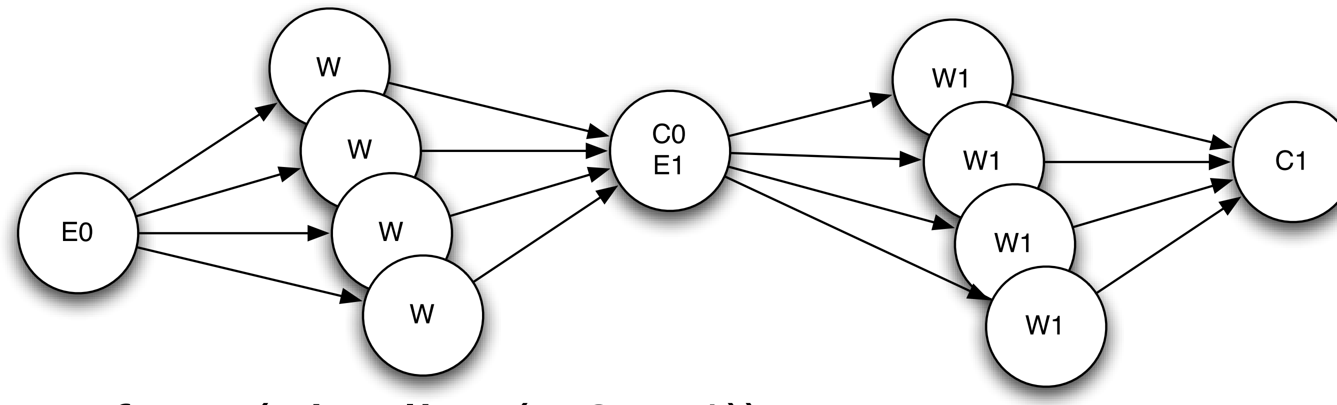
- reordering in sequences of farms ...
  - can be postponed up to the moment we give answers to user(s)

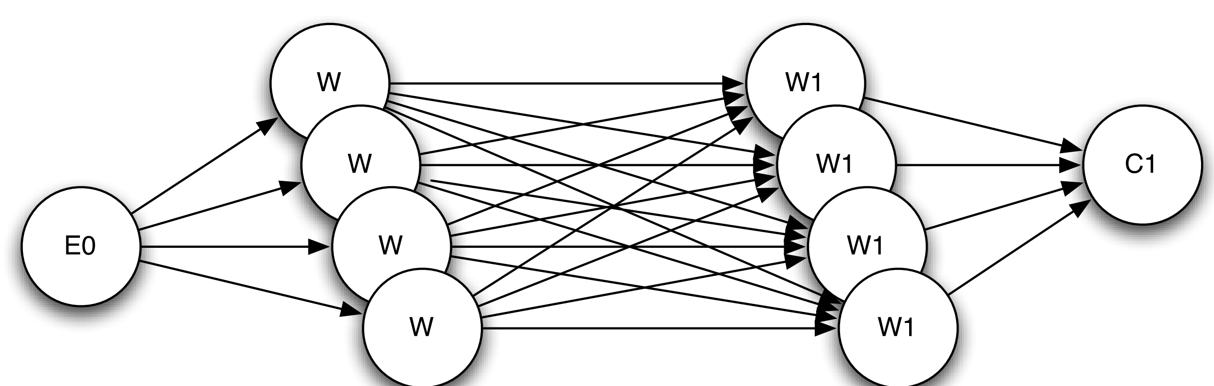


  - reordering just on C1
  - and optimize C0-E1
    - collapse

Optimizations (2)

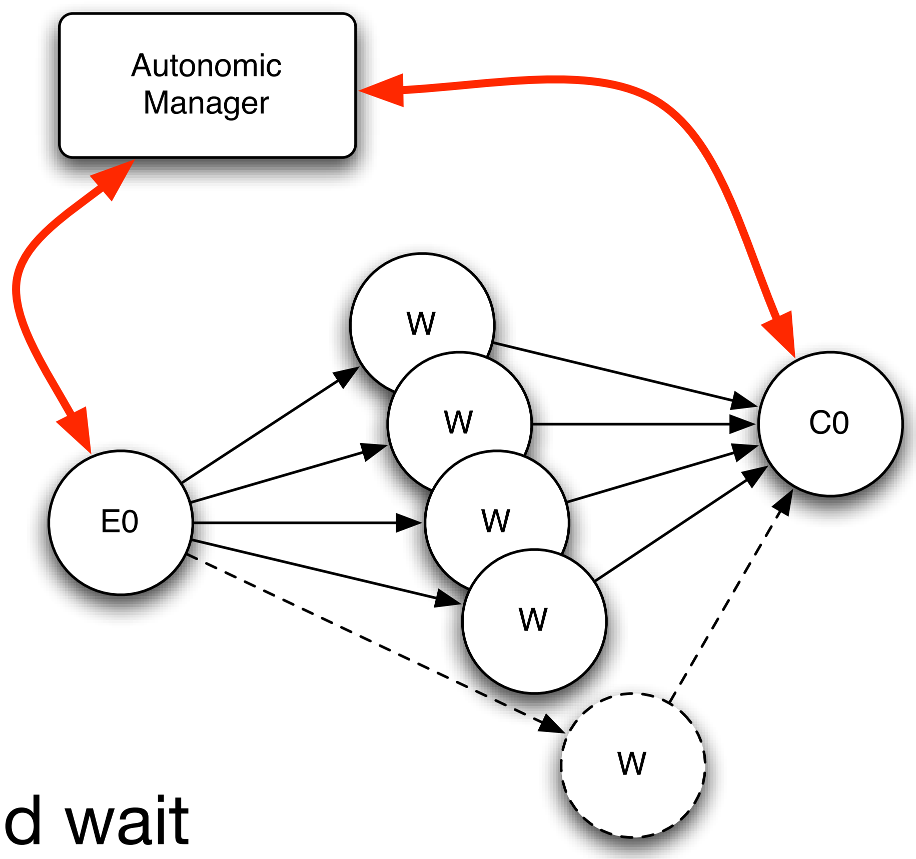
- What about removing E0-C0 ?
  - same number of workers:
    - pipeline(farm(w0), farm(w1)) -> farm(pipeline(w0,w1))
    - parallelism exploitation pattern changes !
  - different number of workers:
    - it depends on scheduling on the second farm:
      - can it be implemented in a distributed way ?





Optimizations (3)

- Fairly simple model
  - more and more workers give more and more performance
  - provided we do not saturate “emitter” bandwidth
- simple “autonomic” manager
  - look at the service time (measured on collector)
  - periodically:
    - add a new worker
    - if service time does not increase, release it and wait



Optimizations (general)

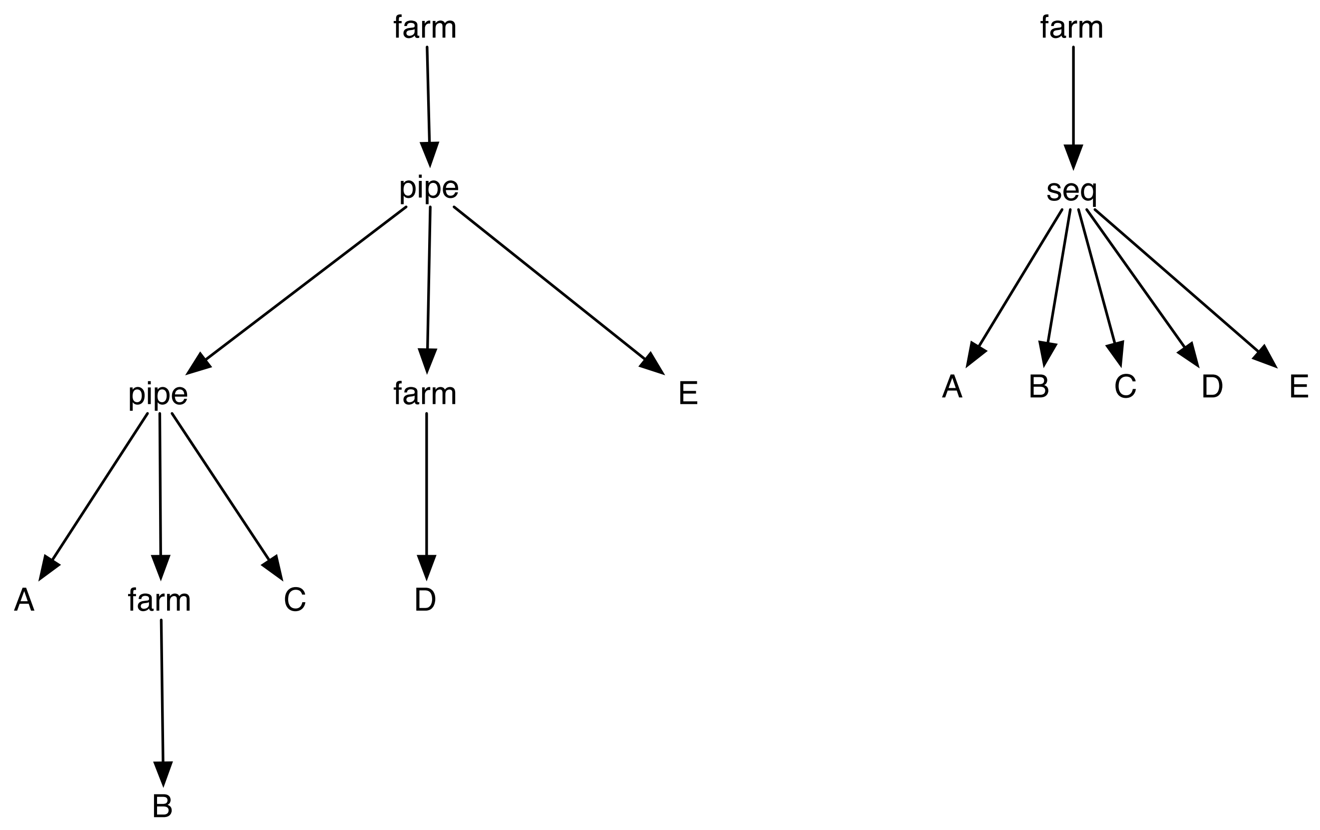
- Again
  - possibile because we declare the pattern
  - actual master/slave code not manageable that way
    - try with MPI !
- Some more subtle things
  - pattern composition transformations:
    - hold independently of the implementation
    - can be proven formally (at a suitable abstraction level)

Optimizations: rewriting

- farm f = f
- pipe f g = function x -> g(f x)
- pipe(farm(f), farm(g)) =
  pipe(f,g) =
  farm(pipe(f,g))
- *not taking into account* non functional features (i.e. parallel execution)
- has implications on the implementation
  - amount of communications / sharing
  - synchronization
  - ...

Optimization: normal form

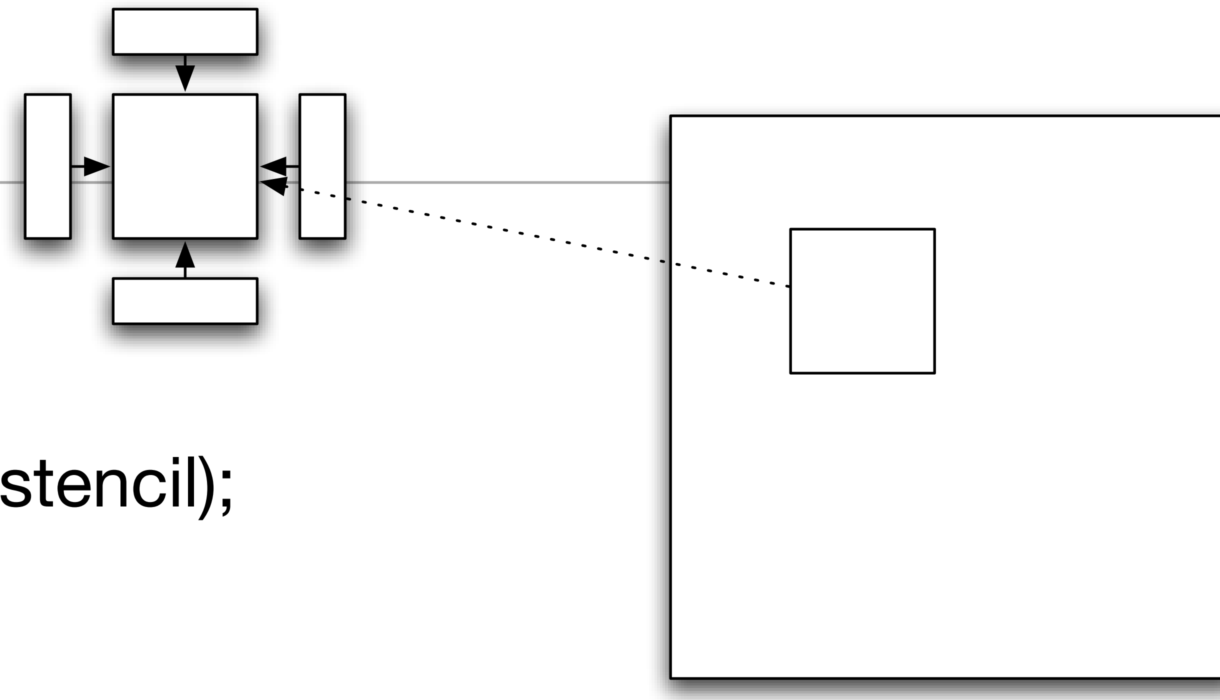
- an arbitrary stream parallel composition (pipeline and farm) can be transformed into a farm of sequential composition, with a better or equal performance
    - save comm times !



Data parallel ?

- most of the task farm stuff inherited
  - simple case: embarrassingly (data) parallel
    - emitter unpacks and distributes partitioned data
    - collector collects and packs computed result(s)
    - scatter/gather collectives needed
      - point-to-point do not scale

Data parallel: stencil



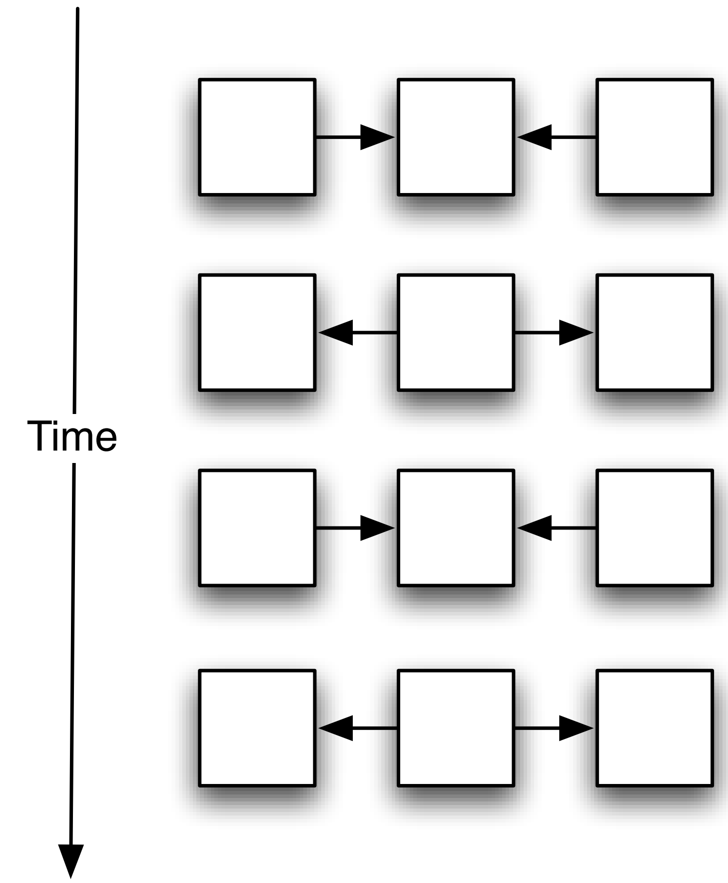
- while (cond)
  forall data partition
    data partition = f(data partition, stencil);
- cond may be a reduce
- stencil may be fixed or variable
- very popular pattern
  - differential equations mostly approximated/solved this way
- further comms among the workers
  - synchronous steps corresponding to iteration

Performance

- more sophisticated models
  - needed to exploit intra task parallelism
    - primary source for speedup in data parallel case
  - needed to take into account stream parallelism
    - additional source for speedup

Iteration problem

- sample vector stencil : xi = f(xi-1, xi, xi+1)
  - implementation in steps
    - “even” processors
      - step 2k: get values
      - step 2k+1: send value
    - “odd” processors
      - step 2k: send
      - step 2k+1: receive



- what if xi computes faster than xi-1 ?
  - synchronization needed or pure tagged data flow implementation

State

- up to now
  - stateless computations
- what about stateful one?
  - most of the models do not apply any more
    - scheduling (auto scheduling vs. state-care one)
    - performance models (accesses to shared state)

State (2)

- Same methodology
  - with more complicated issues

  - e.g. stateful farm
    - workers accumulate partial results in a logically shared variable
    - let rec farm f s = function x :: rx -> (f s x) :: (farm f (op s x) rx)
    - op associative and commutative
      - each worker accumulates its own op contribution
      - “summed” op at the end of the computation

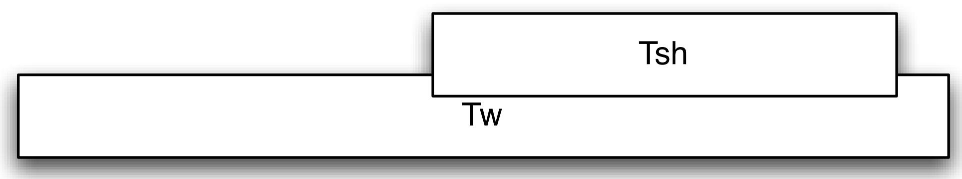
State (3)

- this was the easy case, now the “bad guy”
- e.g. farm with true shared var
- let rec farm f s = function x::rx -> (f x s)::(farm f (f’ s x) rx)
  - f computes the result on the stream
  - f’ computes a new state

- what about scheduling several tasks in parallel ?
- how can they access the state in a safe way ?
- what about performance model ?

Performance model (the bad case)

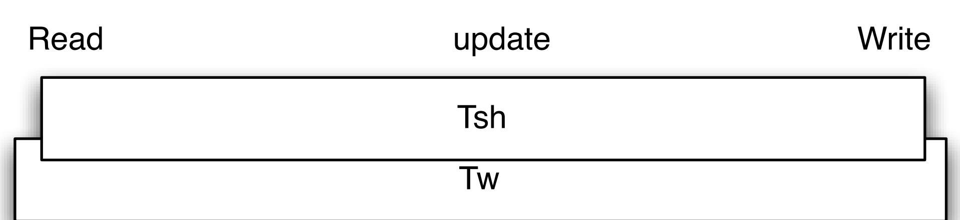
- each worker accesses shared variable
  - takes Tsh to process (read, use, update) out of total Tw
  - partially overlapped Tsh / Tw



    - amdhal law : Tsh is the serial fraction => maximum speedup is

$$\lim_{n \to \infty} speedup(n) = \frac{1}{f} = \frac{1}{T_{sh}/T_w} = \frac{T_w}{T_{sh}}$$

  - more usually:



Divide&conquer

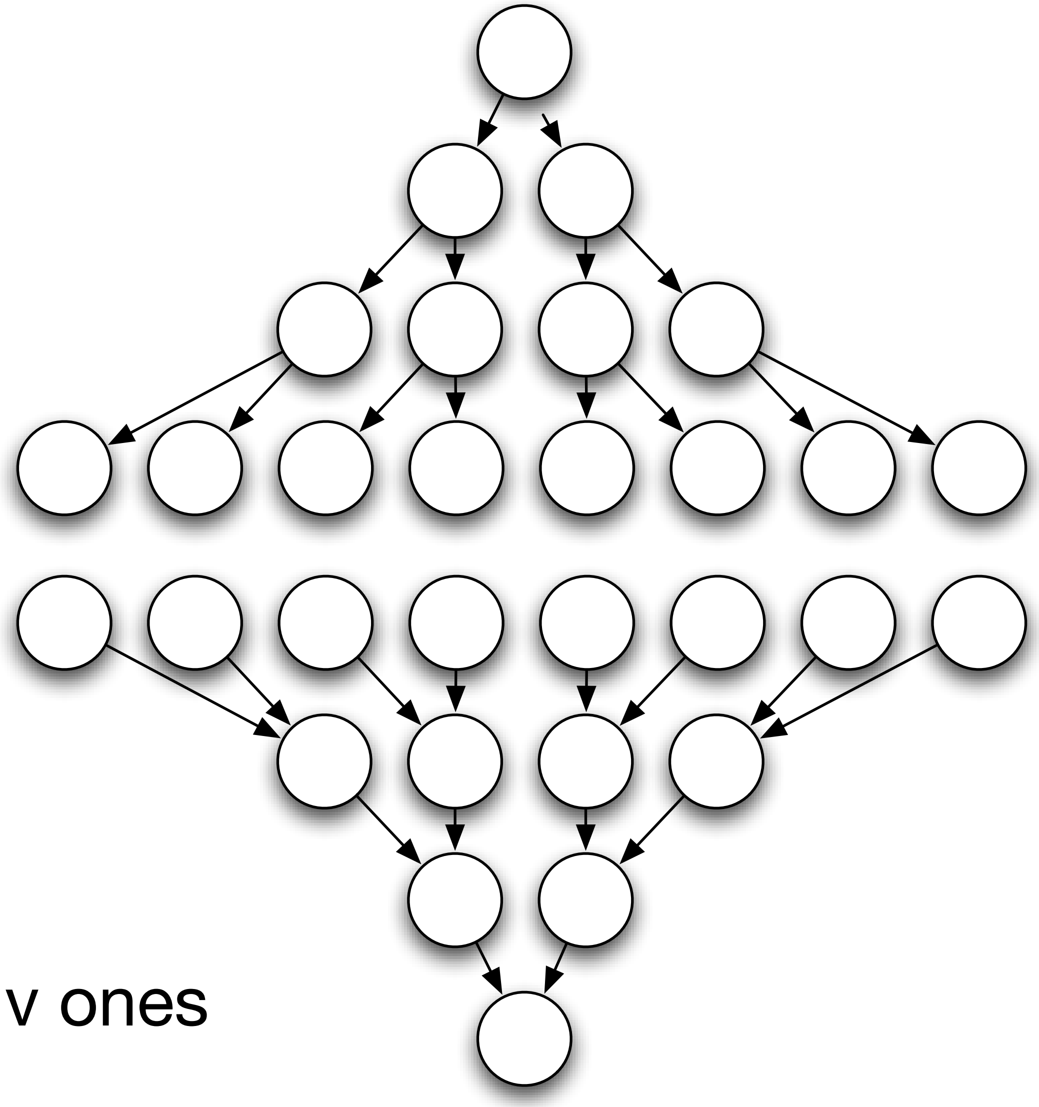
- extremely popular
  - heavy used in seq computations + opportunities for parallel execution
  - let rec div&conquer term leaf div conq x =

```
let rec div&conquer term leaf div conq x =
  if(term x) then (leaf x)
              else (conq
                      (map
                         (div&conquer term leaf div conq)
                         (div x)))
```

- parallelism in
  - generation of (div x)
  - map (embarrassingly parallel)

Divide&Conquer: performance model

- on p processors
  - log(p) divides
  - n(p-1)/p comms
    - n/2 + n/4 + ... + n/p
  - p term
  - log(p) conquer
  - n(p-1)/p comms
    - possibly different size of data w.r.t. div ones



Divide&conquer: problems

- computational grain
  - how much go deep in divide
    - the deeper the finer grain grain the larger overhead ...
  - additional parameter: size_seq
    - if(current_size <= size_seq) compute locally (no more parallelism)
- processor allocation
  - binary div&conq: keep one on local processor, delegate the other
  - more complex strategies in case of non binary div&conquer
  - (fwd pointer) macro data flow interpreter !