ACM/CS 114
Parallel algorithms for scientific applications

Michael A. G. Aïvázis

California Institute of Technology

Winter 2012

Motivations for going parallel

- why bother?
  - *speed*: there are fundamental limits to the processing power of a single processor
  - *throughput*: time to solution is critical for many problems
  - *size*: high resolution requires lots of memory
  - *availability*: the tool exists, use it
- but be careful
  - the commercial market is unstable
  - the computing environment is somewhat primitive
  - software packages and libraries are emerging slowly
  - parallel programming is not hard, but it requires *discipline*

Taxonomy

- an early classification of computer systems focused on the relation between *instruction streams* and *data streams*:
  - SISD: single instruction, single data
  - SIMD: single instruction, multiple data
  - MIMD: multiple instruction, multiple data
- SISD describes conventional serial computers
  - the programming model only: the hardware has moved on...
- SIMD and MIMD are the traditional models for parallel machines
- MIMD systems are often programmed in SPMD mode: single *program*, multiple data
  - when the parallel environment provides a *naming scheme* for the instruction streams, such as processor id, or task name

Architectural issues

- *control:* SIMD vs. MIMD
- *coördination*: synchronous vs. asynchronous
- *memory organization*: private vs. shared
- *address space*: local vs. global
- *memory access*: uniform vs. non-uniform
- *granularity*: the power of each processor
- *scalability*: dependence on the number of processors
- *interconnect*: topology, routing, switching

Tradeoffs

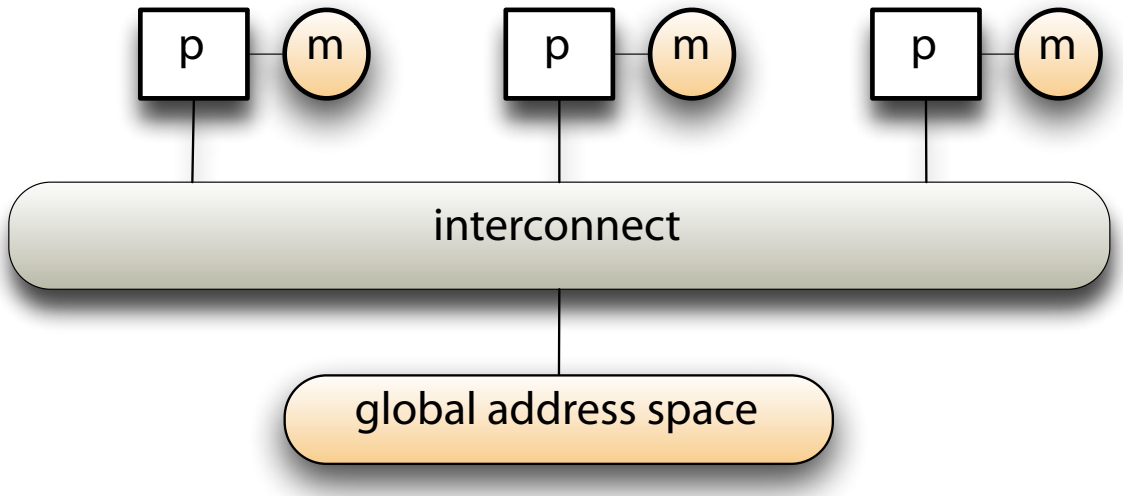
	shared memory	distributed memory
scalability	*harder*	*easier*
programmability	*easier*	*harder*

- shared memory permits parallelizing serial program gradually, focusing on worst bottlenecks first
- distributed memory requires partitioning and distributing both data and work across processors, which usually rules out incremental parallelization

Categories of parallel architectures

- vector or array processor
- SMP: symmetric multiprocessor
- MPP: massively parallel multiprocessor
- DSM: distributed shared memory
- clusters
- hybrids
  - SMP or MPP with vector processors
  - networked clusters of SMPs
  - SMP+GPGPU

Generic parallel architecture



- a trivial but powerful observation: access to memory implies access to information
- hence, it becomes a determining factor for both hardware and algorithm design

Memory hierarchy

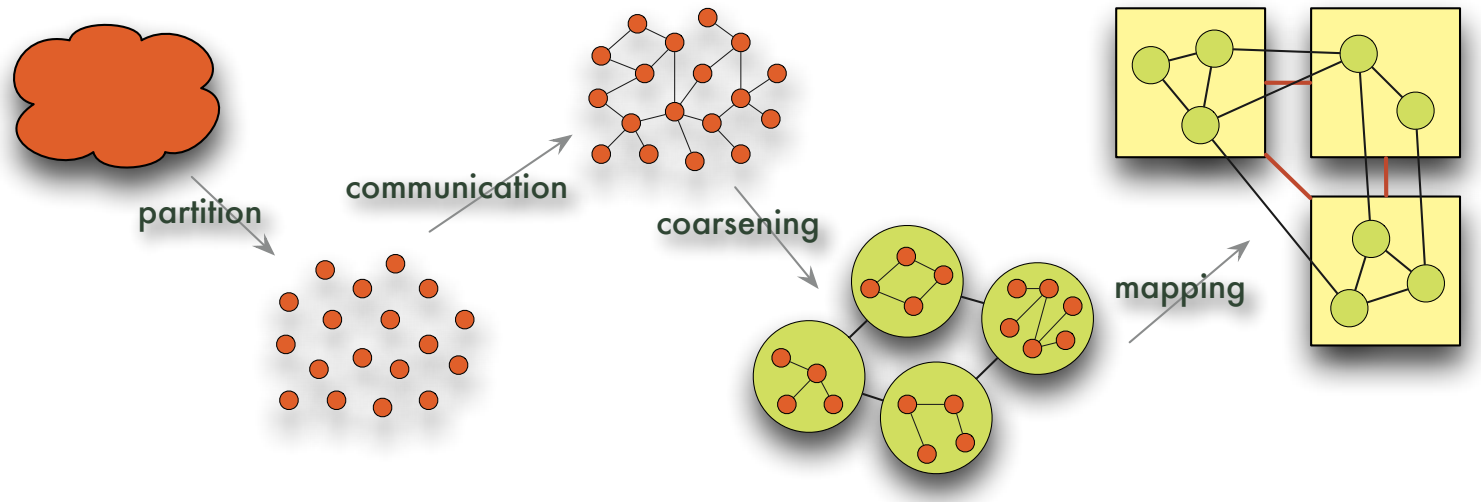
- high performance architectures have a multi-tier memory hierarchy
  - registers
  - on-chip caches, usually referred to as level 1
  - off-chip caches (level 2)
  - random access memory
  - remote memory (off processor)
  - virtual memory, known as paging memory, that usually involves secondary storage
  - secondary storage (disks)
  - tertiary storage (tapes)
- these have latencies and bandwidths that vary by orders of magnitude
- *cache misses* are the most frequently cited reason why real codes only achieve a small fraction of the benchmarked performance of a CPU
  - really small: 10% of peak is considered a success!

Parallel programming paradigms

- *functional languages*: specify what to compute, not how
- *parallelizing compilers*: automatic or semi-automatic detection of parallelism in serial code; mostly loop-unrolling, often with the help of special mark up (pragmas)
- *object oriented*: parallelism encapsulated within distributed objects
- *data parallel*: simultaneous operations on memory; mostly arrays
- *shared memory*: multiple threads executing a pool of tasks using common memory
- *remote memory access*: one sided put/get communication between processes
- *message passing*: two sided, coördinated send/receive communication between processes

Designing parallel algorithms

- *identification*: identify the part of the problem that can be parallelized
- *partition*: decompose the parallelizable part into fine-grained tasks
- *communication*: determine the necessary communication patterns among tasks
- *coarsening*: combine into coarser tasks and adjust the communication patterns
- *task mapping*: assign tasks to processors



Paradigms for parallel algorithms

- *embarrassingly parallel*: mostly independent tasks
- *functional decomposition*: based on computational task (activity)
- *data parallel*: aka *loop-level* parallelism: array operations
- *domain decomposition*: based on the distribution of data
- *divide-and-conquer*: tree-like partitioning
- *pipelining*: multiple overlapping stages

Communication issues

- latency and bandwidth
- routing and switching – not much of an issue any more
- contention, flow control and aggregate bandwidth
- collective communication
  - one to many: broadcast, scatter
  - many to one: gather, reduction, scan
  - all to all
  - synchronization barrier
- assigning work to processors
  - partitioning
  - granularity
  - mapping
  - scheduling
  - load balancing

Factors determining performance

- *concurrency*: maximize the work that can be done in parallel
- *load balance*: make sure the load is (and stays) divided evenly
- *parallel overhead*: work not present in the equivalent serial computation
  - process startup and shutdown costs
  - communication
  - synchronization
  - redundancy
  - speculative work

Computational models

- abstractions for architecture, algorithm analysis, and performance modeling
  - PRAM: Parallel random access machine
  - LogP: latency, overhead, gap, processors
  - BSP: bulk synchronous parallel
  - CSP: communicating sequential processes
  - ... and many others
- all occasionally useful tools for reasoning about implementation strategies for real programs, since they steer you away from common but perhaps non-obvious mistakes
- unfortunately, none is a substitute for understanding the characteristics of your target platform