(Για 2 nodes, όχι για ολόκληρο δίκτυο!)

$$\text{latency} = \text{send overhead} + \text{flight time} + \frac{\text{packetsize}}{\text{bandwidth}} + \text{receive overhead}$$

OCN SAN LAN WAN →

Μεγαλύτερο time of flight
Μεγαλύτερη πολυπλοκότητα

Timeofflight : time for first bit to arrive through network

Send/Recvoverhead : processing required for packet

Packet pipelining: όταν δεν χρειάζεται response για την αποστολή του επόμενου πακέτου

$$BW_{\text{linkInjection}} = \frac{\text{Packetsize}}{\max(\text{SendOverhead, Transm. Time})}$$
$$BW_{\text{linkReception}} = \frac{\text{PacketSize}}{\max(\text{RecvOverhead, Transm. Time})}$$
$$BW_{\text{effective}} = 2\min(BW_{\text{injection}}, BW_{\text{reception}})$$

Shared: κεντρικός δίαυλος, collision detection, εύκολο broadcasting, απλότητα, races
Switched: passive p2p συνδέσεις σε διάφορα disjoint portions, συνδέονται με active switch components. Routing → path granted απο κεντρικό η κατανεμημένο arbiter (collision detection) → αν αποτύχει, συνήθως Packet buffering. Μεγαλύτερο effective bandwidth, πιο πολύπλοκο routing.

Lost wakeup: signal μόνο σε οριακές συνθήκες ξυπνα μόνο ενα thread, δεν παίρνει το lock (δεν προλαβαίνει π.χ.) αρα τα αλλα δεν ξυπνανε ποτε. Λύσεις: `signalAll()` + lock timeout. ↓

Τελικά ισχύει: $\text{FlightTime} = T_{\text{totalProp}} + T_R + T_A + T_S$

$BW_{\text{effective}} = \min(N \times BW_{\text{linkInjection}}, BW_{\text{network}}, \sigma \times N \times BW_{\text{linkReception}})$

Centralized Switch Nets: **Crossbars** N^2switches
MINs $(N/k)\log_k N$ switches
Non-blocking: fat trees (π.χ. Cenes, Clos network)
→ FatTree: $\frac{2N}{k}\log_{k/2} N$ switches

Livelock: πακέτα ακολουθούν συνεχώς non-minimal routes (unbounded number of nonminimal hops)
Deadlock: πακέτα μπλοκάρουν περιμένοντας resources που απασχολούν άλλα packets

$$\text{Latency} = \text{SendOverhead} + T_{\text{linkprop}} \cdot (d+1) + (T_r + T_a + T_s) \cdot d + \frac{\text{PacketSize}}{\text{BandWidth}} \cdot (d+1) + \text{RecvOverhead}$$ → Για δίκτυο

($d =$ number of hops)

Peterson Lock
```
//thread j
victim = j; flag[j] = true;
while (flag[i] &&
       victim == j)
{ // wait }
```

Lamport Lock
```
//thread j
flag[j] = true;
label[j] = max(label[])
           + 1;
while [exists k <> i →
flag[k] &&
(label[k],k) < label[j], j)]
{ // wait }
```

Starvation & Deadlock Free

Problems:
instruction reordering (compilers)
Write buffers (processor arch.)

Lock1: Peterson χωρίς victim
Δουλεύει μόνο όταν το ενα thread πίσω απο το αλλο, deadlock αν interleaved.
Lock2: Peterson χωρίς flag, δουλεύει μόνο αν threads interleaved

TAS Lock
```
Lock(): while (state.getAndSet(true))
{ // wait }
Unlock(): state.set(false);
```

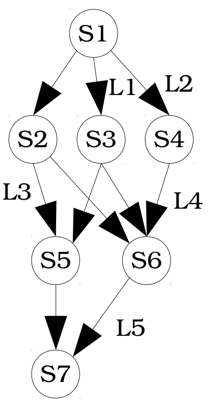
TTAS Lock
```
while (true) {
   while (state.get()) {//wait};
   if (!state.getAndSet(true))
      return;
}
```

Queue Lock (tail: atomic int, mySlotIndex: Thread-local integer)
```
Lock: slot = tail.getAndInc() % size;
mySlotIndex.set(slot);
while (!flag[slot]) { // wait };

Unlock: slot = mySlotIndex.get();
flag[slot] = false;
flag[(slot + 1) % size] = true;
```

Memory bus contention (bad)

Συγχρονισμός: coarse / fine grained. Coarse-grained → lock σε όλη τη δομή. Fine-grained → κλειδώνω μικρά κομμάτια της δομής.
Π.χ. σε add/remove λίστας: κλειδώνω previous + next κόμβους (για συνέπεια).
Optimistic sync: αγνοώ τα locks, διατρέχω δομή, όταν φτάσω εκεί που θέλω παίρνω locks και κάνω `validate()` (έλεγχο συνέπειας δομής).
Lazy sync: χρησιμοποιώ πεδίο marked (συνήθως bit). Όταν κάνω remove διαγράφω logically πρώτα και physically μετά τα marked πεδία.
Η `contains()` είναι non-blocking γιατί χρησιμοποιεί το marked πεδίο.
Non-blocking sync: **`find(head, key)` →** επιστρέφει ένα ζεύγος κόμβων (pred, next) όπου next.key >= key και pred.key <= key. Μέσω atomic `compareAndSet()` διαγράφει logically removed κόμβους. **`add(), remove()` →** καλούν την `find()` πριν διατρέξουν τη λίστα, χρησιμοποιούν atomics (reference + compareAndSet) για να προσθέσουν/διαγράψουν. Η `remove()` διαγράφει μονο logically.




```
S1();
fork(L1);
fork(L2);
S2();
fork(L4);
Goto L3;
L1: S3();
fork(L4);
Goto L3;
L2: S4();
L4: join(3);
S6();
Goto L5;
L3: join(2);
S5();
L5: join(2);
S7;
```

Ιδιότητες Critical Path: 1. Mutual exclusion, 2. Deadlock-Free και 3. Starvation-Free.
1+2 αναγκαίες, 3 επιθυμητή

Τοπολογίες Δικτύων

k-Cube: $d = v = \log_2 P = n$
$b = PB/2$

k-Torus: $v = k \cdot 2$
$n^k = P \to d = k \cdot \text{floor}(n/2)$
$b = 2 \cdot P^{(k-1)/k} B$

k-Grid: $v = k \cdot 2$ $\quad d = k \cdot (P^{1/k} - 1)$
$b = P^{(k-1)/k} B$

k-Tree: $v = k + 1$
$d = 2 \cdot \text{ceil}(\log_k((P+1)/2))$

$$b = \begin{cases} B & k = 2 \\ 3B & k = 3 \\ 2B & k = 4 \end{cases}$$

P: cores
d: diameter
v: node deg.
k: dims
b: bisect. bandwidth

Locks: προστατεύει τμήμα από ταυτόχρονο access
Μπαίνει μόνο το thread που κρατάει το κλειδί.
Semaphore: Ακέραιος → αρχικοποίηση, αύξηση κατά 1, ή μειωση και block αν νεα τιμη == 0.
Monitor + cond. Variable: ζεύγος (m, c) όπου m κλείδωμα και c condition variable. →

Await: yield μέχρι συνθήκη ισχύει
Signal: ξυπνάει 1 ή όλες που περιμένουν.

Clusters
- Homogeneous: ίδιες/παρόμοιες CPU
- Nonhomogen.: διαφορετικές CPU
++: οικονομικό, εύκολη συναρμολόγηση, ρύθμιση, λειτουργία
---: λιγότερο κλιμακώσιμο, μέτρια επίδοση, πιο ενεργοβόρο
Custom Built:
+++: κλιμακώσιμο, υψηλή επίδοση, χαμηλότερη κατανάλωση ενέργειας.
----: υψηλότερο κόστος
$P = CV^2 f \quad (C: \text{Count},\ V: \text{Voltage},\ f: \text{Frequency})$

Προγραμματιστικά Μοντέλα:
1) message passing 2) shared address space
Αλληλεπίδραση με υλικό:
1) shared memory systems → διευκολύνει τον παράλληλο προγραμματισμό, δύσκολη κλιμάκωση
2) distributed memory systems → δυσκολεύει προγραμματισμό, κλιμακώνει σε 1000άδες επεξεργαστές
3) hybrid (combine 1 + 2)

PRAM (parallel random access machine):
N processors/cores, unbounded shared memory uniform access from all processors
→ σοβαρές υπεραπλουστεύσεις (π.χ. ομοιομορφος χρόνος πρόσβασης στη μνήμη και άπειρη μνημη) που δίνουν ανασφαλή συμπεράσματα για εκτέλεση σε πραγματικά παράλληλα systems.
(S/M)I(S/M)D: S = Single, M = Multiple, I = Instructions, D = Data.
(E/C)R(E/C)W: exclusive/concurrent read/write

Parallel Shared Mem. Architecture:
UMA → uniform memory access (access time ανεξάρτητο θέσης μνήμη και επεξεργαστή)
NUMA → not UMA (cc-NUMA: με πρωτόκολλο συνάφειας κρυφής μνήμης – π.χ. MESI).

Ts: χρόνος σειριακού **Tp:** χρόνος παράλληλου,
P: number of processors

Strong scaling: steady problem size
Weak scaling: steady problem size per processor
Λόγοι μη-κλιμάκωσης:
1) Μεγάλο σειριακό κομμάτι
2) unbalanced load
3) synchronization / communication cost
4) memory-bound programs

f: nonparallel portion → ***Amdahl'sLaw***
$\text{Effectiveness } E = \frac{S}{P} \qquad \text{Speedup } S = \frac{T_s}{T_p}$

$T_p = fT_s + \frac{(1-f)T_s}{P}$
$T = T_{\text{comm}} + T_{\text{comp}} + T_{\text{idle}}$

$\text{Classic: } T_{\text{comp}} = \text{Ops} \cdot \text{CPU Speed}$
$\text{Roofline: } T_{\text{comp}} = \text{Ops} \cdot \max\left(\text{CPU Speed}, \frac{1}{\text{OI} \cdot BW}\right)$
$\text{OI} = \frac{\text{ops}}{\text{byte}},\ BW = \text{mem. bandwidth}$

Για πίνακες, αν ολόκληρος ο πίνακας χωράει στη cache πολλαπλασιάζουμε το OI με το μέγεθος του πίνακα, αλλιώς ως έχει (τυπικά λάθος γιατί ακόμα κι αν δε χωράει όλος, χωράνε blocks του)

Task-centric: πρώτα μοιράζω υπολογισμούς, μετά δεδομένα (δυναμικοί αλγόριθμοι, ακαθόριστες δομές).
Data-centric: πρώτα δεδομένα, μετά υπολογισμοί (κανονικές δομές δεδομένων) **Function-centric:** διαμοιρασμός με βάση συναρτήσεις (διακριτές φάσεις και ροή δεδομένων).
Task graph: ακμές → εξάρτηση ανάμεσα σε task
Βάρος ακμής → δεδομένα προς μεταφορά
PGAS → partitioned global address space. Συνδυάζει μοντέλα ανταλλαγής μηνυμάτων και χώρου διευθύνσεων (και local και shared address space).
Loop-dependency matrix: rows → loops, columns → reference variables. 0 → no dependence, 1 → dependence on former executions, -1 → dep. on next executions, * if both.
Parallel loops: μόνο αν ολόκληρη η γραμμή είναι 0 ή όλες οι αποπάνω έχουν θετικό πρώτο nonzero στοιχείο.
MESI:
M → modified, E → exclusive, S → shared, I → invalid
Cache coherence: true sharing → αναφορά στο ίδιο δεδομένο, false sharing → αναφορά στο ίδιο cache line αλλά όχι στο ίδιο δεδομένο
Π.χ. → array padding στο queue lock για να αποφύγουμε το false sharing με τα άλλα threads.

MPI

```
for (i = 0; i < grid[1]; ++i) {
   for (j = 0; j < grid[0]; ++j) {
       disps[i * grid[1] + j] = offset;
       offset += local[1];
   }
   offset += global_padded[1] * (local[0]-1);
}
MPI_Scatterv(sendbuf, scounts, disps, global_block,
&(u_previous[1][1]), 1, local_block, 0, CART_COMM);
north = rank – grid[1]; south = rank + grid[1];
west = rank – 1; east = rank + 1;

if ((rank % (grid[0]*grid[1])) < grid[1]) north = -1;
if ((rank % (grid[0]*grid[1])) >= ((grid[0] – 1) *
grid[1])) south = -1;
if ((rank % grid[1]) == 0) west = -1;
if ((rank % grid[1]) == grid[1] – 1)) east = -1;

if (north != -1) {
MPI_Send(&u_previous[1][1], 1, send_row, north, t,
CART_COMM);
MPI_Recv(&u_previous[0][1], 1, send_row, north, t,
MPI_STATUS_IGNORE);
}
if (south != -1) {
MPI_Send(&u_previous[local[0]][1], 1, send_row, south,
t, CART_COMM);
MPI_Recv(&u_previous[local[0]+1][1], 1, send_row, south,
t, MPI_STATUS_IGNORE);
}
if (west != -1) {
MPI_Send(&u_previous[1][1], 1, send_column, north, t,
CART_COMM);
MPI_Recv(&u_previous[1][0], 1, send_column, north, t,
MPI_STATUS_IGNORE);
}
if (east != -1) {
MPI_Send(&u_previous[local[1]][1], 1, send_column, east,
t, CART_COMM);
MPI_Recv(&u_previous[local[1]+1][1], 1, send_column,
east, t, MPI_STATUS_IGNORE);
}
```

$t_s: \text{startup time}, t_h: \text{hop time}, t_w: \text{bandwidth time}$
$T_{\text{comm}} = t_s + l t_h + m t_w \ (m: \text{msg size}, l: \text{num. hops})$
συνήθως δεύτερος όρος μικρός, παραλείπεται, άρα →
$T_{\text{comm}} = P(t_s + m t_w) \quad (\text{in tree: } \log_2 P(t_s + m t_w))$

Δυσκολία επικοινωνίας ανάλογα με μέγεθος μνμτος, ανταγωνισμό εφαρμογών, κόμβων και απόσταση κόμβων.
Άεργος χρόνος λόγω ανισοκατανομής φορτίου ή αναμονής για δεδομένα.
$T_\infty: \text{execution of largest critical path} \rightarrow \frac{T_1}{T_\infty}$ Max speedup