GPU COMPUTING
LECTURE 05 - PARALLEL COMPUTING

Kazem Shekofteh
Kazem.shekofteh@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg
Inspired from lectures by Holger Fröning

PARALLELISM

OBVIOUS TRENDS

- Sequential vs. parallel processing completely different
- Multi-/Many-core era
    - Applications designed for single-core
    - Concurrency is fundamental for algorithms and applications
- Number of cores/CPU increasing
    - Scalability also fundamental
- Further motivations
    - Performance increase, distributed systems, tolerating I/O Blocking

Parallel programming: Concurrency & Scalability (I & II)

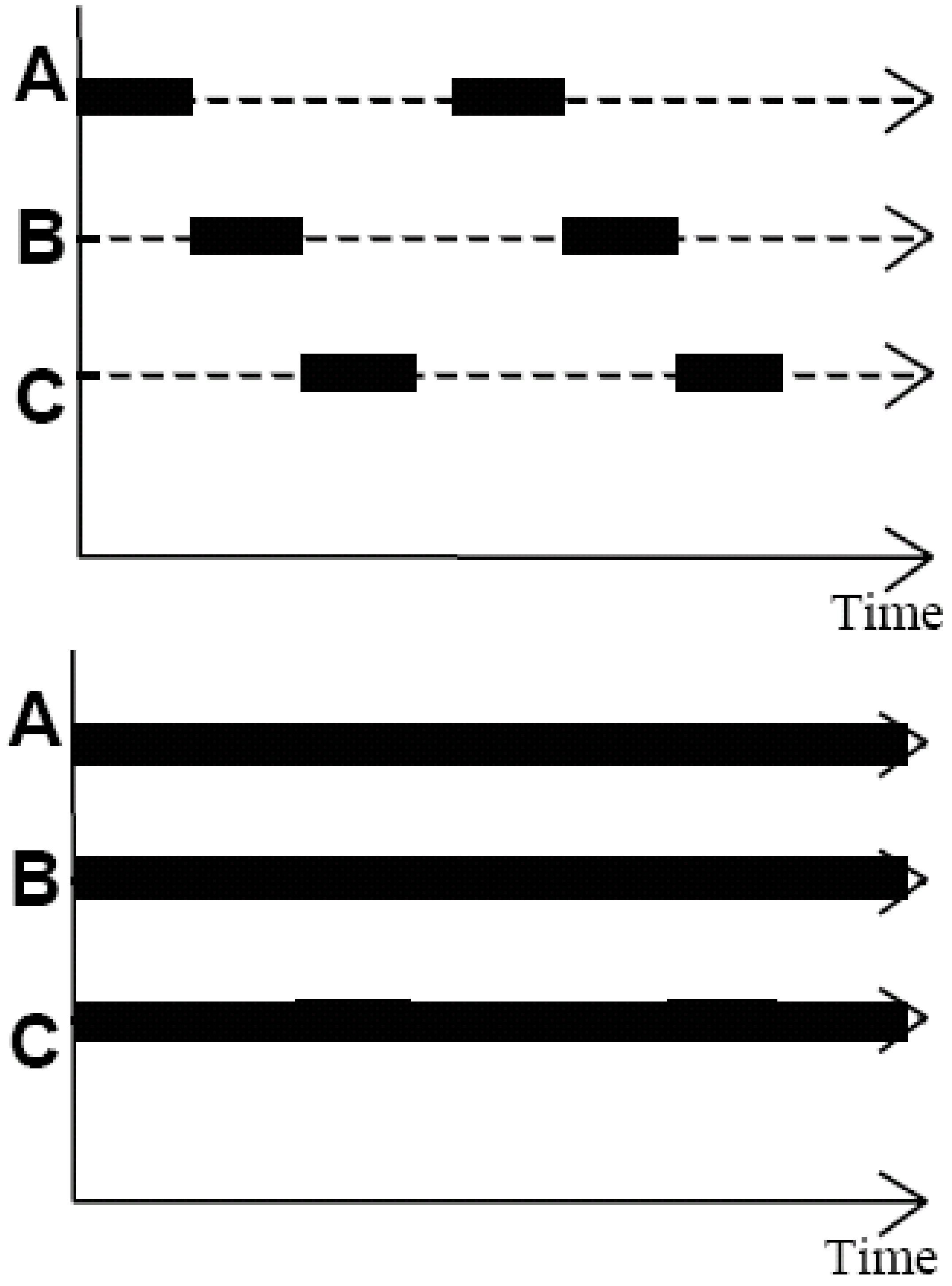
CONCURRENCY

Sequential program

- Single thread of control
- Instructions executed sequentially

Concurrent program

- Several autonomous sequential threads
- Parallel execution possible
- Execution determined by implementation



Implementations

- Multi-programming: executing multiple threads on a single resource (interleaving)
- Multi-processing: executing multiple threads on multiple resources (independent of scale)

CONCURRENCY VS. PARALLELISM

Concurrency is not parallelism!

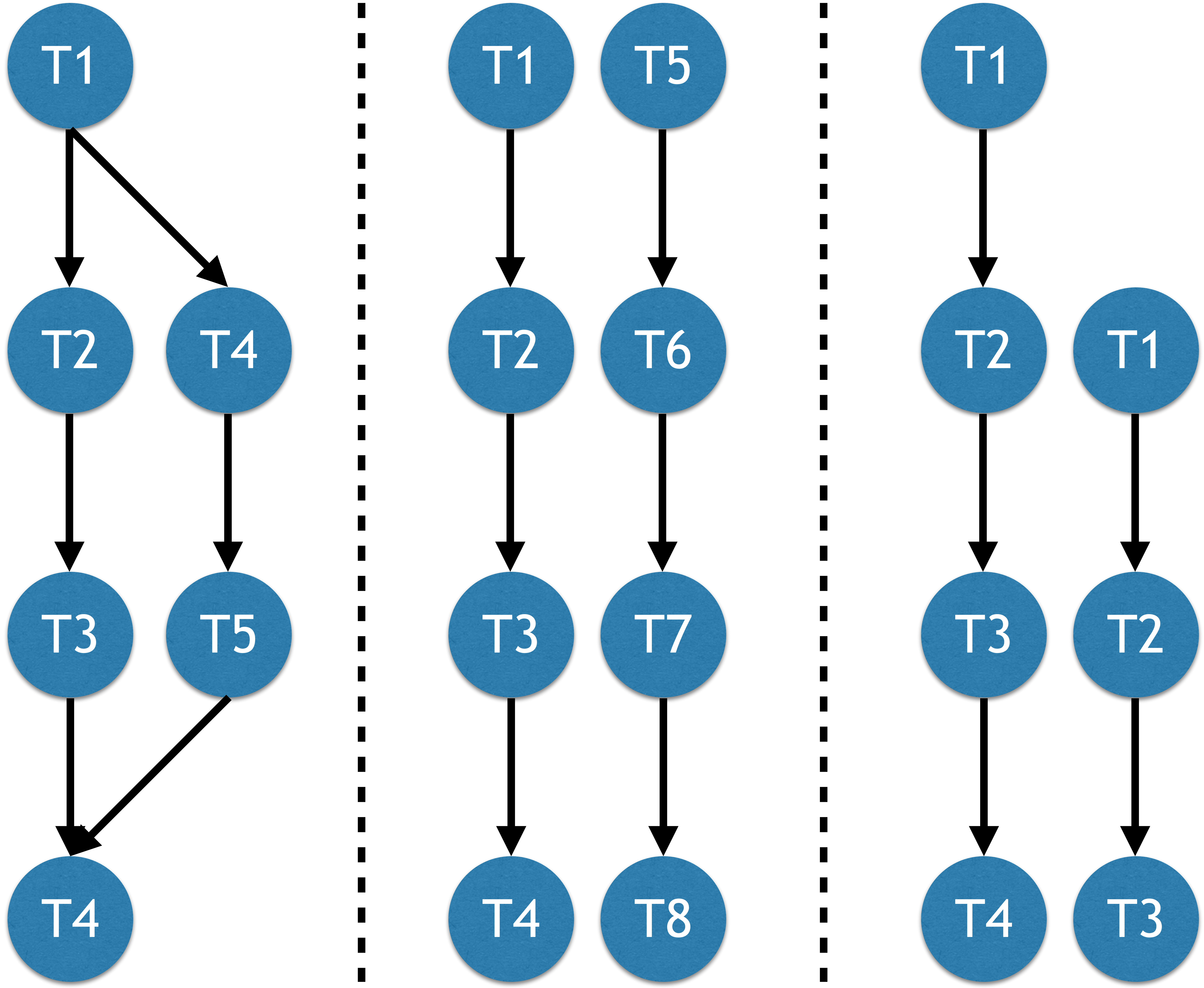
- E.g., concurrency by interleaving

Concurrency = independency

- Concurrent instruction streams can be executed independently
- E.g., in parallel

Pipelining versus replication

- in-order vs. out-of-order pipelining



LEVELS OF (HW) PARALLELISM MODERN APPROACH

Instruction Level Parallelism (ILP)

- Parallelism of one instruction stream
- Huge amount of dependencies and branches
- Limited parallelism (~4-6)

Thread Level Parallelism (TLP)

- Parallelism of multiple independent instruction streams
- Less amount of dependencies, no limitations due to branches
- Limited by the maximum number of concurrently executable I-streams

Data Level Parallelism (DLP)

- Applying one operation on multiple independent elements
- Parallelism depends on data structure
- Vectorization techniques

Request Level Parallelism (RLP)

- Datacenter (Warehouse-scale computers)
- Many requests from many users

PROCESSOR EXAMPLES

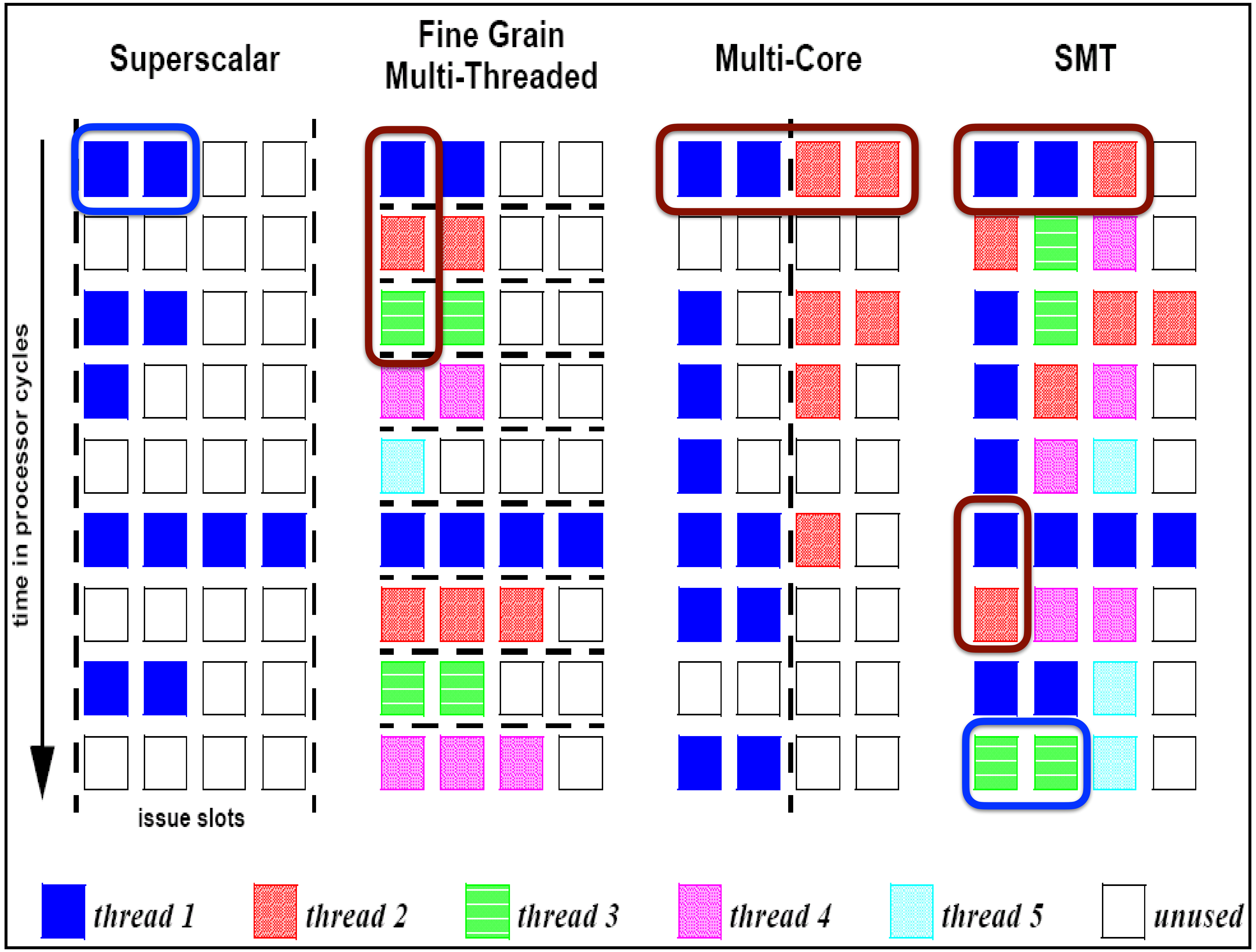
Exploiting parallelism in different architectures

- Issue slots shown
- Dashed line: partition boundary
- Horizontal waste
- Vertical waste

ILP: parallelism from the same thread

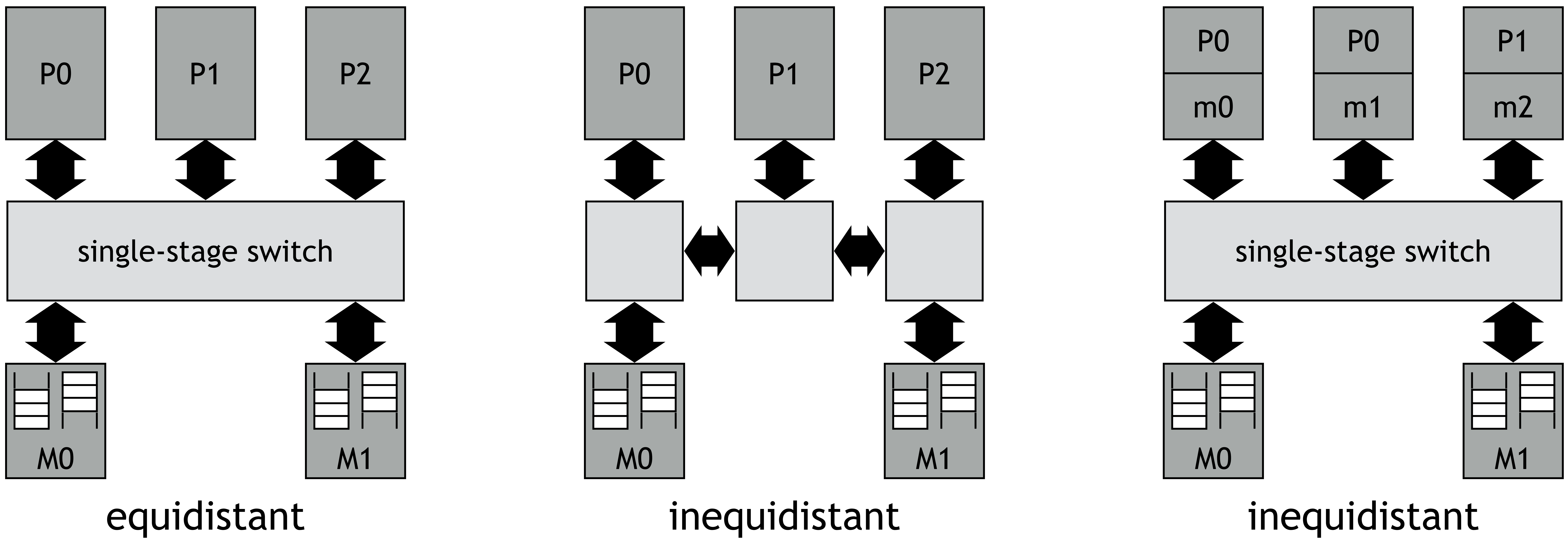
TLP: parallelism from different threads

DLP: parallelism from multiple data elements



COMMUNICATION AND COMPUTE MODEL

NON-UNIFORM MEMORY ACCESS



Parallel programming: Locality (III)

COMMUNICATION MODELS

Plain load/store (LD/ST) - shared memory systems

- Never designed for communication
- Can be fast for SMP, but often unknown costs for NUMA
- Assumption of perfectly timed load seeing a store

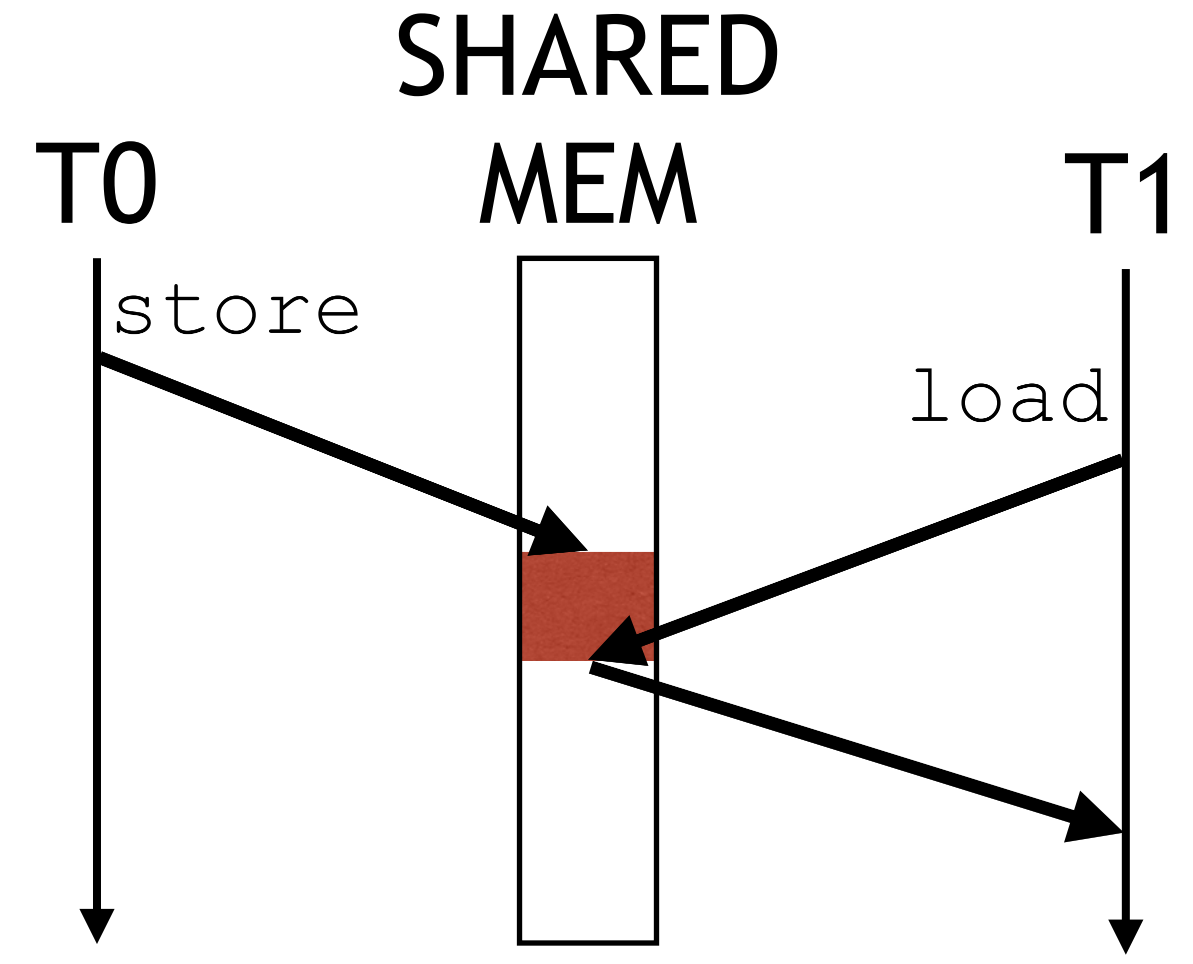
Message passing (MP) - de-facto standard in HPC

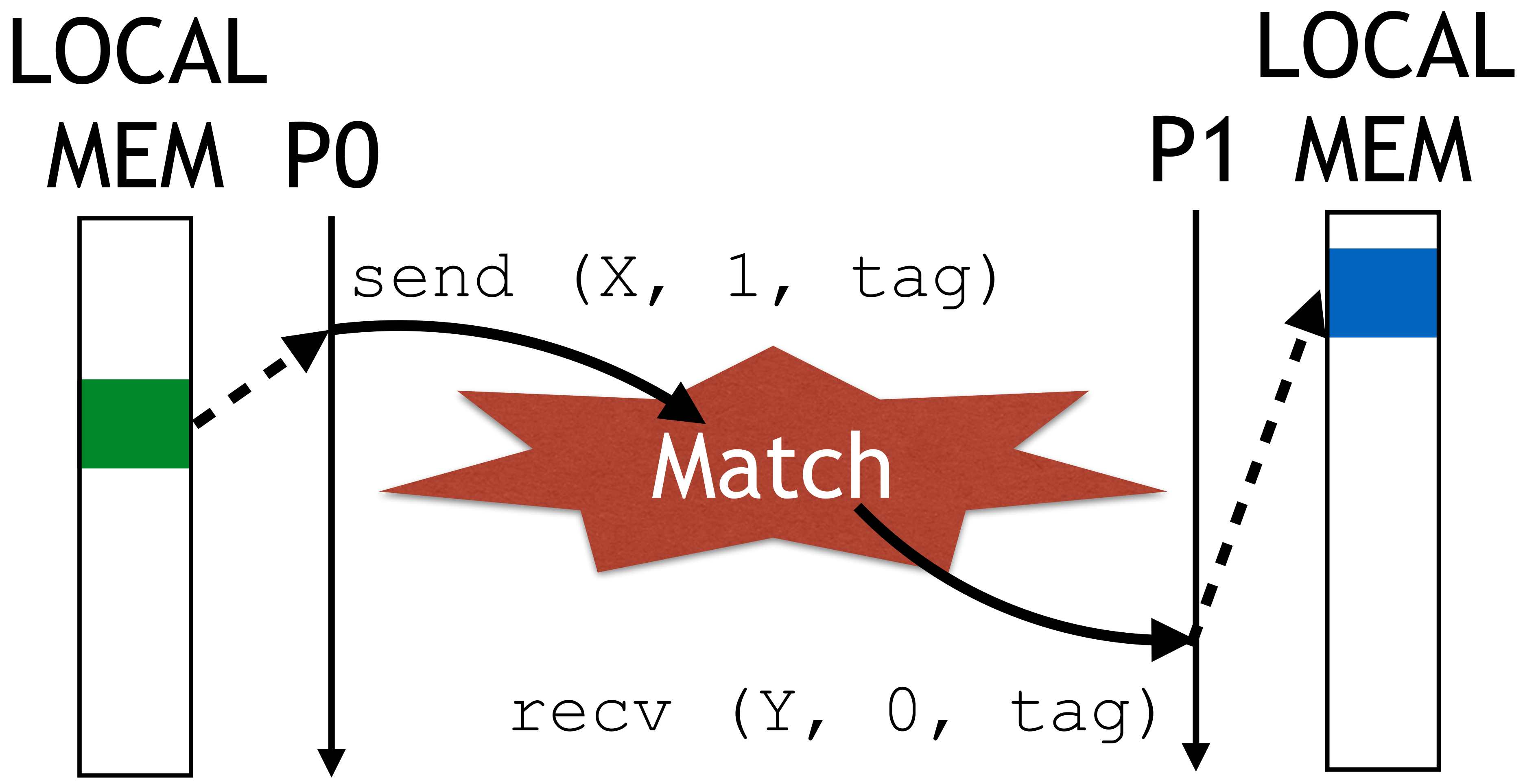
- Various p2p and collective functions
- Mainly send/recv semantics used - ease-of-use
- Overhead due to: copying, matching, progress, ordering

Many more

- Active messages - latency tolerance becomes a programming/compiling concern
- One-sided communication (put/get) - never say receive

Main objective: latency tolerance using overlap





LATENCY TOLERANCE TECHNIQUES

Property	Relaxed Consistency Models	Prefetching	Multi-Threading	Block Data Transfer
Types of latency tolerated	Write (blocking read processors) Read and write (dynamically scheduled processors)	Write Read	Write Read Synchronization	Write Read
Software requirements	Labeling synchronization operations	Predictability	Explicit extra concurrency	Identifying and orchestrating block transfers
Extra hardware support	Little	Little	Substantial	Not in processor, but in memory system
Supported in commercial systems?	Yes	Yes	Yes	(Yes)

David E. Culler, Jaswinder Pal Singh, Anoop Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, 1998

SYNCHRONIZATION

Synchronization is the enforcement of a defined logical order between events. This establishes a defined time-relation between distinct places, thus defining their behavior in time.

- Foundation: dependencies that are being solved using synchronization
  - Communication can include synchronization, but not vice versa
- Communication & synchronization
  - Explicit / implicit
- SIMD: one instruction stream, no synchronization necessary
  - Reminder: vector packing requires reasoning about dependencies (resp. their absence)
- MIMD: synchronization necessary
  - Shared variables, process synchronization, blocking message exchange

COMPUTE MODEL

No one wants to write N programs for N processors

- Reminder: scalability

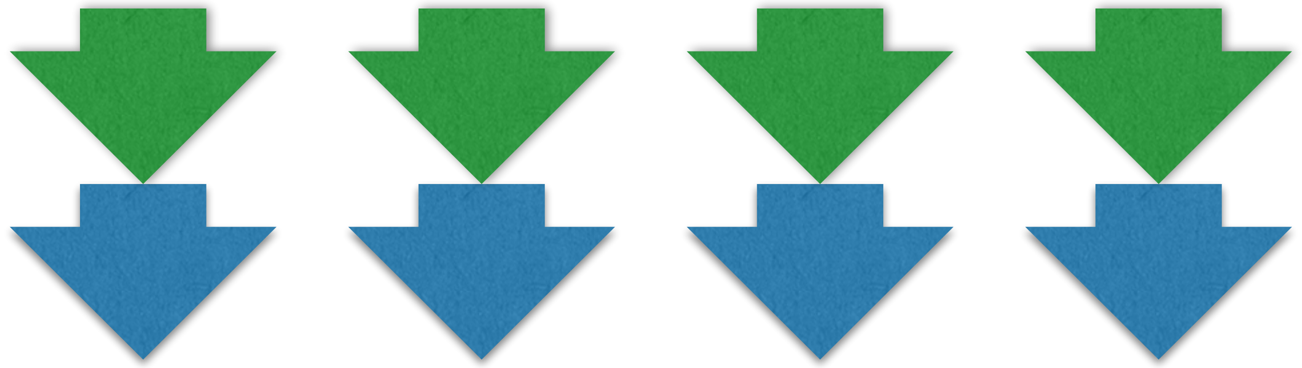
Single-Program-Multiple Data (SPMD)

- Single program that distinguishes different tasks based on task ID
- Producer-Consumer, Master-Slave, Peer

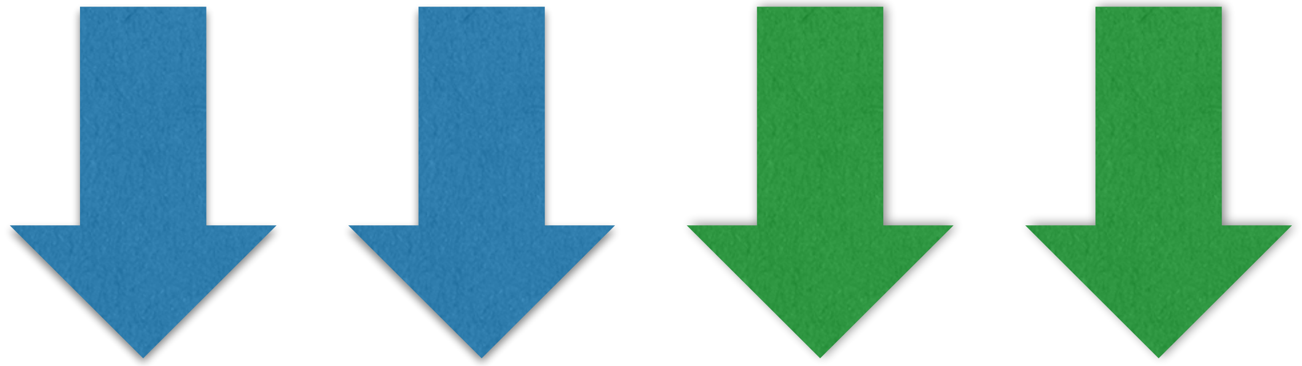
Composition

- Sequential composition: data-parallel languages or SIMD
- Parallel composition: different modules operate on disjoint sets of processors (e.g., MPI)
- Concurrent composition: different modules can operate on the same processors, and execution depends only on availability of data

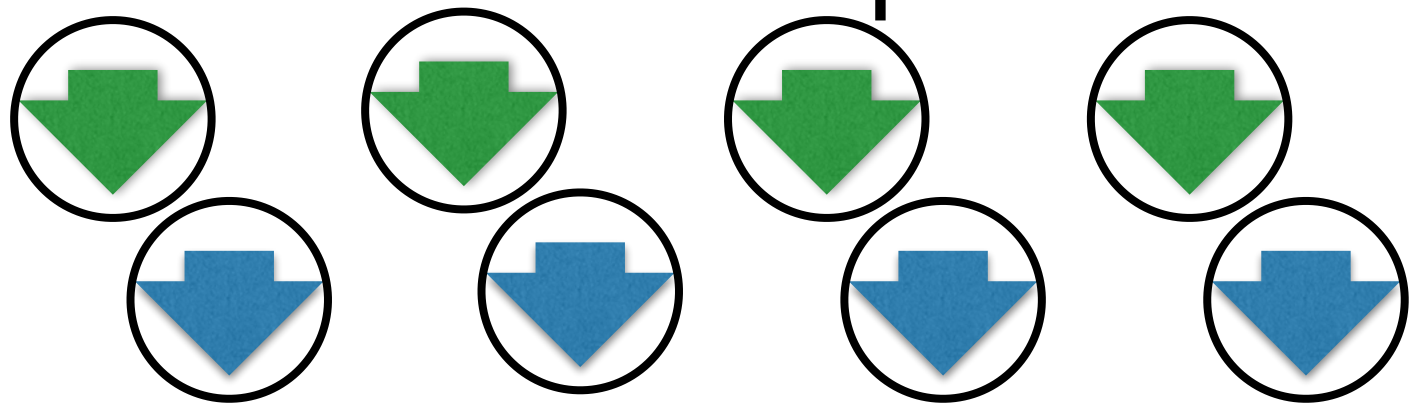
Sequential composition



Parallel composition



Concurrent composition



PARALLELISM

Parallel programming: Concurrency & Scalability (I & II)

Parallel programming: Locality (III)

Parallel programming: Modularity (IV)

ALGORITHM DESIGN

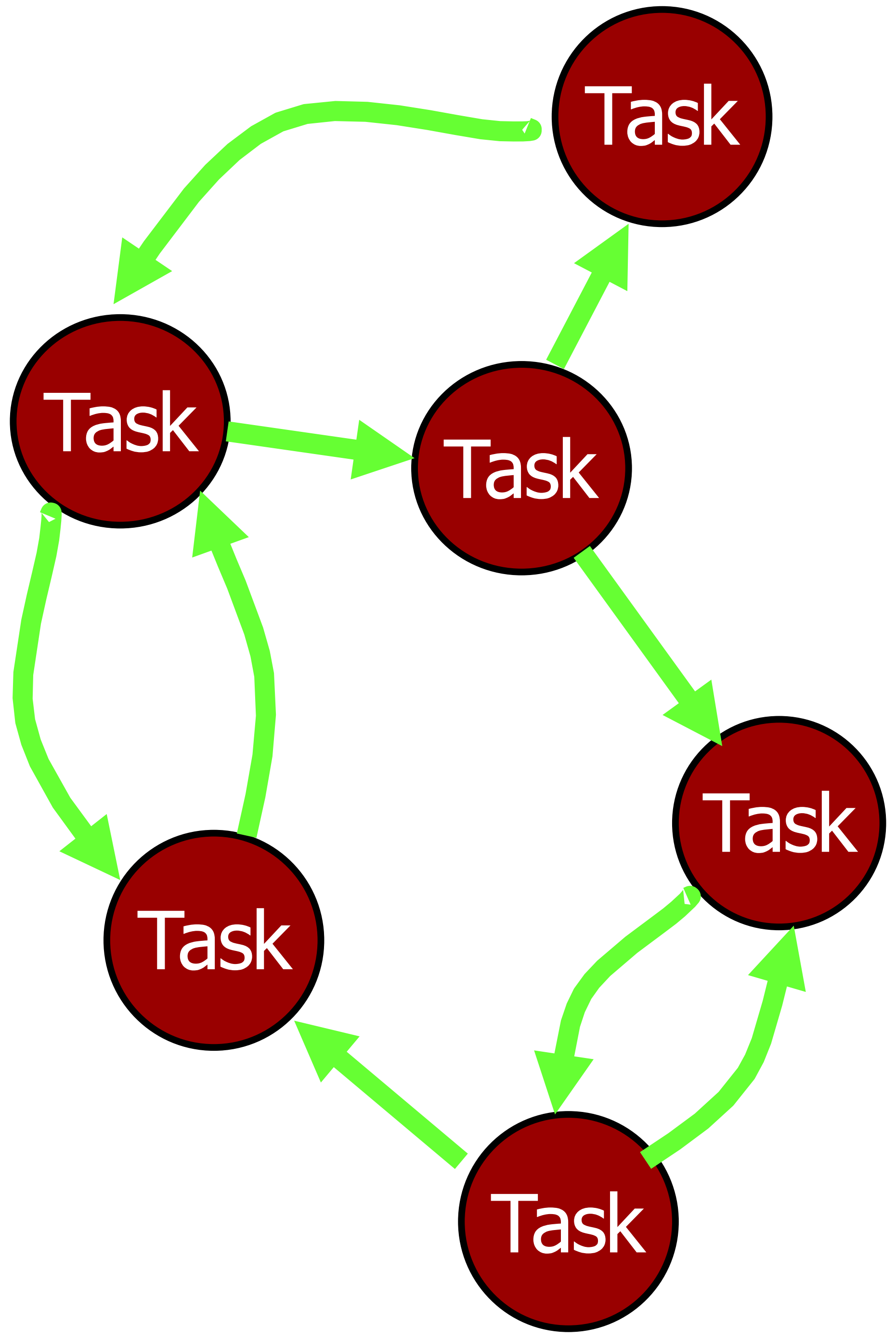
FOSTER'S PCAM

Partition

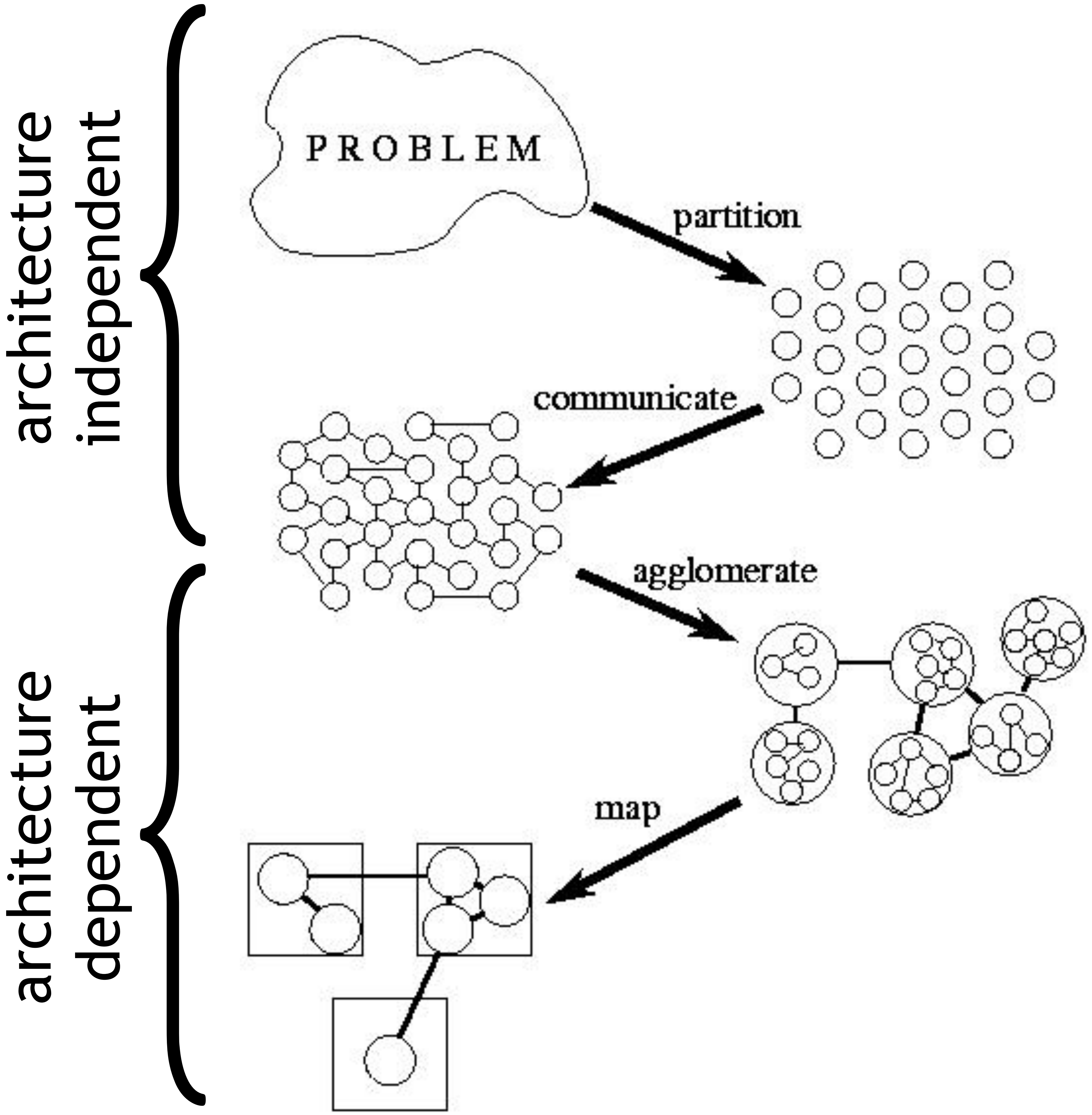
Communicate

Agglomerate

Map



Book is online at:
http://www.mcs.anl.gov/~itf/dbpp



PARTITIONING

Ignore technical aspects like number of processing units

Maximal granularity

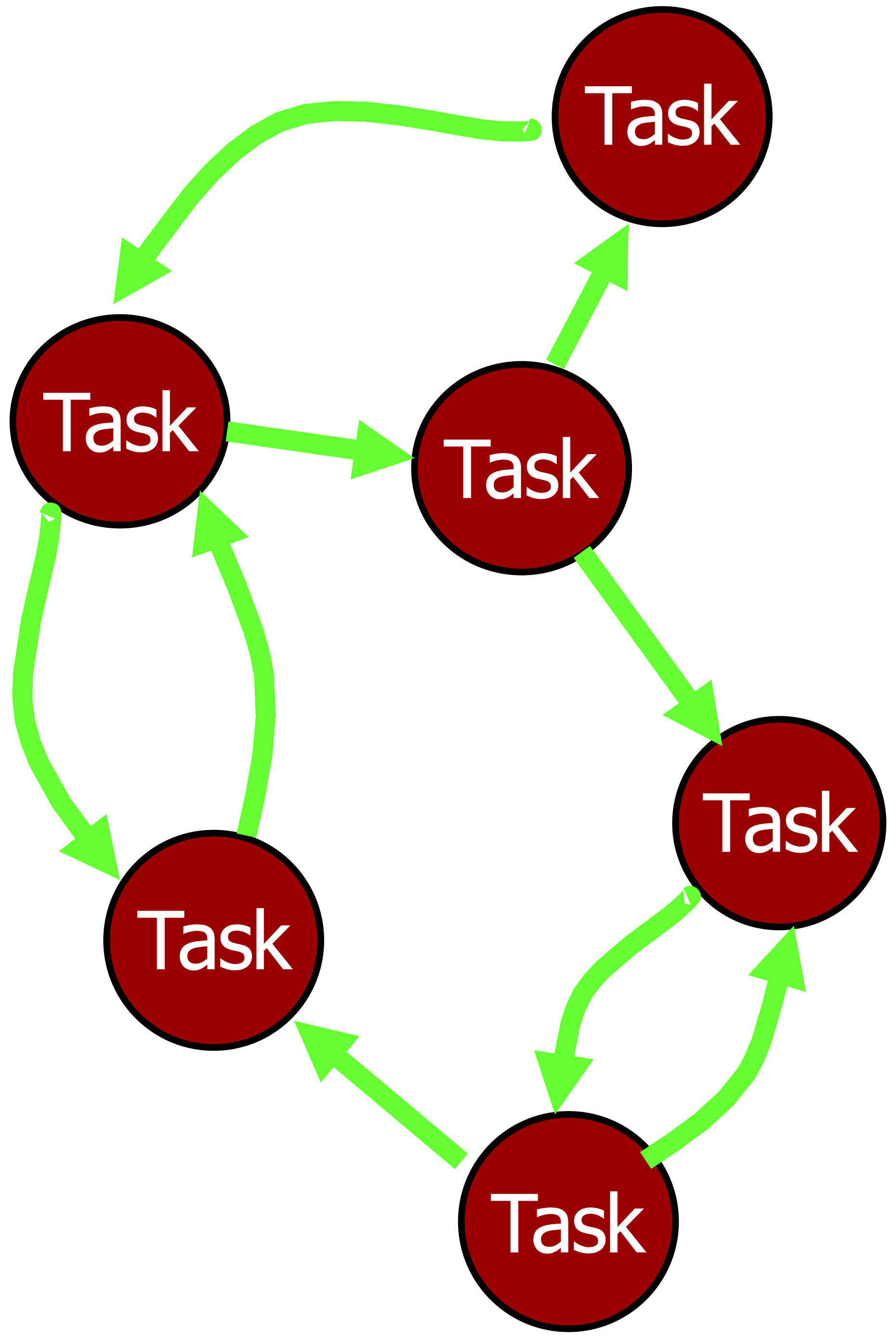
- Number of Tasks >> Number of Processors

Partition computation and data

- Domain Decomposition
- Functional Decomposition
- Pipeline Decomposition

Avoid replication, disjoint partitioning

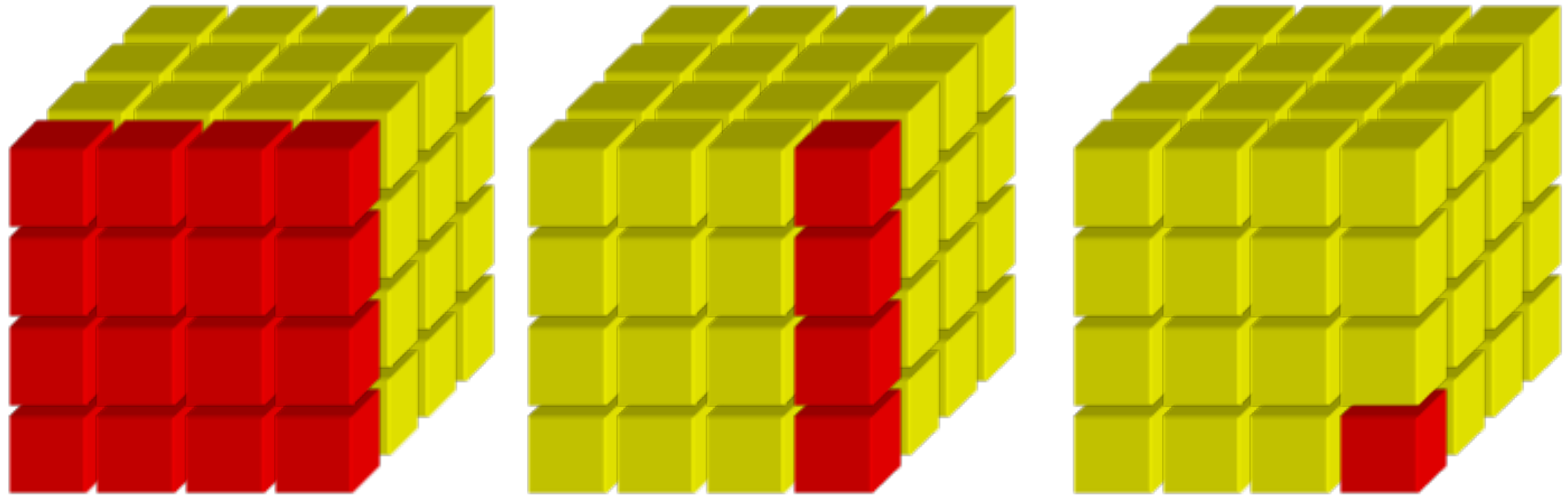
- See also minimization of communication



PARTITIONING

Domain Decomposition

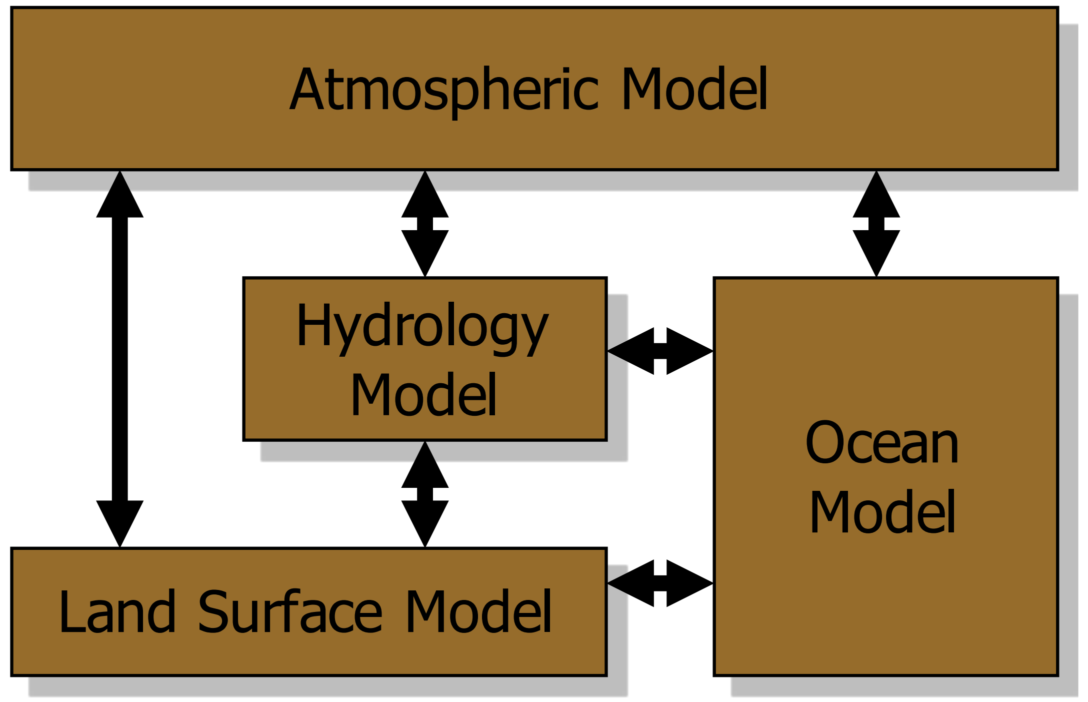
Typical uses: data parallelism, e.g. arrays & trees



Functional Decomposition

Typical uses: function calls, different loop iterations

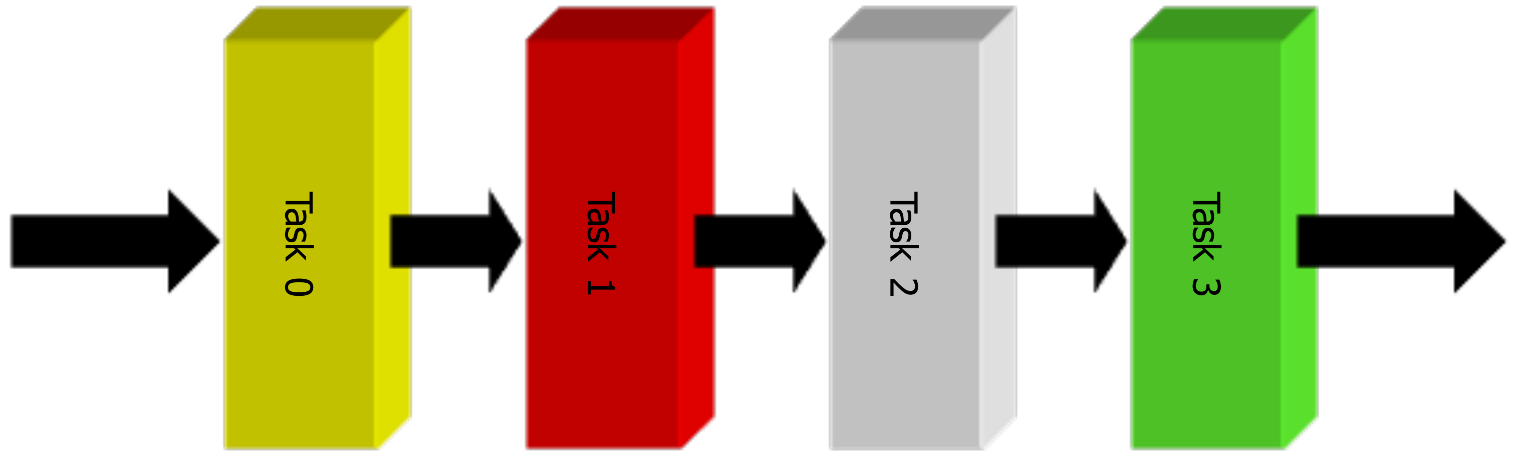
Climate Computing Model



Pipeline Decomposition

Data flow through multiple pipeline stages

Instruction pipelining in modern CPUs



COMMUNICATE

Execution of partitions concurrently, but not independently

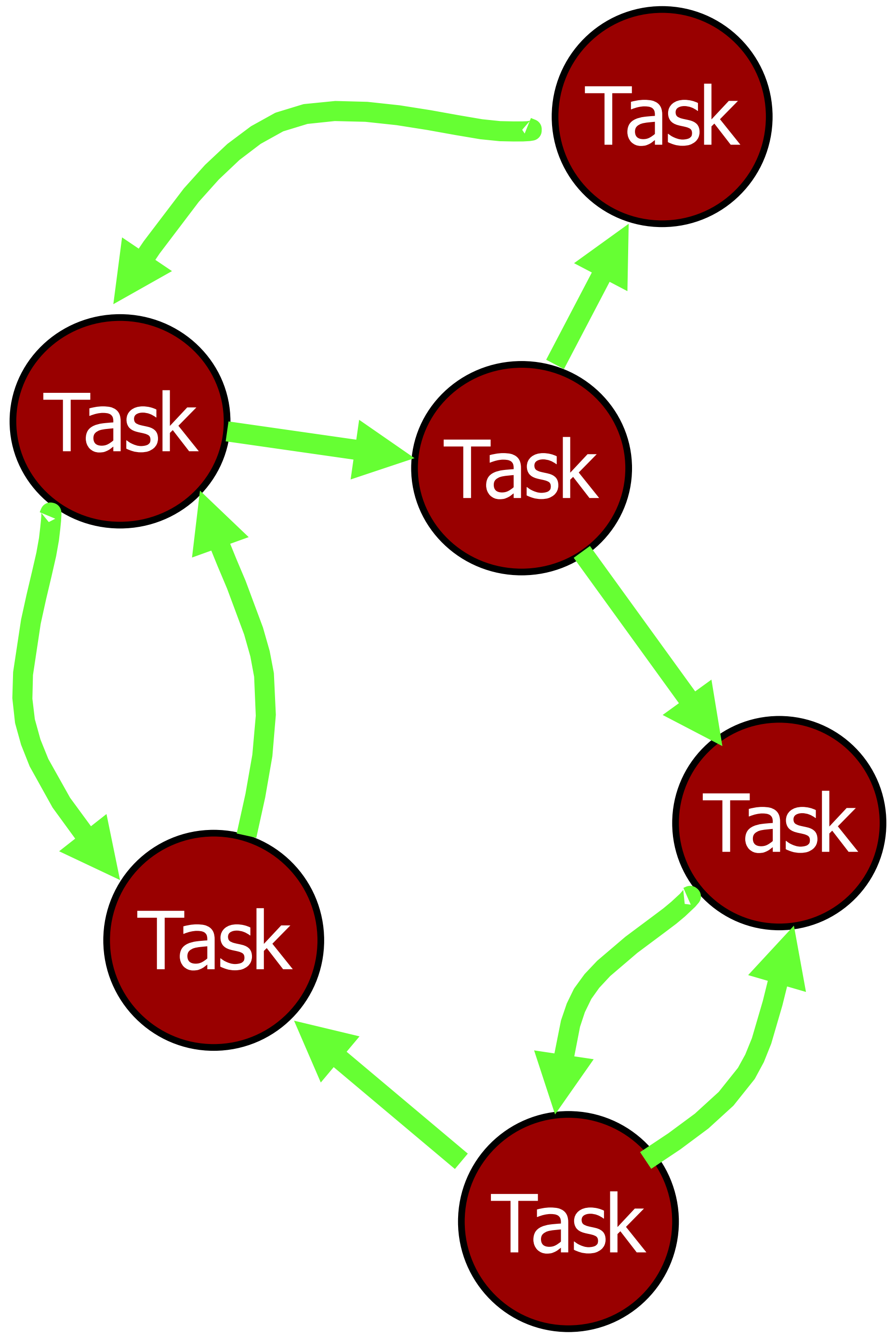
Data dependencies -> communication & synchronization

- Complex for DD, rather simple for FD
- Local/global, structured/unstructured, static/dynamic, synchronous/asynchronous

=> Communication scheme

Data-parallel language

- Requires data-parallel operations and data distribution
- Channels actually not necessary, but help for locality and communication costs

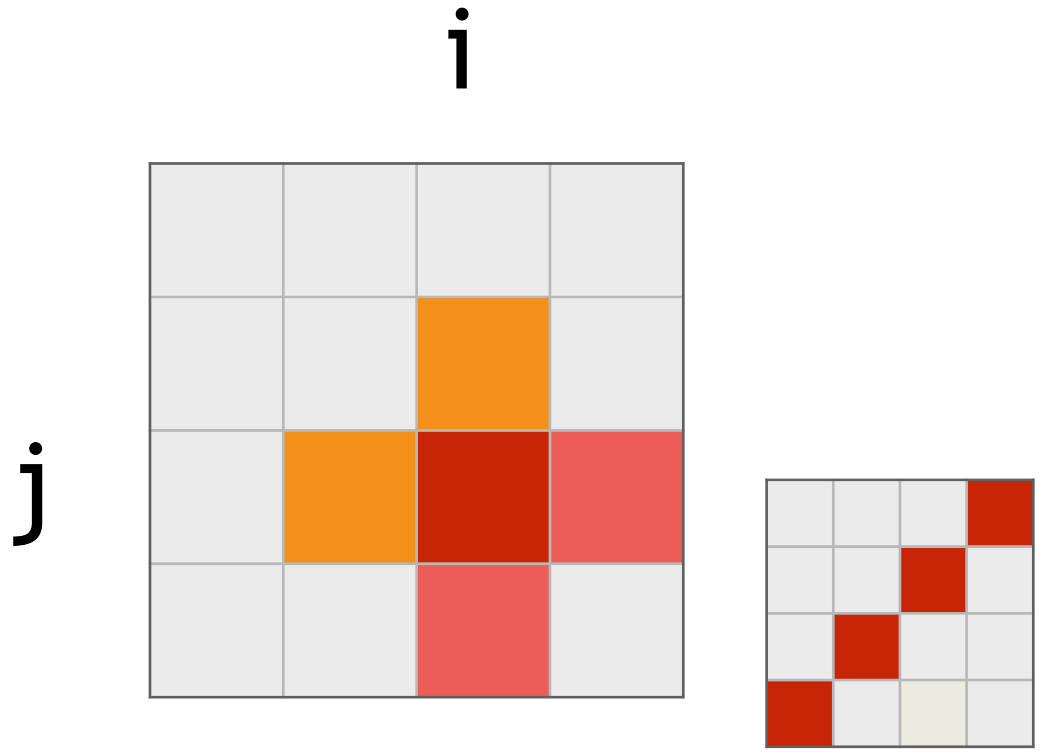


LOCAL COMMUNICATION

Example for local communication: stencil operation

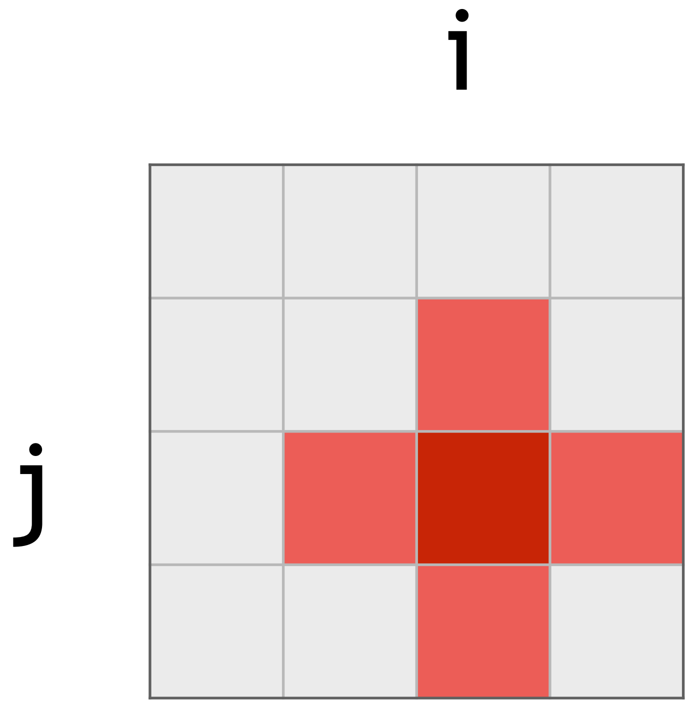
- Simple numerical computation: finite difference method (iterative method used to solve a linear system of equations)
- Gauss-Seidel (GS)

$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t+1)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t+1)} + X_{i,j+1}^{(t)}}{8}$$



- vs. Jacobi

$$X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)}}{8}$$

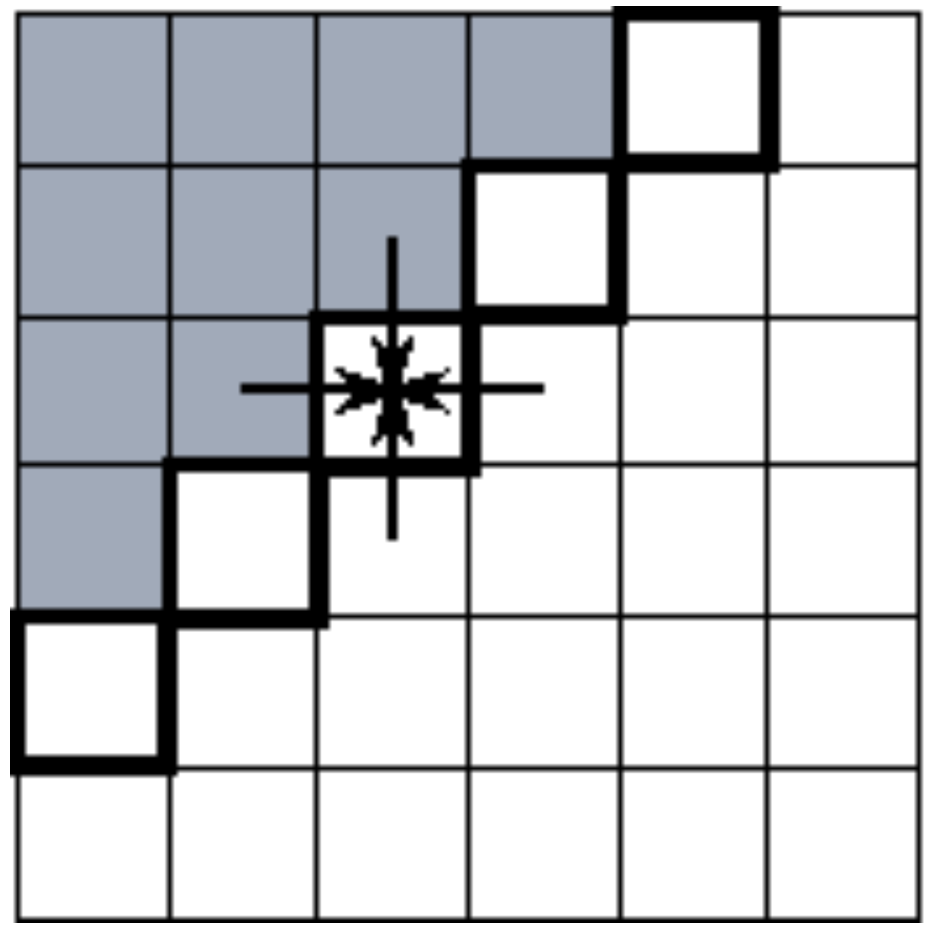


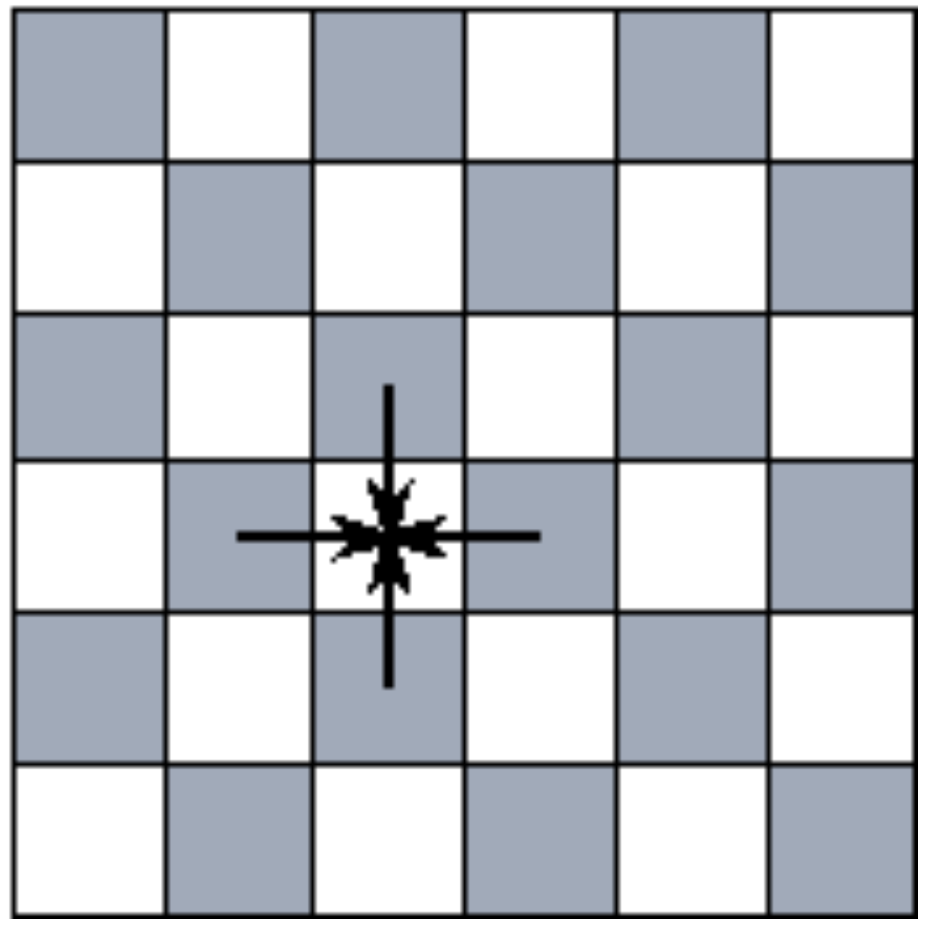
GS optimal for sequential execution (fewer iterations)

- But too many dependencies for parallel execution
- Diagonal wave front or Red/Black method

Jacobi: no inter-iteration dependencies => unconstrained parallelization

LOCAL COMMUNICATION EXAMPLE





Two finite difference update strategies, here applied on a two-dimensional grid with a five-point stencil. In both figures, shaded grid points have already been updated to step t+1 ; unshaded grid points are still at step t . The arrows show data dependencies for one of the latter points. The figure on the left illustrates a simple Gauss-Seidel scheme and highlights the five grid points that can be updated at a particular point in time. In this scheme, the update proceeds in a wavefront from the top left corner to the bottom right. On the right, we show a red-black update scheme. Here, all the grid points at step t can be updated concurrently.
[http://www.mcs.anl.gov/~itf/dbpp]

Excellent example that code optimized for sequential execution often has to be completely rewritten

GLOBAL COMMUNICATION

Global communication

E.g. global addition (parallel reduction)

$$S = \sum_{i=0}^{N-1} X_i$$

Cons: O(N), centralized & sequential

More equal distribution of computation and communication, O(N-1)

$$S_i = X_i + S_{i-1}$$

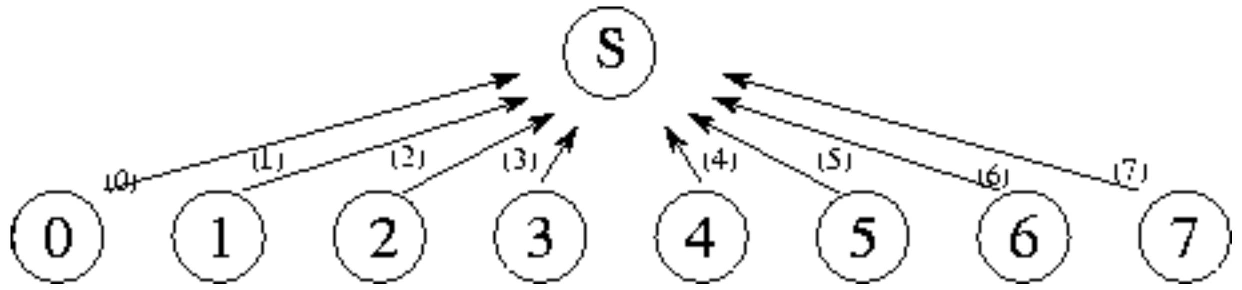
Divide & conquer to exploit parallelism

Tree structures, as long as partitions can be computed independently

Associativity of addition, O(log N)

GLOBAL COMMUNICATION EXAMPLE

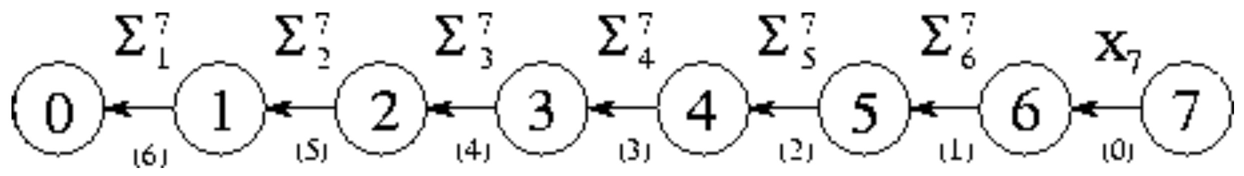
$$S = \sum_{i=0}^{N-1} X_i$$



Approach 1

central accumulator

$$S_i = X_i + S_{i-1}$$

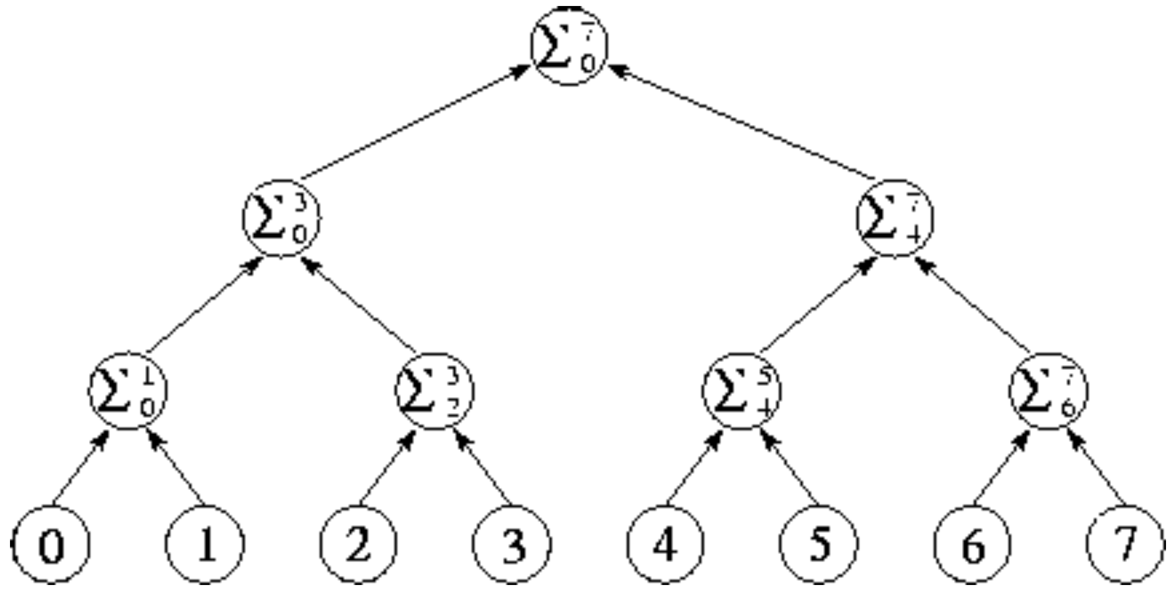


Approach 2

Array structure of N tasks

(improved pipelining possibilities)

$$\sum_{i=0}^{2^n-1} = \sum_{i=0}^{2^{(n-1)}-1} + \sum_{i=2^{(n-1)}}^{2^n-1}$$



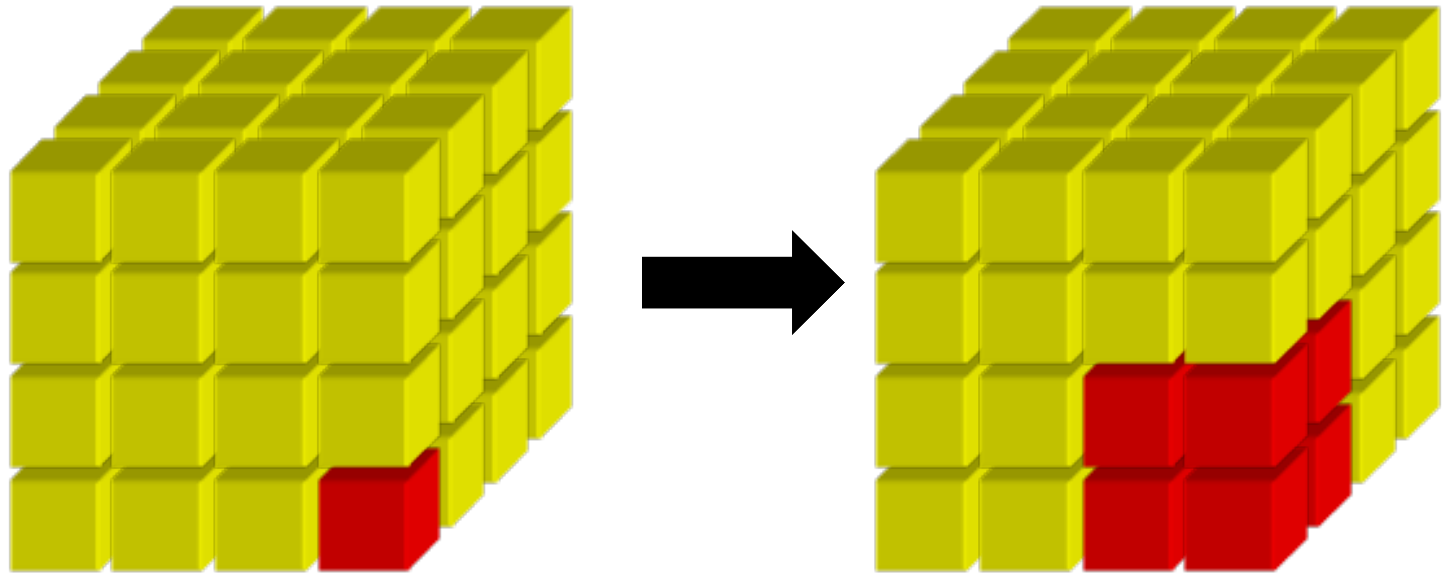
Approach 3

divide-and-conquer

(increase parallelism)

http://www.mcs.anl.gov/~itf/dbpp

AGGLOMERATION



Increasing granularity (coarse-grain)

- From the abstract to the concrete
- Fixing the parallel computing model
- Maintaining flexibility, therefore reducing development costs
- Number of tasks T >= number of processors P

Reducing communication costs

- Fixed & variable fraction (surface-to-volume effects)

Depending on use case:

- One order of magnitude more Ts than Ps (parallel slackness)
- HPC: T = P
- SIMD: T = 1

AGGLOMERATION EXAMPLE

Replication of data and computation to reduce communication

Example: global sum with broadcast

$$S = \sum_{i=0}^{N-1} X_i$$

Chained

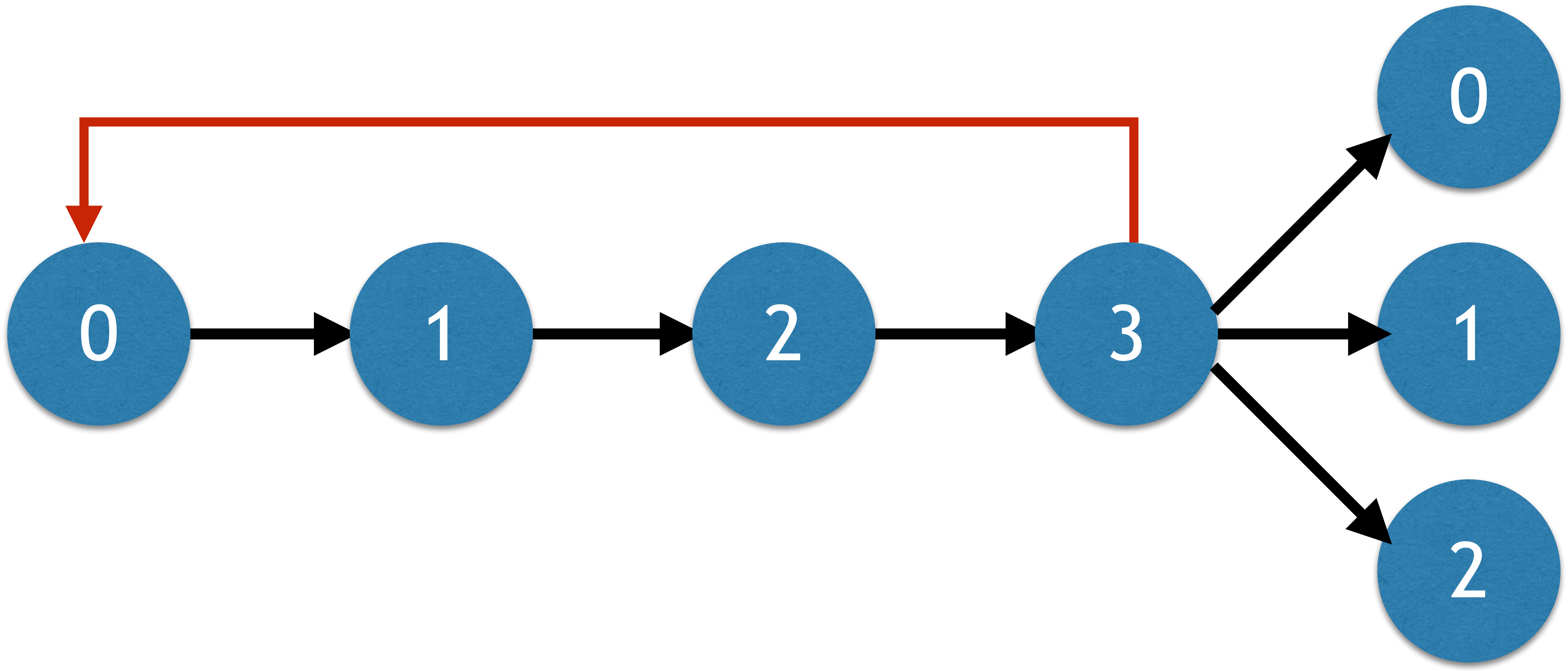
2(N-1) steps (sum & broadcast)

-> Redundant computation in a ring, no broadcast ((N-1))

Tree-based

2 log N steps (sum & broadcast)

-> Redundant computation in a butterfly, no broadcast (log N)



MAPPING

Assignment: task <—> processor & memory

- Place tasks that can execute concurrently on different processors
- Place tasks that communicate frequently on the same processor
- Note that this implies conflicts

Mapping not necessary for

- Uni-processors or shared memory systems with automatic mapping
- Hardware mechanism or the OS responsible for scheduling

Mapping problem is NP-complete

SUMMARY

SUMMARY

Concurrency and parallelism of fundamental importance

- Granularity
- ILP, TLP, DLP

Characteristics of “good” parallel programs

- Concurrency, Scalability, Locality and Modularity

Algorithm design

- PCAM: Partition, Communicate, Agglomerate, Map

Literature

- Foster Online: http://www.mcs.anl.gov/~itf/dbpp

APPENDIX

PARTITIONING

Identify possible decomposition techniques

Domain Decomposition (red)

Functional Decomposition (green)

Pipeline Decomposition (blue)