RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Introduction to High Performance Computing

Lecture 04 – Parallel Computing

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg

Parallelism

Motivation

- 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

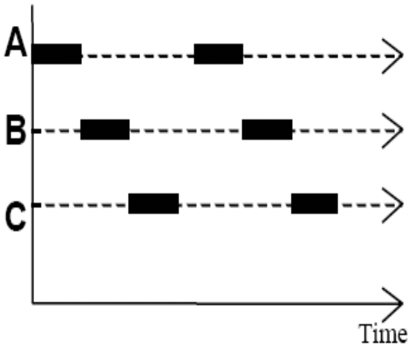
- Several autonomous sequential threads
- Parallel execution
- Execution determined by implementation

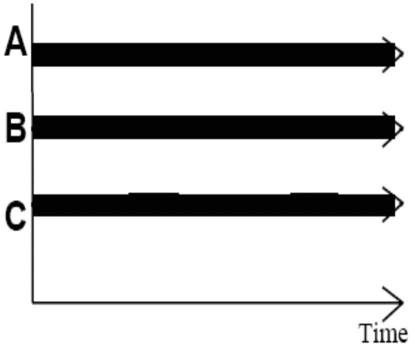
Implementations

- Multi-programming
  - Executing multiple threads on one single resource
  - Time-multiplexing
- Multi-processing
  - Executing multiple threads on one multiple resources
  - Multi-processor, Multi-core
- Distributed processing
  - Executing multiple threads on one multiple independent resources
  - Cluster, Grid, Cloud

Concurrency vs. Parallelism

- Concurrency is not (only) parallelism!
- Concurrency by interleaving
  - Only logical "parallel" execution on one single resource
  - Appearance of "simultaneous" execution
- Parallelism
  - True parallel, simultaneous execution
  - Requires several, parallel resources
- Example for concurrency:
  - Multiple ATMs ("EC-Automaten") and account balance

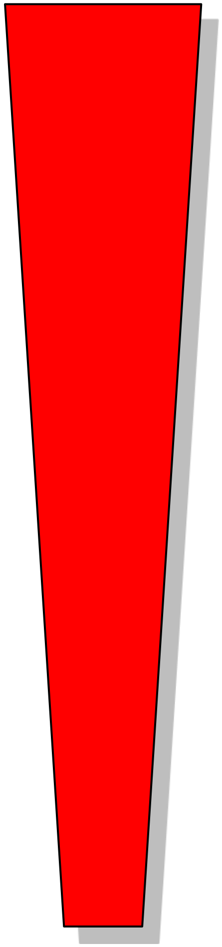




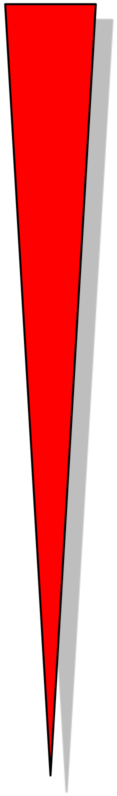
Error-free execution on sequential hardware not necessarily implies error-free execution on parallel hardware

Levels of Parallelism – Traditional Approach

- **Program level**
  - Coarse grained
  - Concurrent execution of multiple programs, or of a single program with different input data sets
- **Procedure level**
  - Medium grained
  - Different parts of a program are executed concurrently on different parts of a computing system, or one single part with different input data sets
- **Instruction level**
  - Fine grained
  - Concurrent computation of multiple variables in one procedure



- **Microcode level**
  - More fine grained
  - Instruction → Operation**s** (Phases)
  - Execution of different phases of different instruction in different pipeline stages or superscalar execution units simultaneously
- **Bit level**
  - Extreme fine grained
  - Processing of words only, consisting of multiple bits

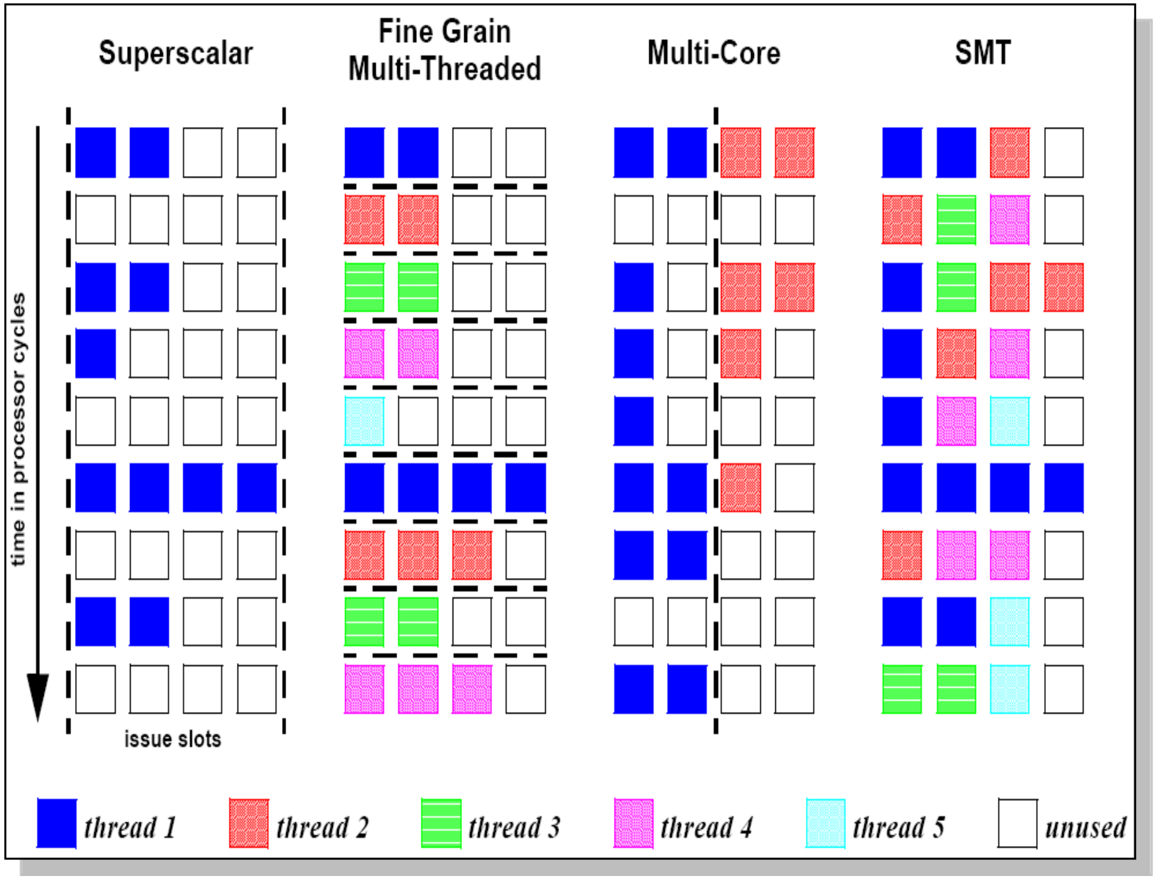


Levels of Parallelism – Modern Approach

- 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 maximal concurrently executable I-streams
- Data Level Parallelism (**DLP**)
  - Vectorization techniques
  - Applying one operation on multiple elements of a data structure
  - Parallelism dependent on data structure

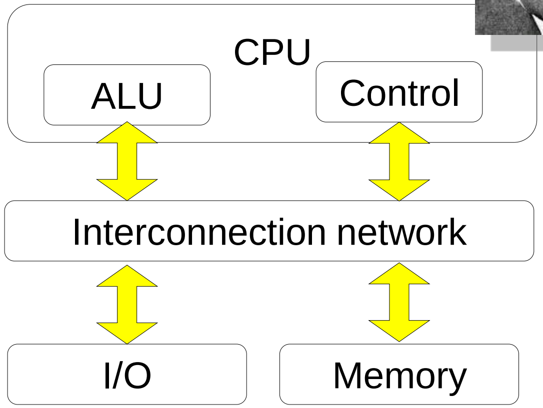
Levels of Parallelism (3)

- Exploiting parallelism in different CPU architectures
  - ILP
  - TLP
  - Why no DLP?
- What about GPUs?



Computing Model

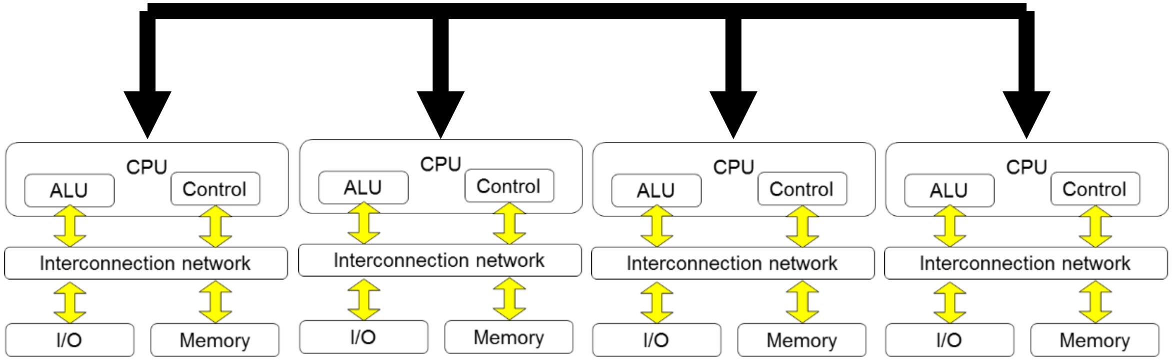
Computing model (1)



- von-Neumann architecture
  - Main units
    - CPU (Control & Compute)
    - I/O
    - Memory
  - “Node” for HPC systems
- von-Neumann bottleneck
  - ALU faster than memory
  - Costs for control and communication higher than computing costs
- Harvard Architecture: Separation of data and instruction memory

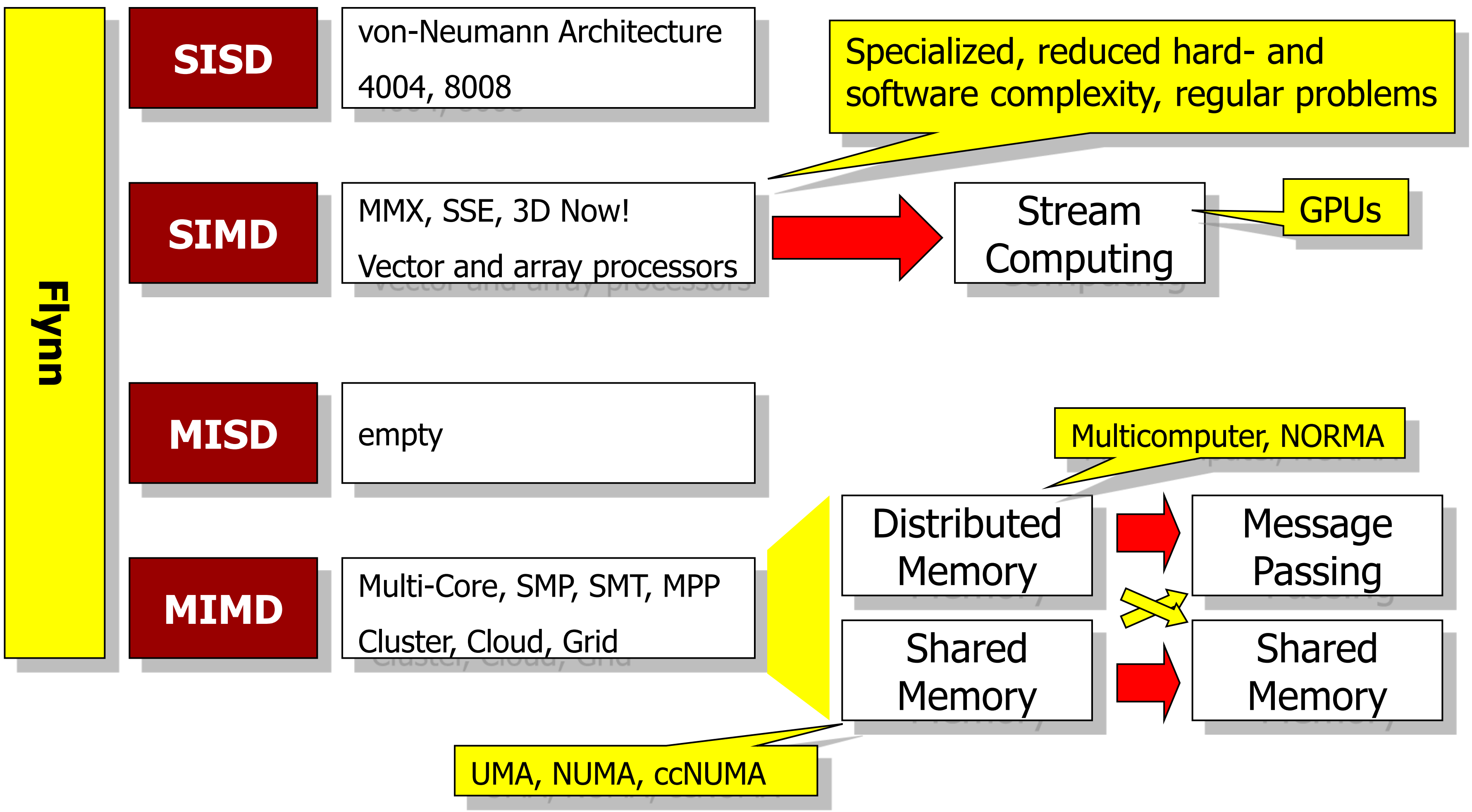
Computing model (2)

- Multicomputer
  - Multiple nodes
  - Interconnection network
- Many parallel instruction streams
- Local (and remote) memory accesses
- According to Flynn?



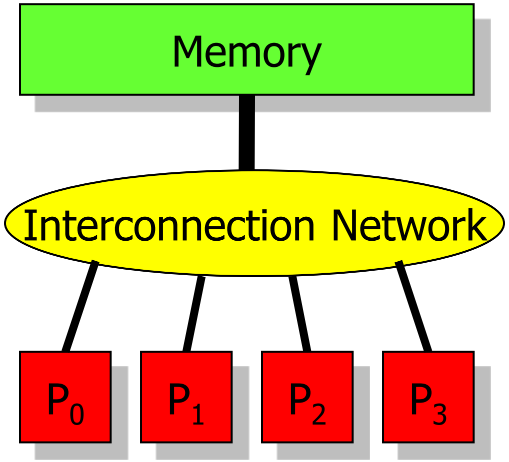
Parallel programming: Locality (III)

Classification by Flynn

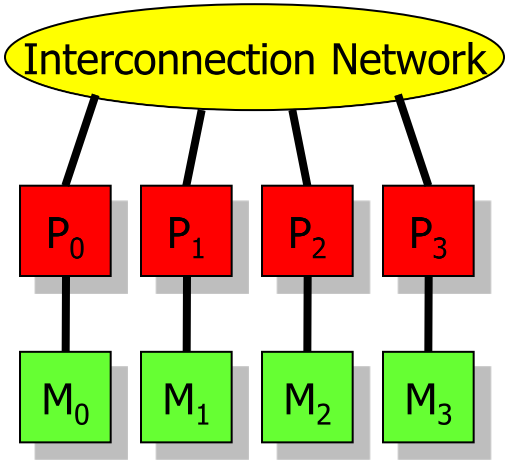


Distributed and Shared Memory

- **Shared Memory**
  - Shared use of one copy
  - Scalability issues
  - Atomicity, locking, synchronization



- **Distributed Memory**
  - Explicit data exchange
  - Only access to local memory
  - Data distribution and communication scheme



Concepts

1. Concurrency
   - Functional Decomposition, Domain Decomposition, Pipeline Decomposition
   - Re-engineering for parallelism
     - Control dependencies, data dependencies
2. Parallel programming paradigms
   - Shared memory: PThreads, OpenMP
   - Distributed memory: Message-passing
   - Data parallel operations (SIMT): CUDA, OpenCL
3. Supporting structures
   - SPMD, loops, master/worker, fork/join, data structures

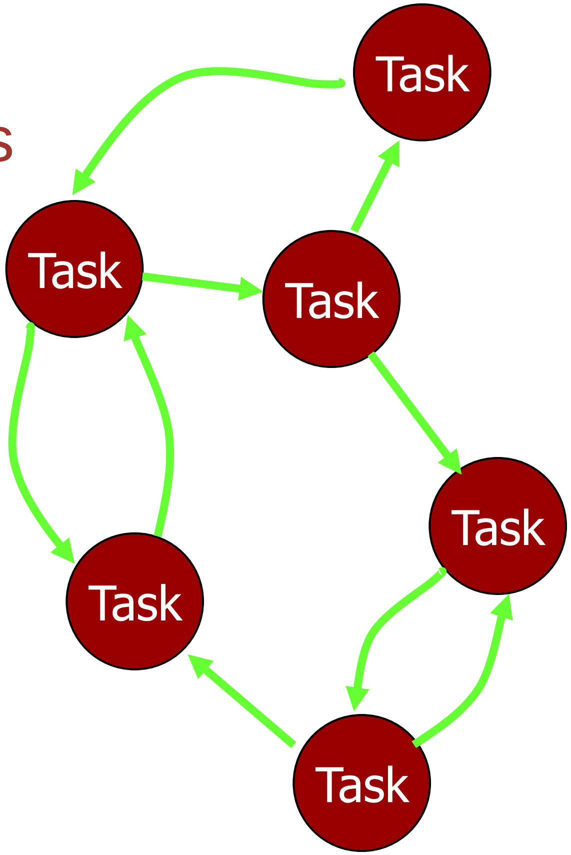
Programming Model

- von-Neumann:
  - 1 node → **1 instruction stream**
- Multicomputer:
  - n nodes → **n instruction streams**
  - Too complex
- Modular approach
  - Simple components made of abstract elements
  - Data structures, loops, procedures
- Single Program Multiple Data (SPMD)

Parallel Programming: Modularity (IV)

Programming Model

- Used for this lecture: tasks & Channels
  - Task
    - Computations
    - Instructions & Memory
  - Channel
    - Communication among tasks
    - Message-based
    - Blocking receives
- Computation & Communication
- Data dependencies



Other Programming Models

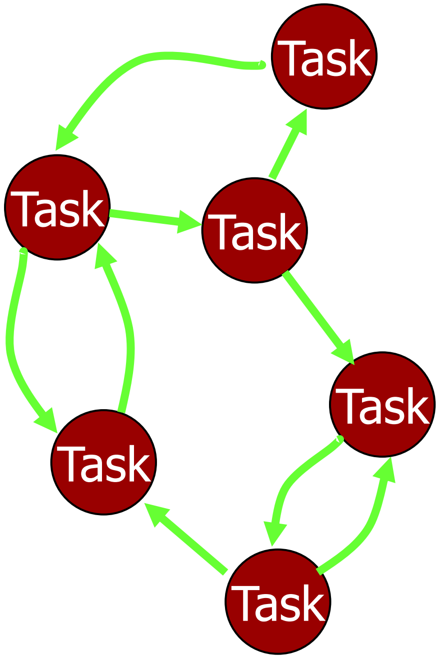
- Message Passing
  - Difference:
    - Message Passing: send to x
    - Task/Channel: send over channel y
  - SPMD
- Data Parallelism
  - Applying one operation to multiple elements of a data structure
  - SPMD
- Shared Memory
  - Uniform memory access from user‘s point of view
    - No explicit communication
    - Locks & Semaphores
  - SPMD

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

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.

- Communication & synchronization
  - Explicit / implicit
- SIMD: one instruction stream, no synchronization necessary
- MIMD: synchronisation necessary
  - Shared variables
  - Process synchronization
  - Blocking message exchange



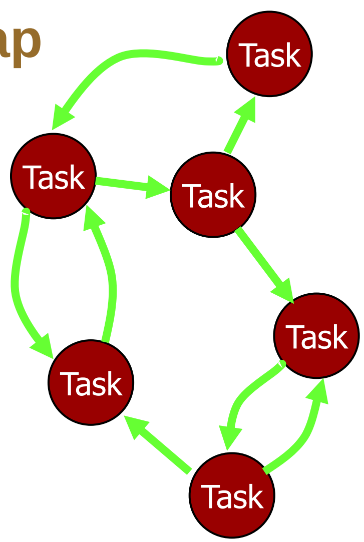
RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

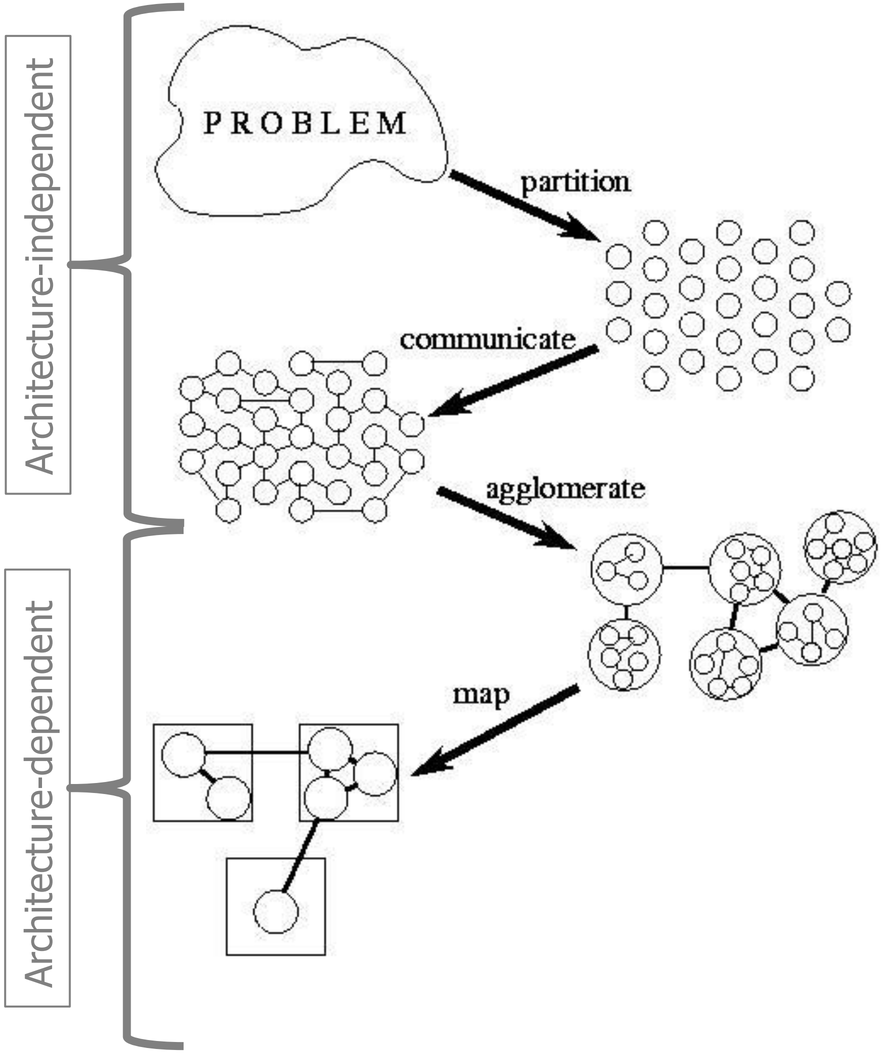
Algorithm Design

Algorithm Design

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

- Foster's PCAM
  - **Partition**
  - **Communicate**
  - **Agglomerate**
  - **Map**





Algorithm Design

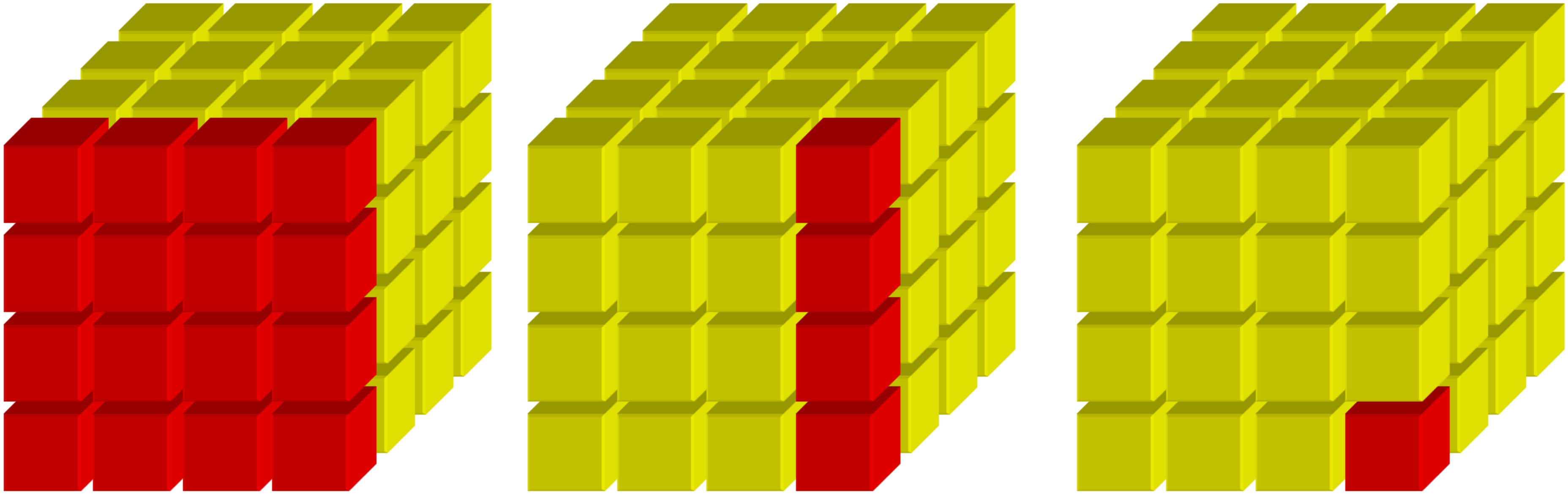
Number of Tasks >> Number of P

- **P**CAM: Partitioning
  - Ignore technical aspects like number of processing units
  - Maximal granularity
  - Partition computation and data
    - Domain Decomposition
    - Functional Decomposition
    - Pipeline Decomposition
  - Avoid replication, disjoint partitioning
    - See also minimization of communication

→ **Partitioning**

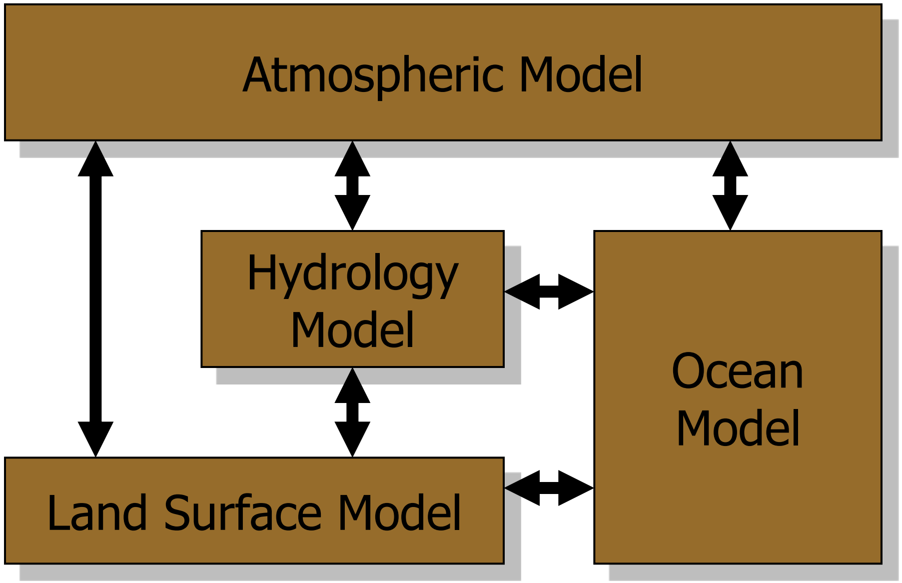
Algorithm Design

- **P**CAM: Partitioning
  - Example 1: Domain Decomposition
  - Typical uses: data parallelism, e.g. arrays & trees



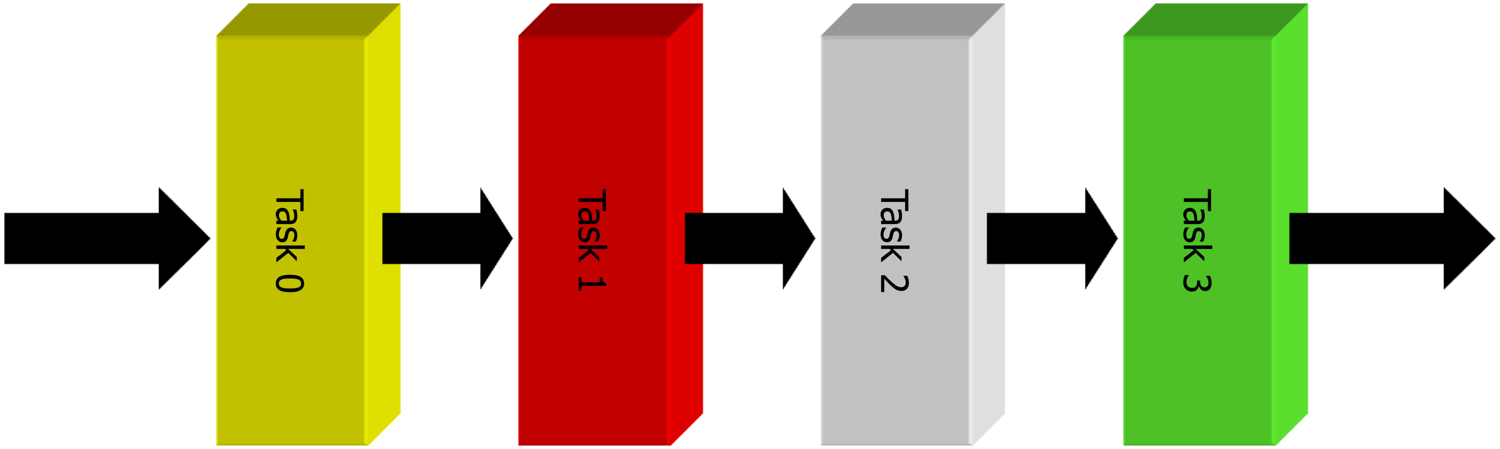
- PCAM: Partitioning
  - Example 2: Functional Decomposition
  - Typical uses:
    - Function calls
    - Different loop iterations
  - Rather too many tasks than too few!

Climate Computing Model



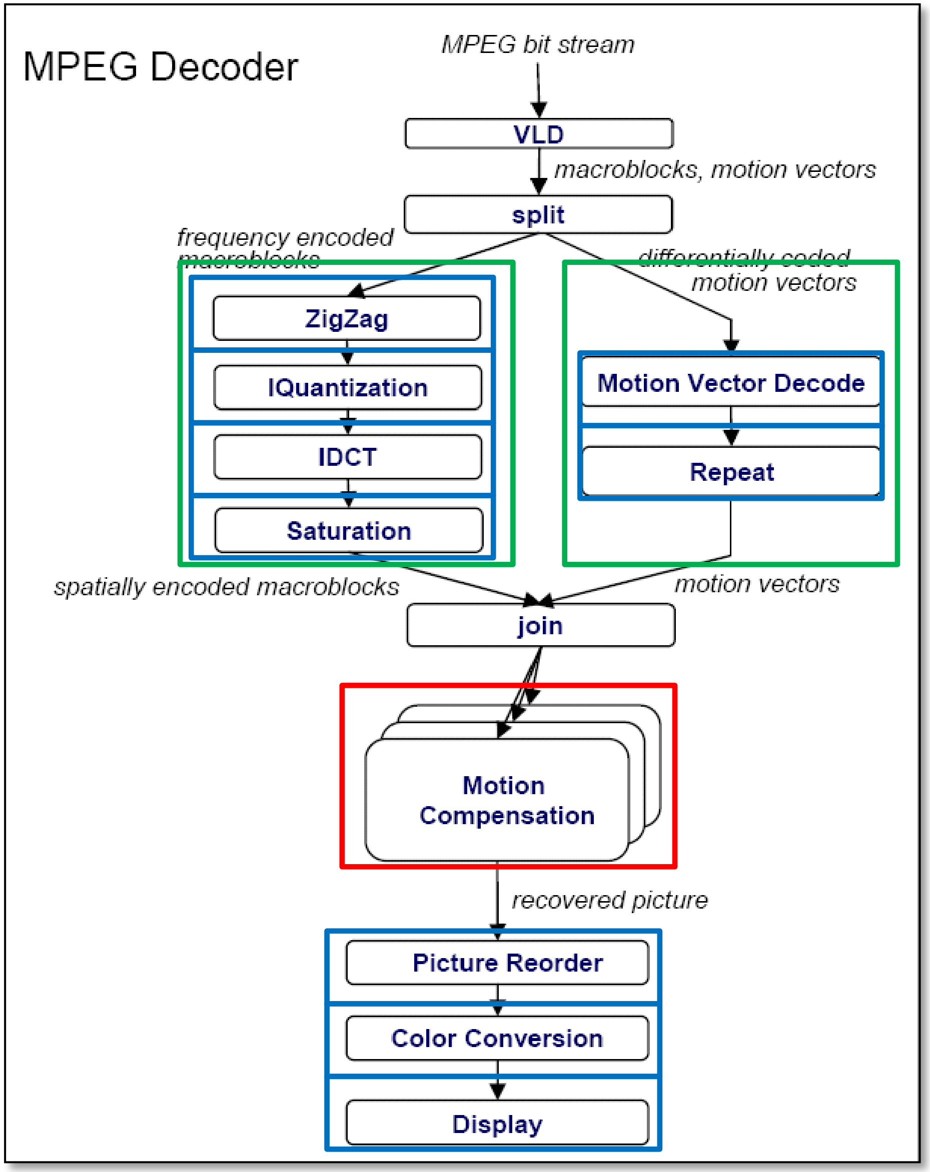
Algorithm Design

- **P**CAM: Partitioning
  - Example 3: Pipeline Decomposition
  - Data flow through several pipeline stages
  - Typical uses:
    - Instruction pipelining in modern CPUs



Algorithm Design

- Identify possible decomposition techniques!
- Domain Decomposition
  - Red
- Functional Decomposition
  - Green
- Pipeline Decomposition
  - Blue



Algorithm Design

- P<u>**C**</u>AM: 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

▪ PCAM: Communicate

- 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
  - GS execution: diagonal wave front or Red/Black method

- PCAM: Communicate
  - 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)**

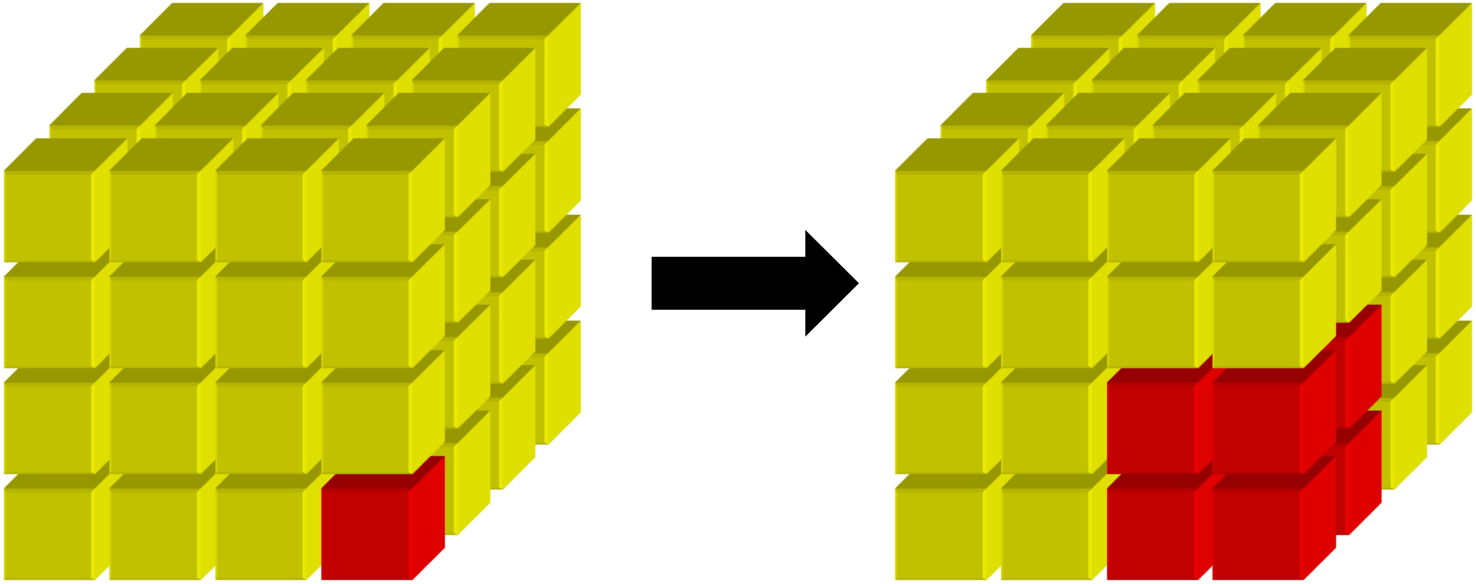
$$S_i = X_i + S_{i-1}$$

- Divide & Conquer to exploit parallelism
  - Tree structures, as long as partitions can be computed independently
  - Associativity of addition, **O(log N)**

- PCAM: Agglomeration
  - From the abstract to the concrete
  - Fixing the parallel computer model
- Goal
  - Increase granularity (coarse-grain)
  - Maintaining flexibility, therefore reducing development costs
- Number of tasks T >= number of processors P
  - Depending on use case:
    - One order of magnitude more Ts than Ps (parallel slackness)
    - T == P (HPC)
  - SIMD: T = 1
  - If T = P, then mapping (almost) done

PCAM: Agglomeration

- Combining of tasks
  - Increase of granularity
- Motivation: Reducing communication costs
  - Fixed & variable fraction (surface-to-volume effects)

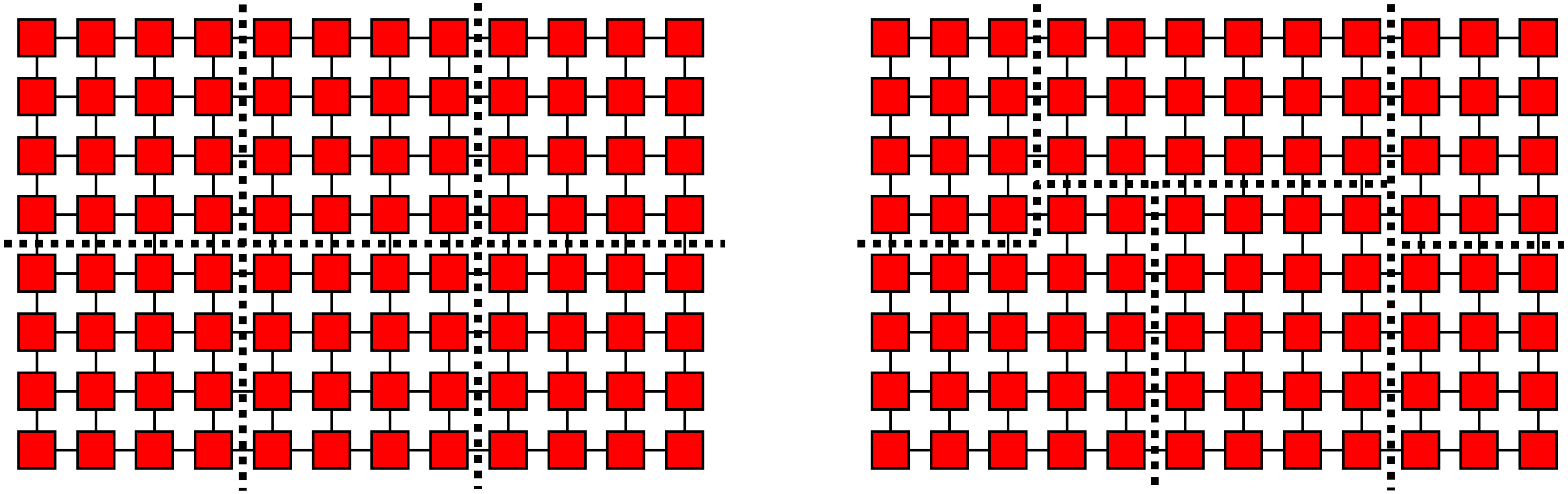


- PC**A**M: Agglomeration
  - Replication of data and computation
- Example: global sum
  - Chained: *2(N-1)* steps (sum & broadcast)
    - ➔ Redundant computation in a ring, no broadcast *((N-1))*
  - Tree-based: *2 log N* steps (sum & broadcast)
    - ➔ Redundant computation in a butterfly, no broadcast *(log N)*

➔Reducing Communication

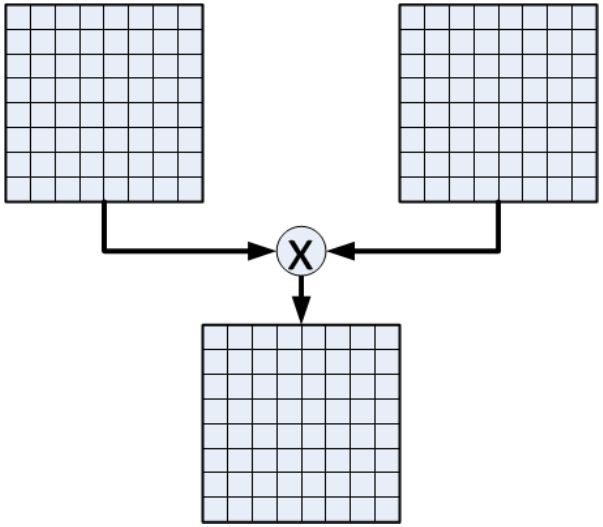
- PCA**M**: Mapping
  - Assignment: task ←→ processor & memory
    - Place tasks that can execute concurrently on different processors
    - Place tasks that communicate frequently on the same processor
    - Note that this implies conflicts
  - Mapping not necessary for:
    - Uni-processors or shared memory systems with automatic mapping
    - Hardware mechanism or the OS responsible for scheduling
- Mapping problem is NP-complete
- Dynamic Load Balancing

- PCA**M**: Mapping
  1. Concurrent tasks on different Ps
  2. Frequently communicating tasks on same P

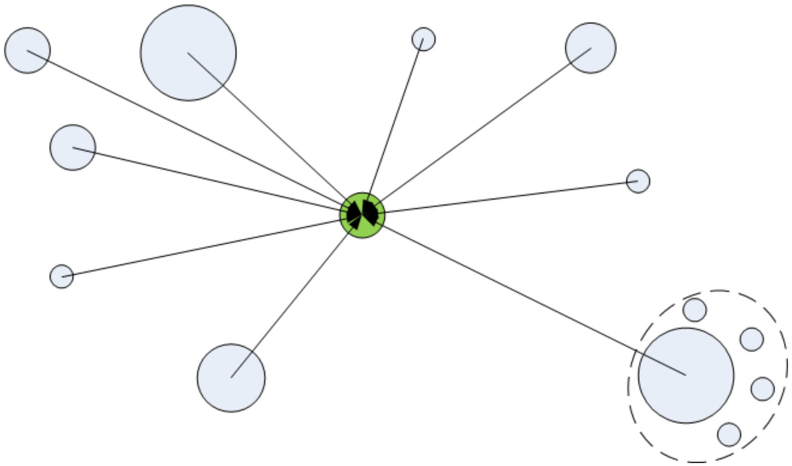


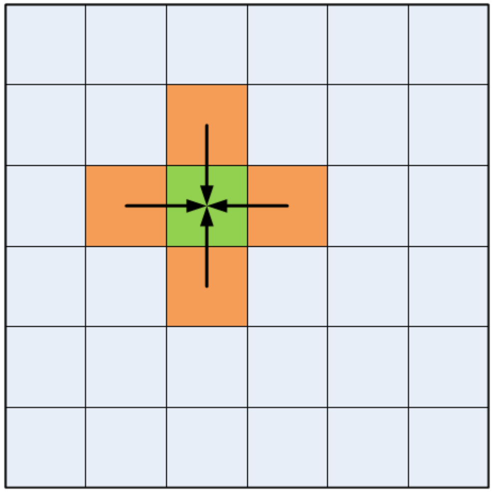
Ready to run!

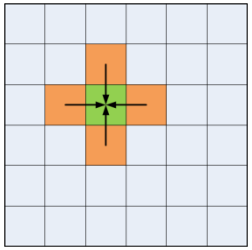
Parallel Computing
Algorithm Design - Examples



- Matrix multiply
- Stencil operation
- N-Body problem



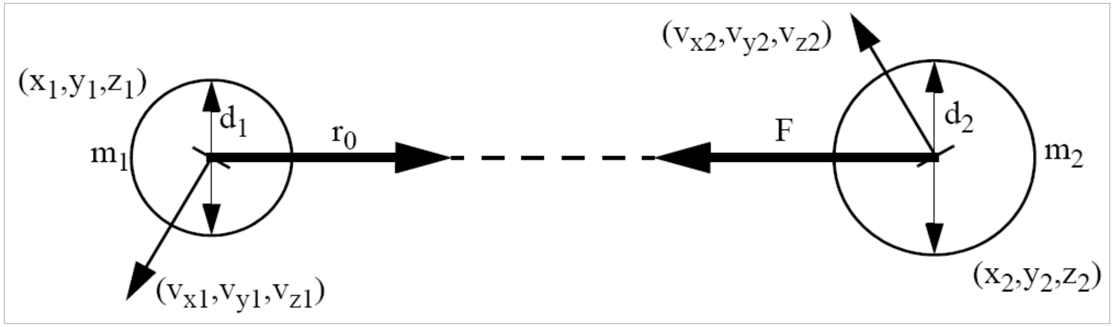


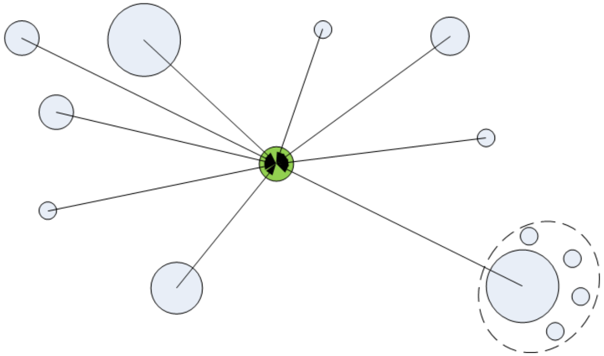


- Stencil codes (e.g. Jacobi method)
  - Approximation by time steps

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

- N-Body codes
  - Gravitational forces, electrostatical forces
  - Smoothed particle hydrodynamics (simulating fluid flows)
  - Superposition
  - Approximation by time steps





Parallel Computing
Summary

- Concurrency and parallelism of fundamental importance
  - Granularity
  - ILP, TLP, DLP
- Characteristics of "good" parallel programs
  - Concurrency, Scalability, Locality and Modularity
- Algorithm design
  - Partition, Communicate, Agglomerate, Map
- Parallel computing highly dependent on architecture!
  - (Flynn's classification,) shared & distributed memory
- Literature
  - Foster Online
    - http://www.mcs.anl.gov/~itf/dbpp
  - Introduction to Parallel Computing
    - http://www-users.cs.umn.edu/~karypis/parbook