Parallel Programming

The Parallel Programming Landscape
Week 10

Administrative

- Thursday 31/3 is
  - Assignment II deadline
  - in-lab viva for both assignments
  - presence is mandatory

“If you build it, they will come.”

- “And so we built them. Multiprocessor workstations, massively parallel supercomputers, a cluster in every department … and they haven’t come.
- Programmers haven’t come to program these wonderful machines.
- The computer industry is ready to flood the market with hardware that will only run at full speed with parallel programs. But who will write these programs?”

Mattson, Sanders, Massingill (2005)

- parallelism requires decompositionality
- parallelism is motivated by scalability
  - time
  - space

Challenges

- unusual angle of attack
- dependencies are complex
- managing overheads
- managing utilisation
- sharing data

Finding concurrency

- task decomposition
- data decomposition
- grouping and ordering
- sharing patterns
- architectural constraints
- [design evaluation and iteration]

Typical algorithm structure

- task oriented
  - task parallelism // divide and conquer
- data oriented
  - geometric decomposition // recursive
- dataflow (data+task) oriented
  - pipelines // events

Programming model

- task mapping
- data mapping
- communication mapping
- synchronisation

Some popular models

Message Passing Interface (MPI)

- used for programming grids, clusters, supercomputers
- "message-passing" (vs. "shared-memory")
- aimed at HPC market
- task and data mapping are explicit
- communication mapping & synchronisation are implicit
- it is basically a communication protocol
- most commonly, one thread per processor

MapReduce (Google)

- processing huge datasets on clusters
- map step: partition input, pass it to node
- reduce step: aggregate results from nodes
- task id explicit but task map implicit
- data distribution and communication mapping implicit
- synchronisation explicit (bulk)
- implementations: Apache Hadoop

Pthreads

- standard Unix/POSIX thread model
- aimed at general-purpose (OS) programming
- explicit task mapping
- implicit data distribution & communication mapping
- explicit synchronisation

Dataflow

- fully automatic/implicit everything
- suitable for multicores, GPUs, grids
- dataflow = changing a variable forces recalculation of
- other variables (like in a spreadsheet!)
- some of the variables are the inputs
- declarative style
- very simple to program

CUDA/OpenCL

- graphics, GPGU programming
- Task Mapping ?
- Data Distribution ?
- Communication Mapping ?
- Synchronization ?

Functional programming

- no side-effects, no interference, schedule-independent
  - f M N P ...
  - map f l
  - reduce f l k

A zoo of models/languages

- HPC: MPI, OpenMP, DataCutter, HPF
- DSP: MathWorks, YAPI, StreamIt
- GP: Pthreads, CUDA, Intel Thread BB, Berkely UPC, Java/Titanium
- Languages: Sequoia (memory-oriented),
- Fortress (functional), Erlang (message passing functional), etc.

“Dwarves”

The Landscape of Parallel Computing Research:
A View From Berkeley
http://view.eecs.berkeley.edu

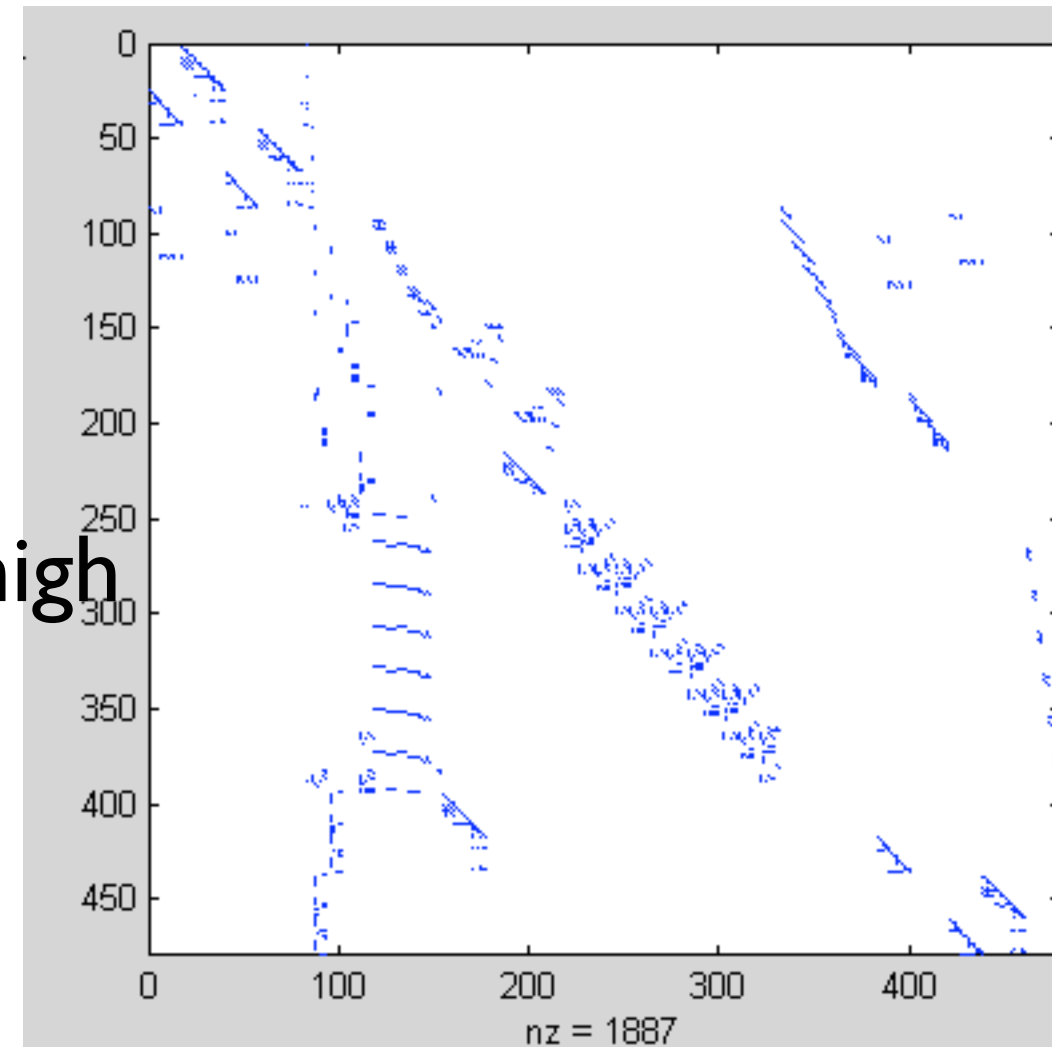
- “A *dwarf* is an algorithmic method that captures a pattern of computation and communication.”

1. Dense linear algebra

- classic vector and matrix operations
- natural match for vector-like machines
- data distribution and load balancing key
- scale very well
- MATLAB, BLAS

2. Sparse linear algebra

- matrices with large numbers of 0s
- dictionary-like data structure (index, value)
- much more complex programming, high number of integer ops for indices
- extremely intricate dependence mappings
- specialised on actual structure of data
- SpMV, OSKI, SuperLU



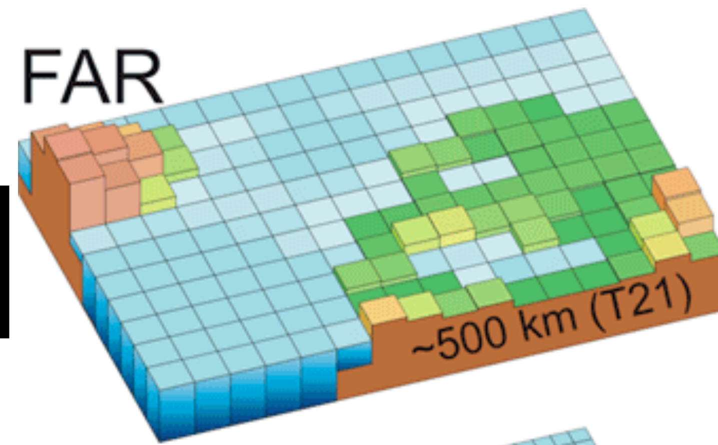
3. Spectral Methods

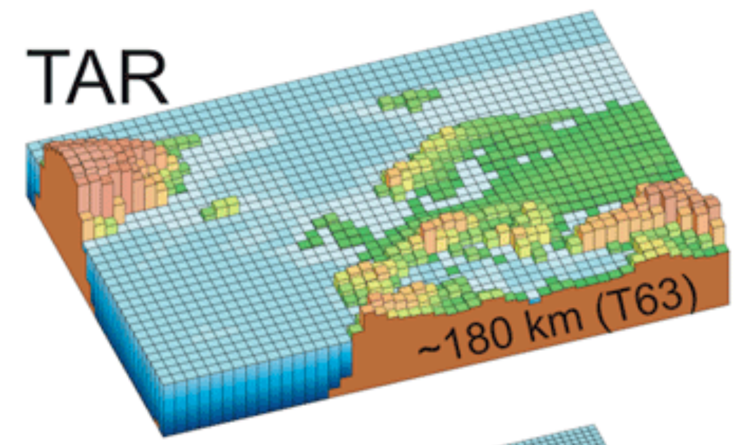
- operate on data in spectral domain
- FFT, wavelet transforms, etc
- butterfly-like operations
- local operations, global reorderings req'd
- highly vectorizable

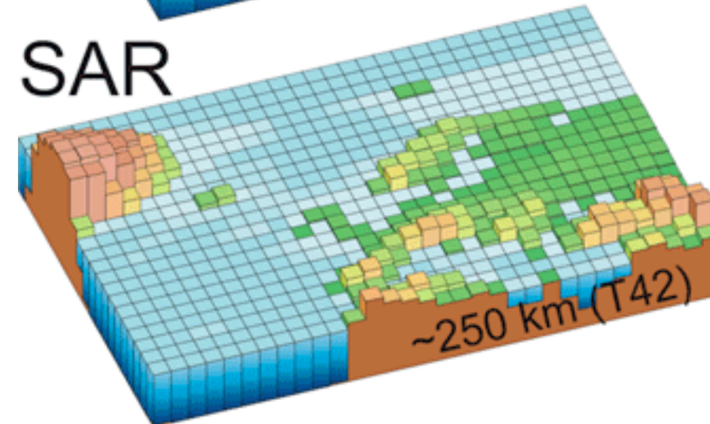
4. Nbody problems

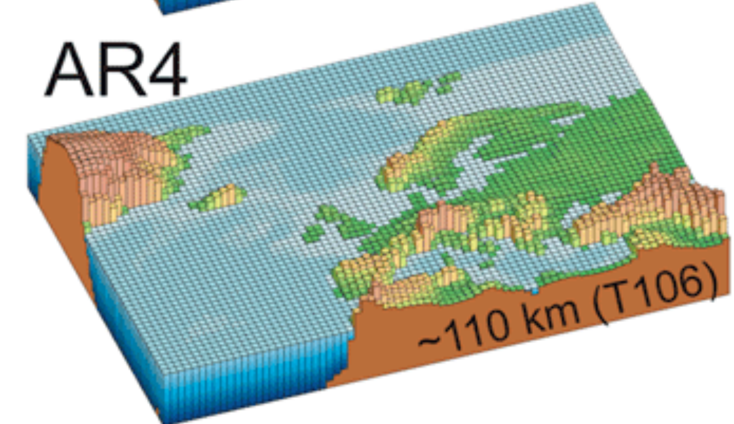
- interactions between many discrete points
- molecullar dynamics, swarms, astronomy, etc
- floating computation >> memory bandwidth (simple)
- better algorithms structure points in hierarchy (far to close, tree-like, etc.)
- (Barnes-Hut, Fast Multipole, etc)
- dynamic load balance tricky

5. Structured grids



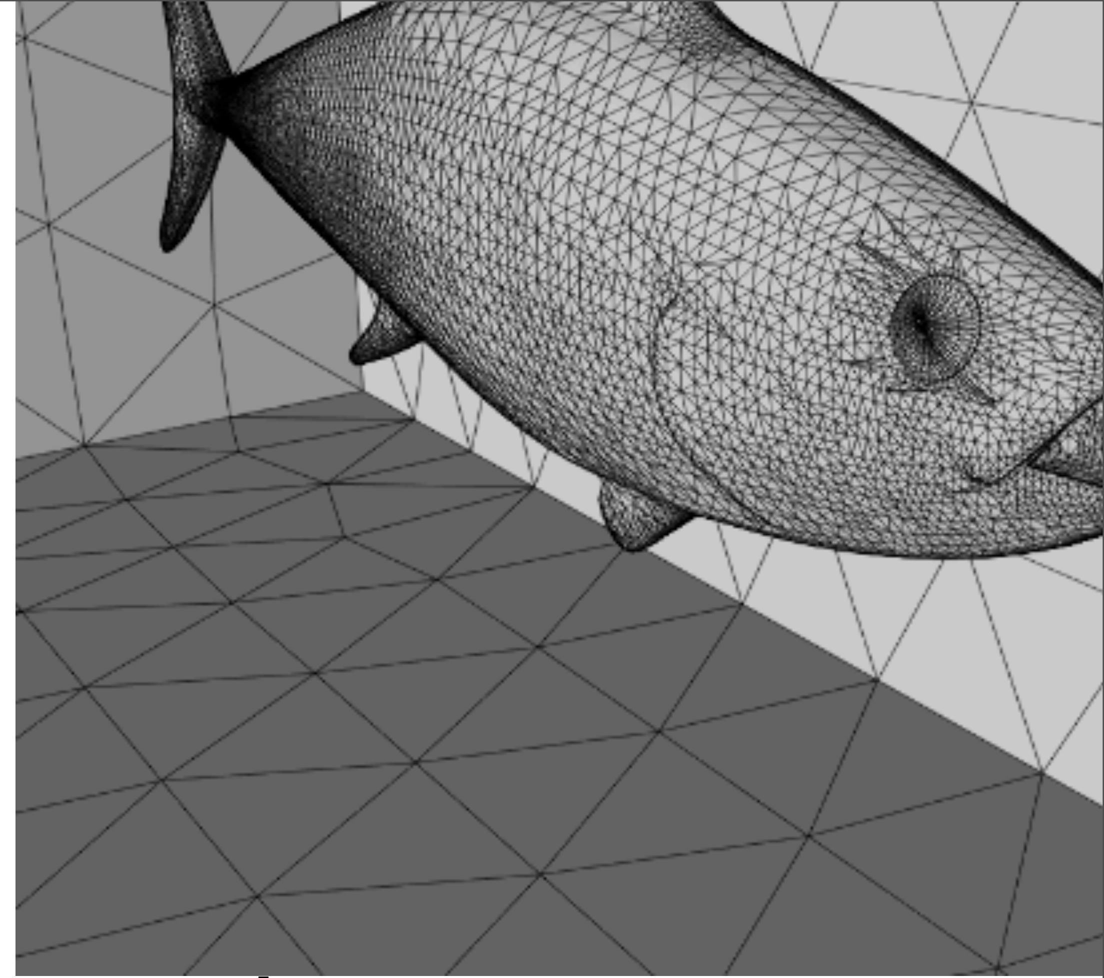






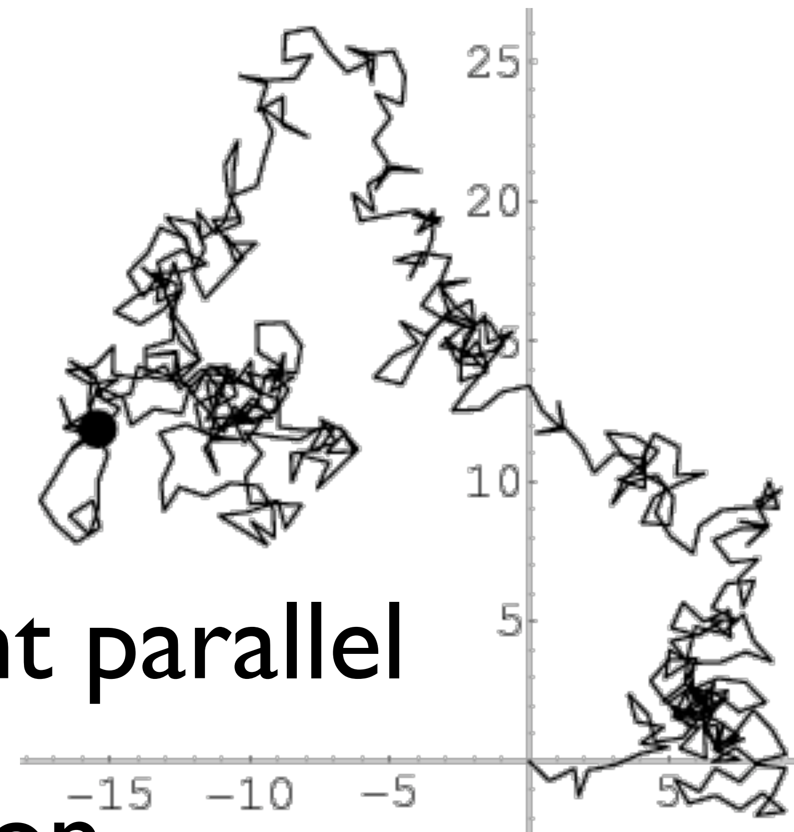
- data is arranged in regular grid
- computation is only local (neighbour-based)
- highly vectorizable, locality helps
- parallel mapping is easy (and static)
- Conway game of life, climate modelling, some finite elements analysis, etc.

6. Unstructured grids



- updates on irregular mesh/grid
- from arbitrary mesh to varied granularity
- element update from neighbours
- massively data parallel but complex data distribution
- similar to sparse linear algebra
- most finite-element analysis

7. MapReduce/Monte Carlo



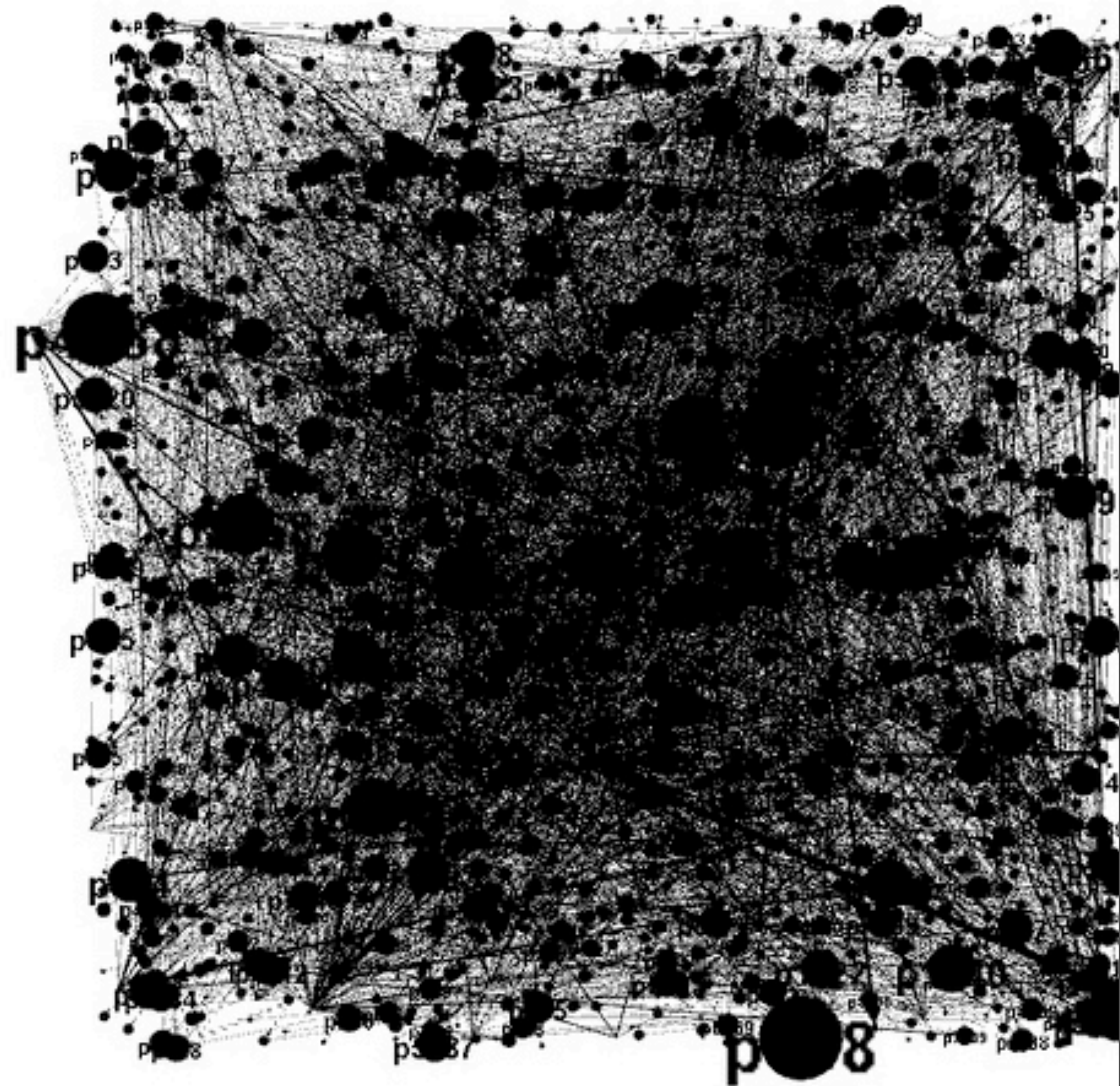
- "embarrassingly parallel"
- repeated random trials (MC)
- MR is more general: independent parallel
- execution followed by aggregation
- nearly no communication required
- very useful for approximation (e.g. pi)

8. Combinational logic

- simple operations on large amounts of data
- exploit bit-level parallelism for high throughput
- e.g. computing checksums or CRCs
- not very well supported by processors
- needs special hardware primitives

9. Graph traversal

- traverse graph,
- examine nodes for property
- very difficult to make paralle
- but not impossible (depend on memory)
- e.g. searching in a graph



10. Backtrack Branch & Bound

- search and optimisation problems in very large space
- e.g. integer linear programming,
- boolean SAT, combinatorial optimisation, TSP
- divide search space in regions to be explored independently
- irregular branch structure means dynamic load balance is a challenge

11. Finite state machines

- computation as a FSM
- sometimes can be decomposed in smaller FSMs
- parallelism often difficult to achieve
- large FSMs require memory access

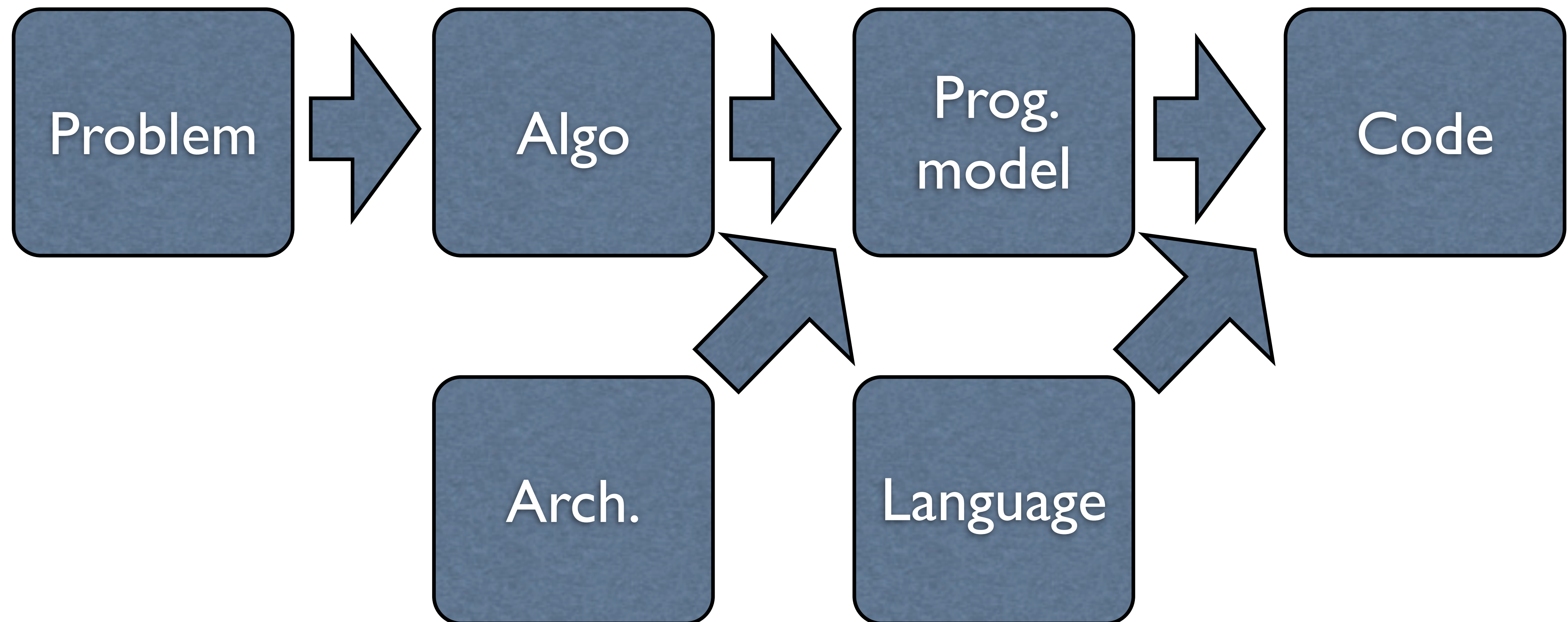
12. Other

- dynamic programming
- graphical models
- sort / databases
- computer graphics / games
- etc.

Common limits to performance

1. Dense matrix: computation
2. Sparse matrix: computation and memory
3. Spectral: memory latency
4. Nbody: computation
5. Structured grid: memory bandwidth
6. Unstructured grid: memory latency
7. Map/Reduce: none
8. Combinational: computation
9. Graph traversal: memory latency
10. FSM: everything

The challenge



Conclusions

- think 1000s of cores/chip
- rethink/abandon common benchmarks
- consider autotuners
- abstract from number of cores
- ... but consider the overall architecture
- focus on: data-level parallelism // independent task parallelism // instruction-level parallelism