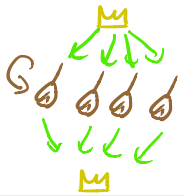
OPEN MP

MODEL

- More threads than processes possible
- Each thread unique id
- Sync. constructs to prevent DATARACE
- Shared & private vars poss
- SPMD
- Fork-join thread model
  - thread forks & creates threads
  - when done: threads join thread



MSG PASSING MODEL

- Finit processes with LM
- Expl. Comm
- No implicit sync. → only comm.
- Nothing shared

MPI

- Finite processes in comm. domain (more than 1 poss)
- Each process identified by rank in domain
- Ranks successive $(0, \dots, p-1)$
- LOCAL DATA
- EXPLICIT COMM → Reliable & ordered
- Structure of com. data ⊥ to node/mode
- Comm. d. reflect physical topology
- No com. cost model
- 3 basic comm models
  - •→• Point-to-Point (diff. modes, local & ~~local~~ compl. semantics)
  - → 1-sided (diff sync., local compl. semantics)
  - ooo collective ops (~~local~~ compl. semantics)

pThreads

Fork-join parallelism
→ can spawn new threads
Symmetric peers
→ threads can wait for others to complete
SPMD
scheduled by OS
No implicit sync.
→ threads are independent
Share same memory
Construct for sync. & coordination
→ MUTEX
→ R/W Locks
→ Condition vars

•→• Point-to-Point

MPI_Send
MPI_Recv
} MPI_Sendrecv
deadlock free
reliable
non-overtaking
ordered

MPI_Recv
MPI_Send

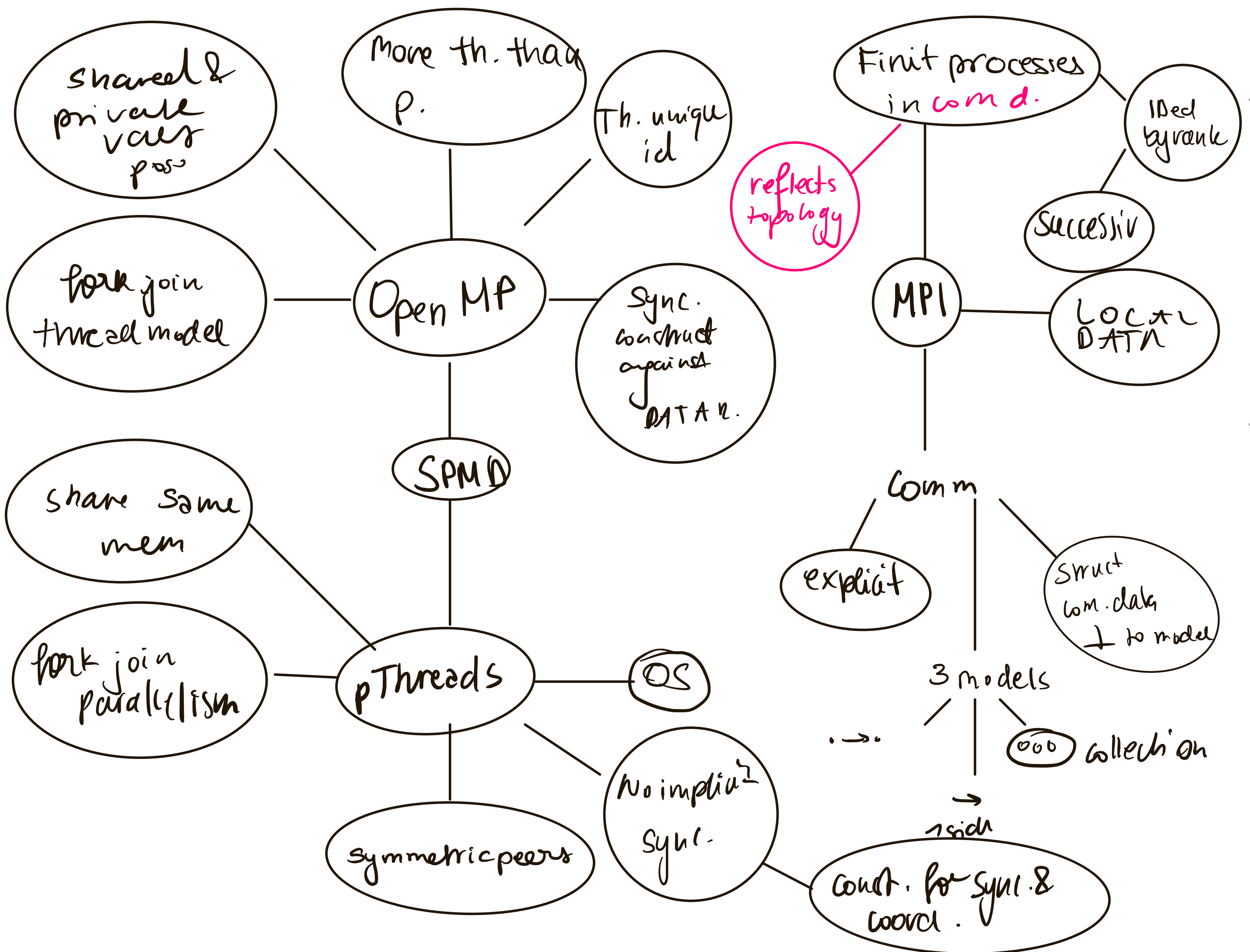
→ 1 sided

MPI_get (MPI_put)
active → passiv
origin → target

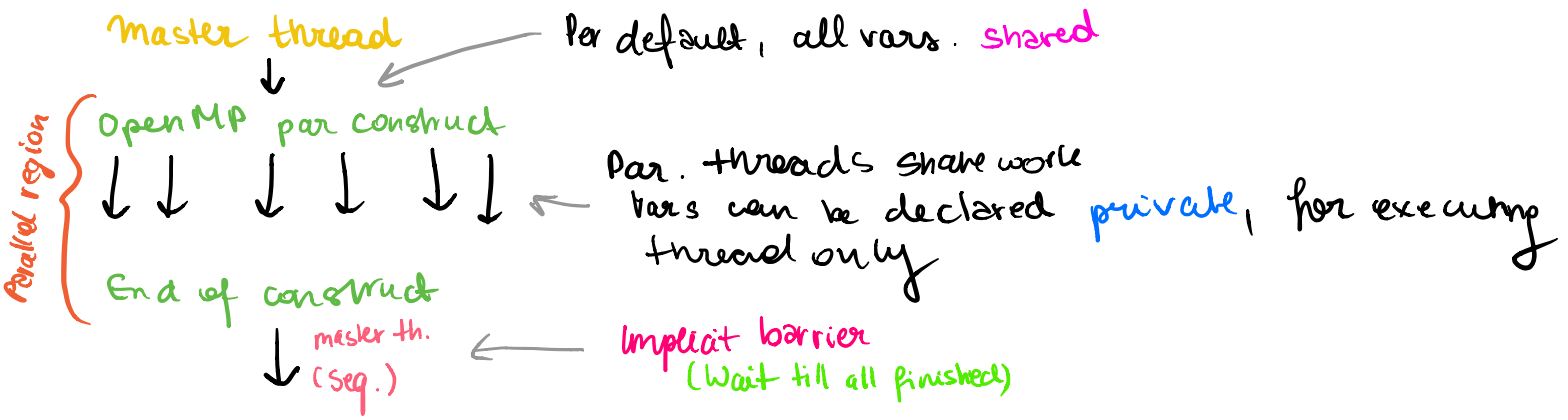
Collective

MPI_Barrier
Bcast (root)
Scatter(v)
(All) gather(v)
(All) Reduce

MPI_All to All
(Ex) Scan


shared & private vars
More th. than p.
Th. unique id
fork join thread model
Open MP
Sync. construct against DATA R.
SPMD
share same mem
fork join parallelism
pThreads
OS
symmetric peers
No implicit sync.
Cond. for sync. & coord.
Finit processes in com d.
reflects topology
IDed by rank
Successiv
MPI
LOCAL DATA
Comm
explicit
Struct com. data ⊥ to model
3 models
collection
1 side

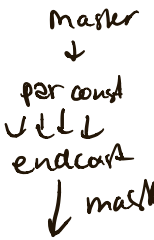
OPENMP



Data transfer between s. & p. copies var. is transparent, impl. & can be controlled

Parallel region

```
#pragma omp parallel
{
   ...
}
```



"memory model"

→ Memory is consistent (state) by end of const.

↳ updates to shared var visible

Threads asynchron

MPI Equivalents

All collective func. blocking & not sync

MPI_Send
MPI_Recv
MPI_request
MPI_Irecv
MPI_Send
MPI_Wait

} = MPI_Sendrecv

MPI_Isend, MPI_Wait } = MPI_Send

MPI_Recv, MPI_Sendrec, MPI_Send } = MPI_Gather

MPI_gather
MPI_Bcast
MPI_Bcast
MPI_Bcast
memcpy

} = MPI_AllGather (= p · Bcast)

1

blocking

not sync

Can do:

i ... root

Bcast i
Bcast j

Bcast } i
Gather }

Bcast } k

Gather } j

Bcast } i
Recv(j) }

Isend(i) }
Bcast } j

•→• POINT-TO-POINT

Reliable
Deadlock-free
Non-overtaking (ordered)

MPI_Send → MPI_Recv
MPI_Recv ← MPI_Send
} Blocking

MPI_Isend
IMMEDIATE
LOCAL

MPI_Irecv

NON Blocking

Successful: $0 \leq rank < size$

1. Sender specifies valid rank within comm. & valid (allocated) buffer
2. Receive with matching source rank & tag is eventually posted on same comm.
3. Sent data $<=$ to received amount
4. Type sig of sent & received data matches

↯ ERROR: Caught by MPI_Send: error

↳ IF NOT: DEADLOCK
↳ OTHERWISE: MPI_ERR_TRUNCATE
|| memory corruption

MPI_Send

- Short msg: Buffered at Sender / receiver → Processed later. MPI_Send can return IMMEDIATELY
- Medium msg: may be buffered locally. Sent when rec. has been posted. MPI_Send can return fast
- Large msg: participation of receiving process needed. MPI_Send cannot return before MPI_Recv becomes active

COST
$\alpha + \beta m$
Latency: Time to get 1st data elem from Source to dest. processor
bandwidth: How much data/time unit (s) can be transferred from Source to dest. processor

→ 1-sided

comm. between 2 processes
ONLY 1 Explicitly involved with comm

MPI_Get →
MPI_Put ←
} Non-Blocking

only origin Initiates comm & provides all args
PUT (origin → target) target not involved
GET
MPI_Accumulate ( )

origin puts/gets data from stand. MPI_buffer (buf, count, type)

local completion semantics

No guarantee that data have arrived before Sync. operation

Comm. window

Collective Operations

All processes involved

Different for ↗ •→• & →

Same root → If diff. roots DEADLOCK

#sent = #received

Matching type signatures (pairwise same size)

Difficult to impl: perf. correct. safety

COLLECTIVE OPERATIONS

MPI_Bcast: Data from root → a (all)

not evenly +v
- MPI_Scatter: Indiv. data root → all
- MPI_gather: Indiv. data all → root

MPI_Allgall: Indiv. all → all

MPI_Reduce: Apply ass. function(+,*,...) to data from each process
res at root

MPI_Allreduce: res to all

MPI_Reduce_Scatter: res scattered/distr.

Pi (non-root)
R(sbuf, rbuf, ... root, comm);
Pj (root)
R(sbuf, rbuf, root, comm);
only sig. for root

MPI_Scan: Prefix sum (reduction)

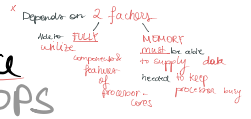
MPI_Exscan: value of proces not used in reduction

MPI_Barrier: Sync. (waits until have reached routine)

△ Caveat = Processor's Point of view

Performance of Processors

= Nominal processor performance
↳ often measured in ŦLOPS
(= Float Point Operations per second)
→ max. num of ŦLOPS



Performance of 1-core
num of instr. per clock cycle
* clock frequency

Performance of multi-core
num of cores * perf. of 1-core

determined by architecture (SIMD, num of pipelines, p. depth, ...)

c.f. ... num of ticks per sec (GHz)

PRAM Model

= Par. Random Access Machine

→ unrealistic model but useful

→ LOCK STEP (= Strictly synchronized)

EREW	Can't ACCESS memory simult.
CREW	Can't WRITE in memory simult.
CRCW	arbitrary CRCW: 1 write succeds
	Priority CRCW: priority scheme determines pr.
	common CRCW: conflicting p. must write same value

easiest model

strongest model

example: a[i] = c[i] + b[i]
a[i+k] = a[i]

(practically not possible)

TIME

$T_{par}(p,n)$
par. time with p proces

$T_{seq}(n)$
seq. running time

SPEED UP

Relative	Absolute	Linear
$\frac{T_{par}(1,n)}{T_{par}(p,n)}$	$\frac{T_{seq}(n)}{T_{par}(p,n)}$	$T_{par}(n) \geq \frac{T_{seq}(n)}{p}$
Compares par alg. against ITSELF	Compares par alg. (seq. alg) against BASELINE	Best possible speedup Argument: par alg. running on p cores can be simulated on a single core no worse then $p*T_{par}(p,n)$
Expresses to WHAT EXTENT p are well EXPLOITED	Expresses HOW well IMPROVES par alg. over the BASELINE	

Not-linear	Simulation can STILL run faster ⇒ new best known
Super-linear	$W_{par} \neq W_{seq}$ → random. alg search alg.

PARALLELISM

$$\frac{T_{par}(1,n)}{T_{\infty}(n)}$$

AMDAHL'S LAW

Assuming an alg. can be divided into

- **Seq fraction** s
  - strictly independent of n
  - strictly seq
- **Parallelizable fraction** r (perfectly)

$r = (1 - s)$

$$T_{par}(n,p) = T_{seq}(n) \cdot \left(s + \frac{r}{p}\right) = T_{seq}(n) \cdot \left(s + \frac{1-s}{p}\right)$$

Max speedup by parallelization

$$\frac{1}{s + \frac{1-s}{p}} = \frac{1}{s} \quad \text{if } p \to \infty$$

→ Devastating ⇒ Seq. alg that satisfy A.D
↳ CANNOT be used as a basis of a good par alg

VICTIMS (of par alg.)

- IN-/OUTPUT
- Seq. Preprocessing
- Seq data structures
- seq. operations

Good alg: Don't fall victim of A.L.
→ seq. part $s(n) \neq$ const. fraction of total work
⇒ BUT depends & decreases with n

TIME

Linear Speedup	Non-linear Speedup
$T_{par}(p,n) = \frac{O(T(n))}{p + t(n)}$	$T_{par}(p,n) = \frac{O(T(n))}{f(p) + t(n)}$
$t(n)$ ... $T_\infty(n)$ non-par term	$f(p)$ ... $f(p) \geq p$ ($p(f(p)) \text{ in } O(p)$)
$T(n)$ .. Parallelizable term	

SCALED SPEED-UP

→ The faster $\frac{T(n)}{t(n)}$ converges ⇒ the faster the speedup converges

Speedup as n increases

$$S_p(n) = \frac{T_{seq}(n)}{T_{par}(n)} = O\left(\frac{T(n)}{\frac{T(n)}{p} + t(n)}\right) = O\left(\frac{1}{\frac{1}{p} + \frac{t(n)}{T(n)}}\right) \to O(p)$$

SCALABILITY ANALYSIS

Strong Scaling A.	Weak Scaling A.
n is Constant	Work / processor is constant (through bigger n),
IF run time ↓ proportionally to p (linear speedup)	IF par running time is also constant alg is weakly scalable
⇒ Alg. strongly scalable	

EFFICIENCY

= comparison of Par against best possible parallelis. of Seq

Parallel efficiency (E_p)

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

Following holds
* $E_p \leq 1$
* If $E_p = e$ (= const) ⇒ LINEAR SPEEDUP
* Cost-opt. alg have constant E_p

Iso-efficiency (Does not change with n)

IF alg. does NOT have const efficiency & speedup is independent of n

THEN: calculate n as function of p
→ Done by calculating e as a function of n & p
⇒ Then changing f. to n

SCALABILITY

$f(p)$

→ Tells how n should grow as a function of p to maintain constant efficiency
→ Should not grow faster than the input size
scaling function $g(p)$
↳ Tells how much n can at most grow if par = const
→ $f(p)$ should be in $O(g(p))$

$g(p) = T_{seq}^{-1}(pw)$

Efficiency varies with the num of processors if the problem size remains constant
→ may be more efficiently solved with fewer processors

Weak Scaling

IF for (desired) constant efficiency e there is a slowly growing function $f(p)$ such that efficiency $E(p,n) = e$ for $n \Omega(f(p))$

$f(p)$ ... iso efficiency function

IF average work per processor $\frac{T_{seq}(n)}{p}$ is const at w the running time of the alg. is const at $T_{par}(p,n)$

FLYNN'S TAXONOMY

= differentiates machines based on inst

SISD	Seq. computer
SIMD	Single inst can work on big am. of data ⇒ vector instructions
MIMD	= general PRAM Each processor has own inst. & data
MISD	= Pipeline like system Single data stream passes through dif. stages
SPMD	All processors execute same program

	Found	processors	operations
Common CRCW	$O(1)$	n^2	$O(n^2)$
CRCW PRAM	$O(\log n)$	$\frac{n}{2}$	$O(n)$
	$n \times n \rightarrow$ 1 matrix		operations
CRCW Pram	$O(n)$		$O(n^3)$

} max of n num stored in array

OVERHEAD & LOAD BALANCE

→ par alg often perform more work than BEST seq
⇒ OVERHEAD

↳ Communication
↳ Synchronization
↳ Preprocessing

If overheads in bounds $(O(T_{seq}(n)))$
→ THEN: Alg CAN be WORK-OPT

Granularity (of alg)
= communication intervals

Coarse	fine
RARE communication	FREQUENT communication

= Problem of making sure that time of all processors is about the SAME

$\forall i,j$:
$T_{par(i,n)} \approx T_{par(j,n)}$

↳ IMBALANCE

If $T_{par}(i,n)$ = time processor i takes $(0 \leq i \leq p)$
⇒ THEN: LOAD IMBALANCE

$$\max_{0 \leq i,j \leq p} T_{par(i,n)} - T_{par(j,n)}$$

Static	dynamic
= when amount of work is divided BETWEEN processors	= exchange of work DURING execution of alg

oblivious	Adaptive
= SUBDIVISION of prob ONLY by input SIZE & STRUCTURE REGARDLESS of input	= prob.-indp. INPUT itself taken into consideration

Embarrisingly / Trivially / Pleasantly Parallel
bs where input can be statically distributed & ⇒ NO interaction is needed

COST & WORK

Cost

$p * T_{par}(p,n)$

= what we "pay" for
→ time of p processor cores occupied by Par

Work

$W_{seq}(n)$	$W_{par}(p,n)$
= num of ops carried out by alg. (= "work time")	= total work carried out by p cores - excluding anything NOT related to alg - time/ops spend by more p or unassigned p

Cost opt

IF cost $pT_{par}(p,n)$

$= O(T_{seq}(n))$

for best known seq alg

Work opt

IF $W_{par}(p,n)$

$= O(T_{seq}(n))$

for best known seq alg

alg = work-opt ~~cost-opt~~ ⇒ Can have LINEAR SPEEP for small range of proce

Par vs Dist vs Concurrent

Par
- Efficiently utilizing dedicated par resources to solve compt probs
- Problem solving efficiency
- → Deals with: alg., impl. & structure of comp. arch.
- Issues: many different parad, lang. & inter.

Interesting problems ⇒ significant interaction between cores
= Theoretical + Practical + Experimental C.S.

Dist.
- of making indep., non-ded. resources AVAILABLE & COOPERATE toward solving specific prob. complexes
- Availability of resources (that aren't ready at hand)

Concurrent
- of MANAGING & REASONING about INTERACTING processes that may / may not progress simult.
- Activities that may / may not happen at the same time
- & Reasoning about
- & Establishing correctness

Same prob can be viewed from all perspectives