

△ Caveat = Processor's Point of view

Performance of Processors

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

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

Par vs Dist vs Concurrent

Par Efficiently utilizing dedicated
per resources to solve comp
↳ Problem solving efficiency
→ Deals with: alg., impl. & structure of comp. arch.
↳ Issue: many different parad., lang. & user.
Interesting problems ⇒ significant interaction between comp.
= Theoretical + Practical + Experimental C.S.

Dist. W of making indep., non-det. resources
AVAILABLE & COOPERATE toward solving specific prob. complexes
↳ Availability of resources (local area: not ready at hand)

Concurrent M of MANUATING & 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

PRAM model

PRAM model

→ unrealistic model but useful
→ **LOCK STEP** (= strictly synchronized)
example: $a[i] = a[i] + b[i]$
(practically not possible)

	EREW	CREW	CRCW
Can't ACCESS memory simult.	Can't WRITE in memory simult.	Arbitrary ARW: no write simult.	Priority (RW): priority scheme determining pr.
Common CREW: conflicting p. must use same value			

FLYNN'S TAXONOMY

= different kinds machines based on inst & data

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

TIME
T_{par}(p,n) per time with p processors
T_{seq}(n) seq. running time

SPEEDUP

Relative	Absolute	Linear
$\frac{T_{seq}(1,n)}{T_{seq}(p,n)}$	$\frac{T_{seq}(n)}{T_{par}(p,n)}$	$\frac{T_{seq}(n)}{T_{par}(p,n)} \geq \frac{T_{seq}(n)}{p}$
Compares parallel against 1 seq. Expresses to what extent p are well exploited	Compares parallel against BASELINE Expresses HOW well p are used over the BASELINE	Best possible speedup against: p seq. running in parallel on a single machine. no single atom. no parallelism

per-linear simulation on still run table at new level
superior
upper # value
→ sequential

PARALLELISM

$\frac{T_{seq}(n)}{T_{par}(n)}$

COST & WORK

Cost	Work
$p + T_{seq}(p,n)$ → time of p processors occupied by par	$W_{seq}(n)$ = sum of p + work done by p processors (= work)
$\frac{p + T_{seq}(p,n)}{p} = O(T_{seq}(n))$ for best known seq. alg.	$W_{par}(p,n)$ IF $W_{par}(p,n) = O(T_{seq}(n))$ for best known seq. alg.

alg = work + cost ⇒ can have **LINEAR SPEEDUP**
cost = cost ⇒ for small range of processors

OVERHEAD & LOAD BALANCE

→ par alg often perform more work than BEST seq ⇒ **OVERHEAD**
= Problem of making sure that time of all processors is about the same
W_{i,j}: $T_{par}(i,n) - T_{seq}(i,n)$

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

Static	dynamic
= when amount of work is divided BETWEEN processes oblivious = Substitution of post-hoc input size & STRUCTURE then into results of computation of input	= exchanging of work DURING execution of alg = minimizing Friction for every parallel in which input can be initially distributed

granularity (of alg)
communication intervals
= cost of communication
BASE FREQUENCY
communication

if overheads in bounds
→ THEN: alg CAN be **WORK-OPT**

AMDAHL'S LAW

Assuming an alg. can be divided into

- Seq fraction S
 - strictly independent of n
 - strictly seq
- Parallelizable fraction r
 - perfectly

$$r = (1 - S)$$

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

$$\text{Max speedup by parallelization} \\ \frac{1}{S + \frac{1-S}{p}} = \frac{1}{S} \text{ if } p \rightarrow \infty$$

→ Devastating ⇒ Seq. alg that satisfy A.D. CAN NOT be used as a basis of a good parallel

⇒ VICTIMS (of par alg.)

- IN-OUTPUT
- Seq. Preprocessing
- Seq. operations

Good alg: Don't fall victim of A.L.
→ Seq part seq ≠ const. fraction of total work
⇒ BUT depends & decreases with n

TIME

Linear Speedup	Non-Linear Speedup
$T_{par}(p, n) = \frac{O(T(n))}{p + t(n)}$ $t(n) \dots T_{seq}(n)$ non-parallel term	$T_{par}(p, n) = \frac{O(T(n))}{f(p) + t(n)}$ $f(p) \dots f(p) \geq p$ $p(f(p)) \ln(q(p))$

SCALED SPEED-UP

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

Speedup as n increases

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

SCALABILITY ANALYSIS

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

INTRO

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

$g(p) = T_{seq}^{-1}(p \cdot w)$
Efficiency varies with the num of processors if the problem size remains constant
→ may be more efficiently solved with fewer processors

EFFICIENCY

= comparison of Par against best possible paralleliz. of Seq

Parallel efficiency (E_p)

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

Following holds

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

Is efficiency
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

Weak Scaling

IF for (derived) constant efficiency e there is a slowly growing function $f(p)$ such that efficiency $E(p, n) = e$ for $n \in \Omega(f(p))$
⇒ $f(p) \dots$ is 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 $T_{par}(p, n)$

CACHES

Basic Information

- Works on larger blocks of memory address
- Maps addresses of words in memory to addresses in cache
could be 64 bytes
- Thought of as being segmented into small blocks
→ Each block can be mapped to some cache line
→ Cache line: Stores memory block (word, multi info)
Cache lines
Block = cache line

SHARED MEMORY

3 Types

Directly Mapped	Fully associative	(L-way) Set ass.
Each block mapped to 1 distinct in cache	Each block can go anywhere	Like directly mapped cache BUT space for MULTIPLE blocks per position

Granularity
= unit of memory block
When address is read → whole memory into cache line address belongs to is read
Can benefit applications with 2 types of locality of access

Locality of access	
For application & alg.	
Temporal L.	Spatial L.
Memory address is REUSED frequently → DON'T evict	Addresses of same block are also used → use of data elements within close storage locations

Block already in cache
→ UPDATED

On WRITE

Block NOT in cache

WRITE ALLOCATE

Block allocated

NON-ALLOCATE

Memory is written directly

HIT / MISS RATE
= ratio of hits/misses

ON READ

1. Block is calculated (word belongs to)
2. Check whether block in cache or not

HIT
When p. reads word → block in cache

MISS
When p. reads word → block NOT in cache Have to read in cache line → line can be full ⇒ EVICT some line

Cache misses

Cold (compulsory)	Capacity	Conflict
No address blocks in the cache	All times are full line has to be evicted	Already element in line
	Only FULLY ASSOCIATIVE	Only Directly & Set associative

ON UPDATE

Block already in cache

WRITE-THROUGH
CACHE

→ Block updated & value written

directly to memory for

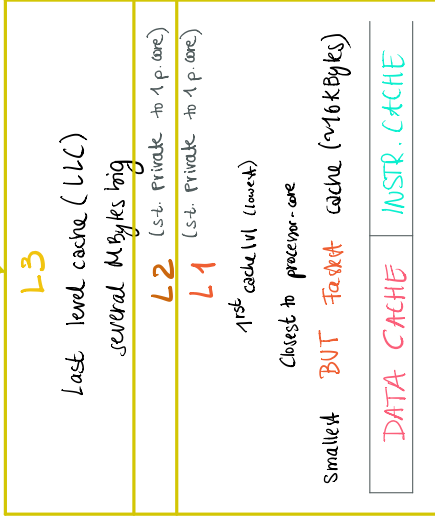
Cache line not in memory
→ WRITE BACK
WRITTEN when line

is evicted

MULTI-CACHES

Basic info

- = Modern **MULTICORE SYSTEM** (processors in system, threads)
- Structured in several dimensions
- Hierarchy of caches of increasing size



FALSE SHARING

= When 2 caches have same blocks of memory
 ↳ UPDATE to 1 block will cause whole line of other to be replaced
 ⇒ caused by **granularity** of cache system

PADDING

= wasteful strategy that uses ONLY 1 address per block



SHARED MEMORY

Has another cache as well
 → **VIRTUAL CACHE (TLB)**
 Translation look-aside

Cache coherence
 Guarantees that p. cores have a **CONSISTENT** view of individual memory addresses

Cache coherence problem

= When 1 block is updated in 1 cache & also stored in another cache of another processor

Cache system	Non coherent
If line of other cache will be updated at SAME POINT on time, inconsistent in cache	If line will never be updated

Solution: Cache coherence protocol
 ↳ May cause lots of **cache traffic**
 ↳ Non-trivial task
 ↳ can affect performance

Current shared multi-processor systems = **Cache coherent**

Cache coherent system

	IF	THEN
① LOCAL CONSISTENCY	C_1 & C_2 at t_1 & C_1 at t_2 & no other C to a between t_1 & t_2	C_1 & C_2 at t_1 & C_1 at t_2 & no other C to a between t_1 & t_2
② UPDATE TRANSFER	C_1 & C_2 at t_1 & C_2 at t_2 & no other C to a between t_1 & t_2	C_1 & C_2 at t_1 & C_1 at t_2 & no other C to a between t_1 & t_2
③ ORDER (write consistency)	C_1 & C_2 to a at same t	C_1 & C_2 to a at same t

M x M M. & CACHE PERF.

Access locality matters $\rightarrow$ M x M block

$\hookrightarrow$ Problem: Given: Matrix A (n x l), Matrix B (l x m) } in
Matrix C (n x m) } out

$$\text{ToDo: } C[i, j] = \sum_{k=0}^{l-1} A[i, k] \cdot B[k, j]$$

```
for (i=0; i<n; i++) {
  for (j=0; j<m; j++) {
    C[i][j] = 0.0;
    for (k=0; k<l; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

$\Rightarrow$ seq app. $O(n \cdot m \cdot l)$ or $O(n^3)$

Can be parallelised

- All three loops
- Order of parall. matters
 - $\rightarrow$ Differences
 - $\rightarrow$ Because mat. are accessible
 - $\rightarrow$ Different index order causes more / fewer cache misses

BLOCKED M x M

START: Split matrices into blocks

$\hookrightarrow$ THEN: multiply with 6

nested loops

$\Rightarrow$ block size can be

chosen depending on

cache size

$\Rightarrow$ CACHE - AWARE ALG.



CACHE - OBVIOUS

RECURSIVE M x M

Recursively split A & B in
smaller submatrices until very
small
 $\rightarrow$ iterative solution

SHARED
MEMORY

PTHREAD

C: use #include <pthread.h>

MEMORY SYSTEM

Thread	= Smallest execution unit that can be scheduled
Characteristics	
Fork-join parallelism	Thread can SPAWN ANY number (up to limit) → Can wait for completion of threads Referenced by thread identifiers
Symmetric pass	Threads are S.P. ANY thread can wait for completion of ANY other thread via thread identifiers
SPMD	Threads have same program (SPMD)
MIMD	BUT: can have different traces (MIMD)
OS	Threads are scheduled by OS
Synchronisation	No implicit synchronisation of threads they progress independently from each other Share the same memory
Share	Constructs for coordination & synchronisation & updates to shared objects are provided:
Coordination constructs for synchron.	MUTEX Readers - & - Writers - lock Condition variable

RACE CONDITIONS

When 2 threads update a variable at same time outcome may be write of either thread	→ Result = NON-DETERMINISTIC
Special race condition = DATA RACE	
2 or more threads access shared memory	
One of access is a write	

DATA RACE

= special kind of race condition
When 2 or more threads access a shared object & at least 1 of the accesses is an UPDATE
→ Undesirable to determine whether data race or not

Cache system part of memory hierarchy	- From fast low lvs → Slower bigger higher lvs
Write Buffer for memory writers (FIFO)	
For multi-core CPUs not every processor has direct connection to main memory	→ INSTED connected to memory controller
Some processors closer to memory controller than others	→ addresses access times are not uniform

SHARED MEMORY

APPL. PERF. & MEM. HIER.

MEMORY CONSISTENCY

When process 1 sets a certain flag in memory & process 2 checks if that flag is set it may happen that the flag is still in the write buffer & process 2 may read a wrong value
Frameworks like pthreads or OpenMP help

Memory - bound	Compute - bound
When execution (+ associated reading & writing from memory) of instructions is faster than reading or writing the instr. in memory	Other way around, instructions are faster than they are executed

SUPER-LINEAR SPEEDUP

Through memory hierarchy of large caches, working sets of each processor get smaller & eventually can fit into the fastest caches

CRITICAL SECTIONS

Basic info

= Piece of code that should **NOT** be *executed concurrently* by SEVERAL threads

Thread running code in critical section

Should exclude other threads from running code & need to cooperate to ensure this

⇒ Guarantees that C.S. is being excluded

by at most 1 thread

→ **MUTEX**

MUTEX PROBLEM

Basic info

= Guarantees **MUTEX** & not trivial prob.

MUTEX

Basic info

= mutual exclusion

In pthreads lock

= mutex

& only shared objects are allowed to be updated by acquiring a mutex to do so
Identified by pthread_mutex_t type
Have to be initialized before (statically / dynamically)

CONDITION VARIABLES

Basic info

→ Associated with **MUTEX**

- Thread waits on variable till *signal* occurs
- Thread can *signal* one thread only
- **BROADCAST** signals all threads at once



MONITOR

concepts with *condition*-
with *signals*
& *waits*

SHARED MEMORY

BARRIER

Basic info

Similar to mutex with conditions

CONCURRENT INIT.

1. Thread executes init. code before other threads begin

ATOMIC OPERATIONS

Atomic instructions

= Carry out instructions that cannot be interfered with other threads

→ Example: $a = a + 27$

→ Atomic instruction through

fetch-and-add

! Crashing threads during *a.i.* will **!**

NOT cause *DEADLOCKS*

Wait-free Cause Atomics execute in $O(1)$

Lock-free

IF ANY thread will be able to execute instruction in a bounded amount of time (*not always given*)

LOCKS

Basic info

= ? programming model to guarantee MUTEX on a CRITICAL SECTION
Have to be DEADLOCK-FREE
STARVATION-FREE IF Thread will be starved
CONTENTENDED = Locks where many threads are competing
Fair IF locks provide stronger starvation freedom guarantees

Types

Mutex-locks	Allow execution of some code when lock could not be acquired
Spin-locks	Threads test by busy waiting
Blocking-locks	Waiting threads are blocked & put by OS
Readers-Writers locks	1 lock for read & 1 lock for write

LOCKS IN DATA-STRUCTURES

Trivially make data structures work with parallel alg. by using a single lock for all operations on structures
OTHER way: Construct CONCURRENT DATA STRUCTURES

Provide **MUTEX** as follows

- Thread that want to enter **critical** tries to **acquire** corresponding lock
 - SUCCESS**
 - FAILURE**
 - Enter **C.S.**
 - Release** lock when Done
- => **THREADS** are **BLOCKED**^{*} till lock is **acquired**

If thread can't acquire lock
-> can't progress
-> **BLOCKED**

LOCK lock acquire operations
UNLOCK lock Release operations

SHARED MEMORY

PROBLEMS LOCKS

DEADLOCKS

- can happen easily
- Can also happen when same thread tries to **acquire** same lock again
- when threads crash during critical section

Harmful SERIALIZATION

- > Can be caused by lock around critical section

STARVATION

PROGRAMMING MODEL

Basic Info

Like pThreads
= Fork-join thread model → BUT: threads are less explicit than in pThreads
- MasterThread forks & creates working threads - when finished working threads join back to MasterThread
SPMD All threads execute the same program
ID Each thread has a unique id
threads > # processors possible shared & private variability in threads are possible
SYNCHRONIZATION constructs to prevent RACE CONDITIONS

Parallelism mostly implicit through work-sharing
→ SMP

OPENMP

Basic info

= Open Multi Processing
• Standard for C & Fortran
• Thread-based

Has to be compiled in gcc with -fopenmp
file to be included with #include <omp.h>



Make data races possible

PARALLEL REGIONS

Forcing starts at parallel region
Designated by #pragma omp parallel [...]
Number of threads in parallel region cannot be changed after start
Thread number set by the runtime environment or library call or num-threads() pragma

LIBRARY CALLS

omp-get-thread-num	returns id
omp-get-num-threads	returns number of threads
omp-get-max-threads	returns max num procs. threads
omp-get-wtime	returns wall clock time in seconds
omp-get-wtick	returns tick resolution of timer

SHARED VARIABLES

Default is all variables before region are shared
All variables declared in region are private
Sharing of variables set by clause in pragma
- PRIVATE(vars...): Makes uninitialized copies of variables
- FIRSTPRIVATE(vars...): Makes copies & initializes value to value before parallel region
- SHARED(vars...): Declares variables as globally shared
- DEFAULT(shared none): Makes variables shared / NOT shared by default
Use like #pragma omp parallel private (a,b,c) shared (d) default(none)
GOOD PRACTICE: Set no variables to shared per default

LOOP OF INDEPENDENT ITERATIONS

LOOP SCHEDULING = Assignment of iteration blocks to threads
Each iteration has to be executed EXACTLY ONCE
DATA RACES have to be avoided

#pragma omp for [clauses...]
for (loop range)

All threads must have same **START** & **END** for i & same steps

Loop ranges have to be **FINITE & DETERMINED**
↳ No while loops

End condition has to be in the form of $i < n, i \leq n, i > n, i \geq n, i = n$
 $i = i - inc$

= Canonical loops

COLLAPSING NESTED LOOPS

Transforming multi loops into 1

#pragma omp parallel for collapse (depth) [clauses...]

Depth = amount of nested loops to be parallelized

#pragma for [clauses...]

Shortcut for parallel region with 1 loop

Next statement → for loop

ALWAYS **BARRIER** after

SHARED MEMORY OPENMP

! openmp is slower!

MASTER & SINGLE

#pragma omp master

Declares statement to be executed by master thread only
 $id = 0$

Other threads will simply skip

BARRIER

#omp barrier

No thread can continue until all threads have reached barrier

SECTIONS

Code is split into small pieces that can be executed in parallel by available threads

#pragma omp sections

Outer structure

End of block is implicit barriers
→ **NO WAIT** period

Variables can be designated (first)private

#pragma omp single

Statement is executed by either one of the threads

- other threads will **WAIT**

UNTIL all threads have reached **END**

- Can be eliminated by **NOWAIT**

AFTER **SINGLE**

* might cause race conditions

- Allows making vars private/firstprivate
↳ Can't do this with **master**

#pragma omp section

Marks an independent code section

which thread executor which

section → Designed by runtime system

Best case: Each thread executor a section

All threads run in parallel

REDUCTIONS

#pragma omp for reduction
(over: var: variables)

is clause of for

Operators are '+', '-', '*', '&', '||', 'min', 'max'

LOOP SCHEDULING

= How each thread is assigned to each iteration

Loop range is divided in consecutive chunks

static	dynamic	guided
chunks SAME size → Each thread gets c. after another ⇒ computation of chunk assignments is very fast (low overheads)	Each thread grabs next chunk it can	with chunk size changes Determined by number of iter. divided by threads

Set like `schedule (static | dynamic | guided [, chunksize])`

- For `guided` `chunksize` = minimum size

- If no size is given ⇒ default size is used

Alternative: `schedule (auto | runtime)`

OpenMP compiler determines scheduling set at runtime

SHARED MEMORY OPENMP

TASK & TASK GRAPHS

#pragma omp task [clauses...]

task = like function call

Completion of task may

NOT happen IMMEDIATELY

→ AT LATEST when

completion is requested

tasks can access any variable of the thread

→ IF those variables are

CHANGED in the task

DATA RACE may occur

task can be designed as FINAL

& TASKS not generated any additional tasks

#pragma omp taskwait [depend(...)]

wait for completion of generated tasks

Only DEPENDENCY CLAUSE allowed

COLLAPSING NESTED LOOPS

Transforms multiple loops into one

#pragma omp parallel for collapsing (depth) [clauses...]

Depth = amount of nested loops to parallelized

SPECIAL LOOPS

OMP can use SIMD (vector) operations

```
#pragma omp simd [clauses...]
```

Followed by canonical for loop

Loop executed by mult. threads

Each chunk executed with SIMD ops

```
#pragma omp parallel for simd [clauses...]
```

Same

BUT: with declared parallel region

```
#pragma omp taskloop [clauses...]
```

Recursively break iteration range into

smaller ranges ($\Rightarrow$ Executed by threads)

Initiated by single thread

Size can be adjusted by `granularity()`

LOOP WITH HOPELESS DEPENDENCIES

```
#pragma omp ordered
```

For loop dependency models that can't

be handled differently (normally)

ONLY 1 possible ordered block

in loop

$\hookrightarrow$ other parts of iteration can still be performed in parallel

SHARED MEMORY OPENMP

CILK

Inspired OMP

Not supported anymore (2018)

Constructs:

`cilk_spawn` (like task)

`cilk_sync` (like taskwait)

`cilk_for` (like taskloop)

Execute threads through

WORK STEALING ALGORITHM

Each thread has a task queue

$\hookrightarrow$ If local tasks done for thread

$\Rightarrow$ STEALS thread from

Don't there are no more tasks to

steal

MUTUAL EXCLUSION CONSTRUCTS

```
#pragma critical [cname]
```

Threads will wait & ONLY 1 THREAD will execute code of critical section

Section can have a name

Shared variables can be updated by task in critical section

```
#pragma omp atomic [read|write|update|capture]
```

For simple C.S.

Allow FETCH-AND-ADD (FAA)

atomic operations

Update: $x++$, $x = op \dots$

Capture: $y = x++$, $\dots$

LOCKS

DON'T have condition variables

Recursive (nested) locks are possible in OMP

RAM

(= Naive, dist. mem par computers)

P processes inter-connected through dedicated comm. nw.

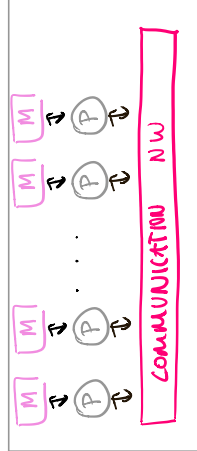
- distributed over processors
- LOCAL to processors

Each processor can ONLY DIRECTLY access own

local memory

→ Explicit communication needed for solving non-trivial problems

DIST MEM. PAR. PROG. MODELS



→ Processes (active entities) executed by processors

INDEPENDENT NON-SHNC. Processes execute

LOCALLY stored program on LOCAL DATA

with other processes EXCL. through (explicit) communication

Communication driven

Explicit message passing & SHNC.

between pairs

($m_1, p, m_1, occ, m, \dots$)

Data driven

Data struct. diff.

according to rules

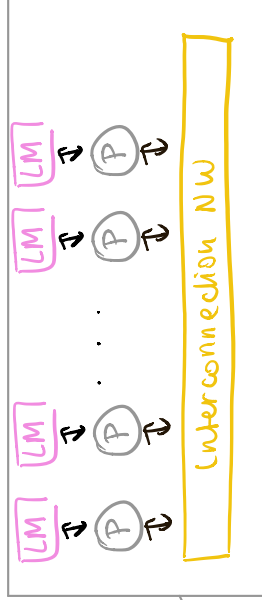
if p. changes data

struct. of other p

→ THEN through comm

($l, p, c, \dots$)

DISTRIBUTED MEMORY PARALLEL SYSTEMS



DIST. MEM. SYSTEM ARCHITECTURE

SINGLE PROCESSORS with LOCAL MEM

communicate through comm. nw

Properties (determining comm. perf.)

- Structure: Topology
- Capabilities:

1 or more ops per nw. component

Routing (technique)

Switching strategy

Cost: Latency & bandwidth

Time to get 1st data elem from source to dest. processor

How much data/min unit(s) can be transferred from source to dest. processor

TOPOLOGY

(Uni) Directed graph $G = (V, E)$

Processors (uni/bi) directed link

+ NW switches

Types

DIRECT NW

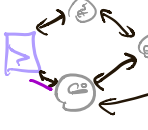
only processors

→ mesh / torus, hypercube, Kautz- $2-N$, ...

INDIRECT NW

Processors connected via dedicated switch nodes

→ Flat tree, Clos nw, $\otimes$ NW, $\otimes$ NW



Path (v,w)

= len of shortest

path lower

bounds comm.

latency between (v,w)

NETWORKS

Worst pos.	Linear processor array $\square$ processor ring $\bigcirc$
Tree	Bi / k-ary trees Bisection ($T=1$)
D-Dim. mesh	Processors idled by int Special torus nw → edges wrap around Special torus nw
Hypercube	Build as torus nw with 3 - 6 dim
Multi-stage	Fully-connected: 1 ported comm. g. → In each comm up / shp. 1 processor can send from at most 1 other processor
Ideal bisection: $(\frac{n}{2})^2$ diam (T) = 1	EXPENSIVE

COMMUNICATION ALG

IN NW

Uni-directed $\rightarrow$ SEND[RECV]

Bi-directed $\leftrightarrow$ SEND[AND] RECV
to/from same processor: Telephone m.
to/from diff. processors: Send-recv is directional

Broadcast

= collective operation

1 processor has data to be comm.

to all processors

Problem: How to transmit data from

ROOT to every other node

in min. comm. steps

- lower bound:

$\lceil \log_{k+1}(p) \rceil$ in

k -degree with

p nodes

$\Rightarrow$ ROOT $\rightarrow$ VN $\dots$
VN VN VN $\dots$ virtual nodes of nws

Comm. COSTS

Linear transmission cost

assumes time of $\alpha + \beta m$

$\alpha \dots$ start up latency

$\beta \dots$ time/unit of data

DISTRIBUTED MEMORY PARALLEL SYSTEMS

PROGRAMMING MODELS

Concrete nw properties abstracted

$\rightarrow$ FULLY CONNECTED NW

Processes

• NOT SPMC

• comm. through impl./exp. transmissions

$\downarrow$

DEADLOCK-free

+ correct

DIST. system programming models

INDEPENDENT NON-SPMC. Processes execute

LOCALLY stored program on LOCAL DATA

with other processes EXCL. through

(explicit) communication

Communication driven

Data distr. driven

Explicit message passing & SPMC. between pairs (MPI, PVM, Occam, ...)

Data struct. diff. according to rules if p. changes data struct. of other p $\rightarrow$ THEN through comm

diameter(G)

$\max(|\text{shortest path}(u, v)| \mid \text{overall } u, v \text{ in } V)$
= lower bound on comm. steps between 2 elems

degree(G)

max degree (edges of G)

"cost-factor"

high degree gives potential for more simult. comm.

multi port

$\Rightarrow$ more costly (cables)

bisection width

= num of edges to remove so

that graph is partitioned

into 2 equally large subsets

$\min(|\{u, v\}| \mid u \in V_1, v \in V_2)$

overall partitions V_1, V_2 of V

with $|V_1| \approx |V_2|$

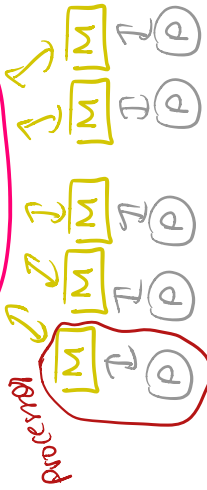
$\rightarrow$ lower bound transport ops

$\Rightarrow$ processors have to

exchange info. with

all other processes

MSA PASSING MODEL



message passing progr. model
 ↳ structures parallel processes through **send** → **recv** message.

Comm. Sep. Processes
 ↳ NO DATA RACES

CHARACTERISTICS

Finite set of processes

Each p. identified by rank in domain

local data

Expl. comm. **send** → **recv** (all p. expl.)
 ↳ **RELIABLE** (shared nothing)

No implicit SYNC (ONLY comm.)

(+)

data explicit (when, what, where)

↳ **CORRECTNESS**

Deterministic → **NO RACE CONDITIONS**

locality = $\mathcal{O}$ to performance

MPI

Message Passing Interface
 = Standard for par. pr. in msg passing model
 - HPC applications
 - Paradigmatic

Folie 77

MPI

(more than 1 possible)

more than 1 domain poss.

× correct (no bits missing)
 reliable (msg sent, will be received)
 deadlock be recovered
 free without comm.
 keine Deadlocks

Exception:
 1-sided-comm.

DESIGN PRINCIPLES

High performance

(Clock to HW fund., low protocol stack overhead)

Portability & Scalability

Memory efficiency

Coexist with par. interfaces

Support const. of tools

heterogeneous sys.

MIMP / SPMD

library building

appl. specific Libraries

MPI MODEL

Finite set of processes in communication domain

Processes identified by rank in c.d.

P. operate on **LOCAL DATA**

EXPLICIT COMMUNICATION

Reliable & Ordered

Comm. modelly

• → Point-to-Point

LOCAL & LOCAL

comp. loc. semantics



Collective
 MPI-Barrier
 MPI-Alltoall (ex) Scan
 Scatter (v)
 (All) Gather (v)
 (All) Reduce

• → Point - to Point
 MPI - Send
 MPI - Recv
 MPI - Sendrecv
 MPI - Recv
 MPI - Send

{ MPI Sendrecv
 deadlock free
 reliable (ordered)
 non - overlapping

→ 1 sided
 MPI - get
 (MPI - put)
 origin
 active
 → target
 passive

MPI

Collective
 MPI - Barrier
 MPI - Broadcast (root)
 MPI - Scatter (v)
 MPI - Gather (v)
 MPI - Reduce
 MPI - Alltoall
 MPI - Scan

MPI Equivalents

All collective func. blocking & not sync

MPI_Send	} = MPI_Sendrecv
MPI_Recv	
MPI_request	
MPI_Irecv	
MPI_Send	
MPI_Wait	

MPI_Isend } = MPI_Send
MPI_Wait

MPI_Recv } = MPI_Gather
MPI_Sendrecv
MPI_Send

MPI_gather	} = MPI_Allgather (= P <u>Bcast</u>) blocking notsync
MPI_Bcast	
MPI_Bcast	
MPI_Bcast	
memcpy	

Can do:

i...root

Bcast i
Bcast j

Bcast i
gather }
Bcast j
gather } j

Bcast i
recv(j) }
Bcast(i) } j
Bcast

POINT-TO-POINT

reliable

Deadlock-free

Non-overlapping (ordered)

MPI-Send $\rightarrow$ MPI-Recv
MPI-Recv $\leftarrow$ MPI-Send

MPI-Send
IMMEDIATE
LOCAL

MPI-Recv

Blocking
Non Blocking

Successful:

① Sender specifies valid rank within comm.

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 locally
Medium msg may be buffered locally

Sender receiver sent when rec. has been posted

Processed later

MPI-Send can return fast IMMEDIATELY

Large msg participation of receiving process needed

MPI-Send cannot return before MPI-Recv becomes active

MPI-Send can return fast

COST
 $\propto \alpha + \beta m$
Latency Time to get data
bandwidth data transferred from source to dest process

1-sided

comm. between 2 processes ONLY 1 Expiry involved with comm

MPI-Get $\rightarrow$
MPI-Put $\leftarrow$

Non-Blocking

only origin provides all args
MPI-Accumulate ()
origin puts/get 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 $\rightarrow 0 \rightarrow 2 \rightarrow$

Same root $\rightarrow$ 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 $\xrightarrow{a}$ all
 not evenly $\rightarrow$ MPI_Scatter: Indiv. data root $\uparrow$
 MPI_Gather: Indiv. data all $\rightarrow$ root

MPI_Alltoall: Indiv. all $\rightarrow$ all

MPI_Reduce: Apply op, function($t, f, \dots$)
 to data from each process
 res at root
 MPI_Allreduce: res to all
 MPI_Reduce_Scatter: res
 scattered /
 distr.
 Pi(non-root)
 Pj(root)
 only sig.
 for root

MPI_Scan: Prefix sum (reduction)

MPI_Exscan: value of process
 not used in

MPI_Barrier: Sync. (waits until
 have reached routine)
 reduction