703308 VO High-Performance Computing
The 13 Dwarfs of HPC

Philipp Gschwandtner

Overview

- The 13 Dwarfs of HPC
  - abstract application categories

Motivation

- MPI API or concepts such as data vs. task parallelism are still pretty low-level characteristics of parallel programs
- we need to be able to recognize higher-level classes of HPC applications and discuss them
- this lecture presents the most prominent classes of HPC applications
  - many new applications you encounter will fit into these categories or are a combination of them

How and why are dwarfs defined?

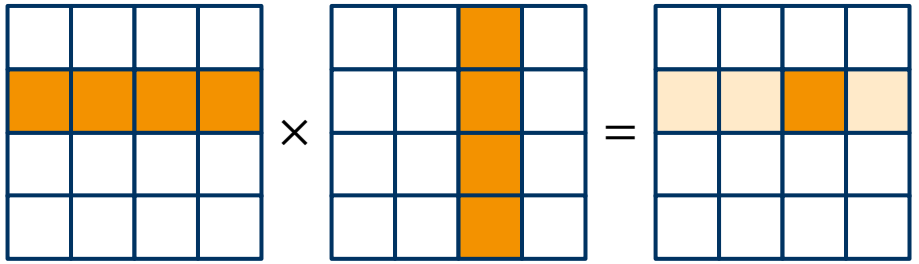
- group applications by similarity in computation and data structures
  - first published by Asanovic et al in *The Landscape of Parallel Computing Research: A View from Berkeley*
  - https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf
- purely algorithmic, implementation-independent
  - enables cross-platform reasoning and cross-application knowledge/resource sharing (e.g. libraries)
- serve as small, abstract, high-level benchmarks for studying new
  - programming models
  - communication patterns
  - hardware architectures, topologies
  - …
- used to kick off innovation in all of these aspects

7 original dwarfs of HPC

- What are we going do discuss?
  - 1. Dense Linear Algebra
  - 2. Sparse Linear Algebra
  - 3. Spectral Methods
  - 4. N-body Methods
  - 5. Structured Grids
  - 6. Unstructured Grids
  - 7. Monte Carlo Methods
- What have you already heard?
  - matrix mul (first MPI lecture)
  - N-body
  - heat stencil
  - Monte Carlo π

Dense linear algebra

- data is stored in densely-populated matrices (or vectors)
  - data is stored uncompressed (“as is”)
  - data access via strides, often unit stride
- e.g. matrix multiplication, LU decomposition, Gauss-Seidel, …
- rarely done manually, there are a TON of libraries out there



Dense linear algebra: characteristics

- naïve implementations usually memory bound
  - remember: memory wall!
  - caches and prefetching helps
- simple but significant data structure
  - stride often enables/prevents vectorization (SIMD)
  - fastest-changing index affects cache efficiency (hence matrices are often transposed)
- still the default measure for performance in HPC
  - e.g. TOP500 uses HPL, a high performance LINPACK benchmark
  - but nowadays not the only one (e.g. HPCG)

```
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    double tmp = 0.0;
    for (int k = 0; k < N; ++k) {
      tmp += A[i][k] * B[k][j];
    }
    result[i][j] = tmp;
  }
}
```

```
vgatherqpd ymm0{k2}, [rax+ymm5*1]
vmulpd  ymm0, ymm0, YMMWORD PTR [rdx+rdi]
...
```

Side note: TOP500 list: Rmax vs. Rpeak

- Rmax: achieved by software
  - high performance linpack (HPL) benchmark
  - linear algebra stress-testing
- Rpeak: achievable by hardware
  - product of: number of FP units per CPU, their FP instructions per cycle, clock frequency, and number of CPUs

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107

Taken from the June 2024 list of the Top500

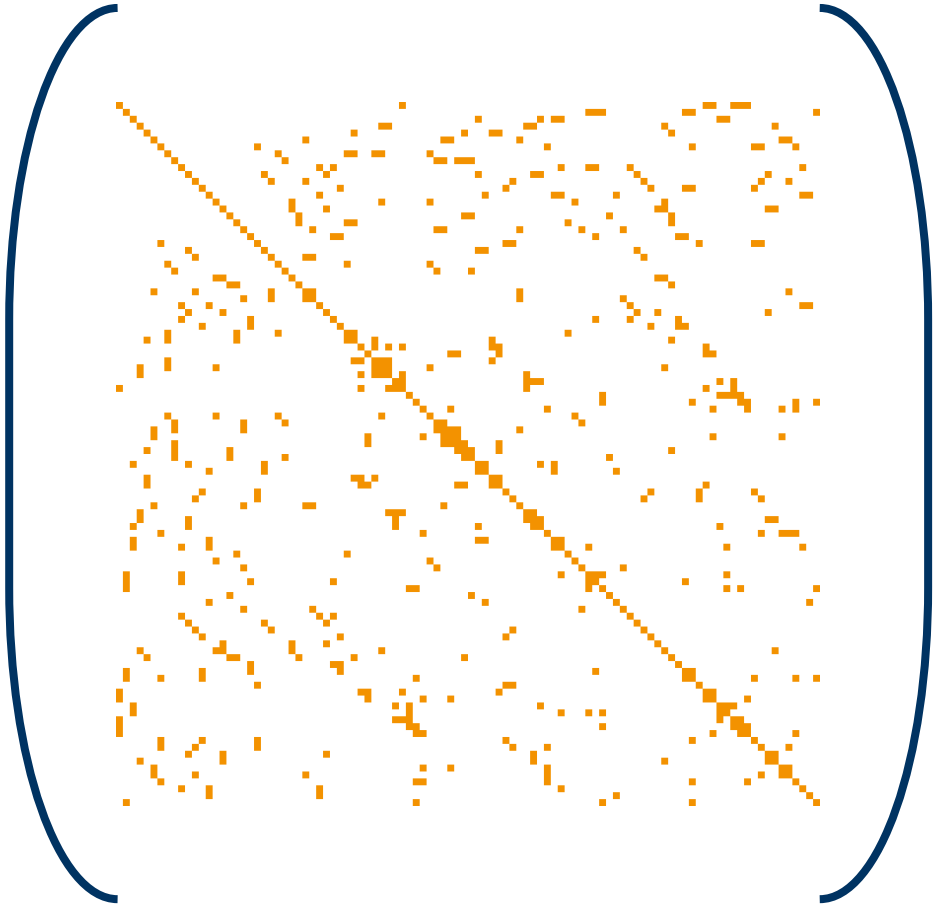
Dense linear algebra: optimizations

- loop blocking or tiling
  - do not work on single elements but smaller blocks (e.g. 2x2 or 32x32)
  - exploits locality and cache
  - also, lots of other HOTs (Higher Order Transformations)
- vectorization (SIMD, e.g. SSE/AVX)
  - might entail modifications, e.g. transposing matrix A or B in matrix mul
- hardware-specific instructions
  - e.g. fused multiply-add (FMA)

```
for (int ii = 0; ii < N; ii += ib) {
 for (int jj = 0; jj < N; jj += jb) {
  for (int k = 0; k < N; ++k) {
   for (int i = ii; i < ii+ib; ++i) {
    for (int j = jj; j < jj+jb; ++j) {
      // ... process single tile
    }
   }
  }
 }
}
```

Sparse linear algebra

- data is stored in sparsely-populated matrices (or vectors)
  - (vast) majority of data is zero
  - data is stored in compressed format
  - data often accessed indirectly via indices
- e.g. conjugate gradient, Google's PageRank, data mining



Sparse linear algebra: characteristics

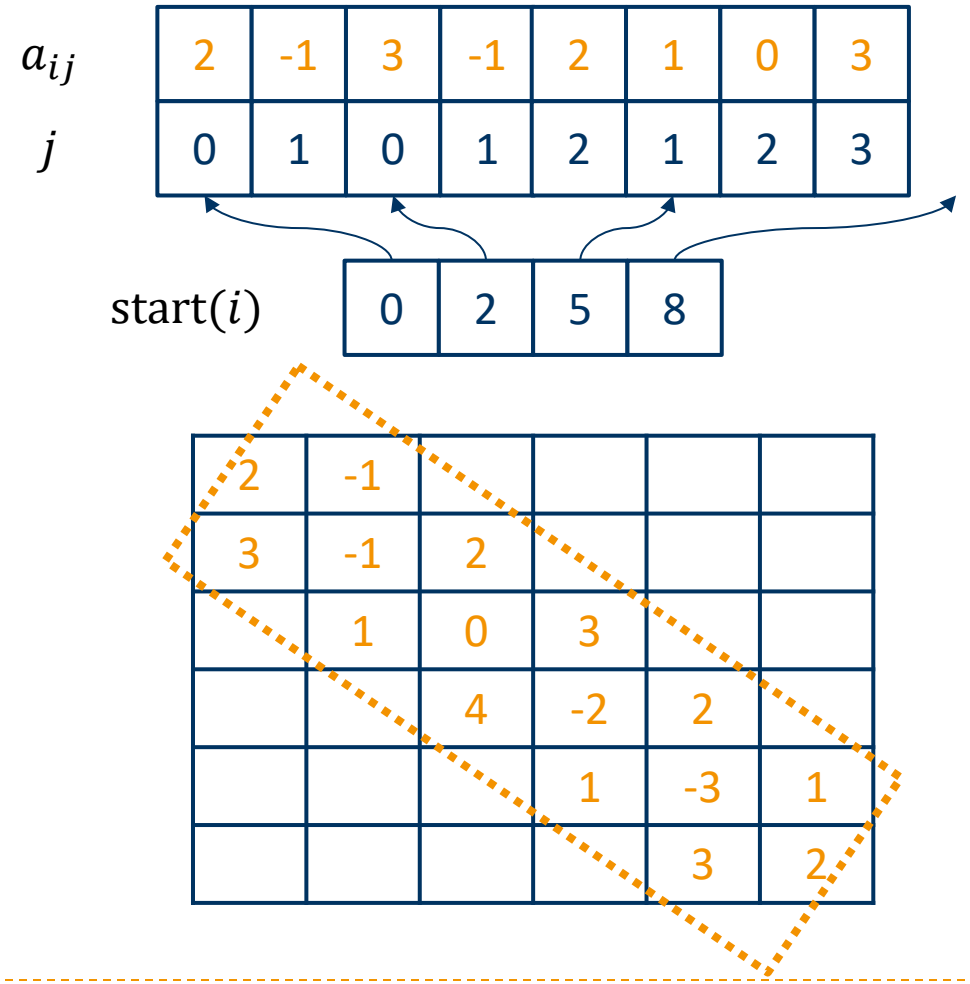
- computationally or memory limited
  - depends on sparsity of data, data structure representation and algorithm
- different data structures available
  - e.g. coordinate scheme (COO) or "triplet format" or similar: (i, j, a_{ij})
  - array of structs (AoS) vs. struct of arrays (SoA)
  - not necessarily sorted!

```
typedef struct sparseElement {
    int i; int j; double value;
} sparseElement;
sparseElement sparseMatrix[SIZE];
sparseMatrix[0].i = 0;

// ############## vs. ##############

typedef struct sparseMatrixT {
    int i[SIZE]; int j[SIZE];
    double values[SIZE];
} sparseMatrixT;
sparseMatrixT sparseMatrix;
sparseMatrix.i[0] = 0;
```

Sparse linear algebra: optimizations

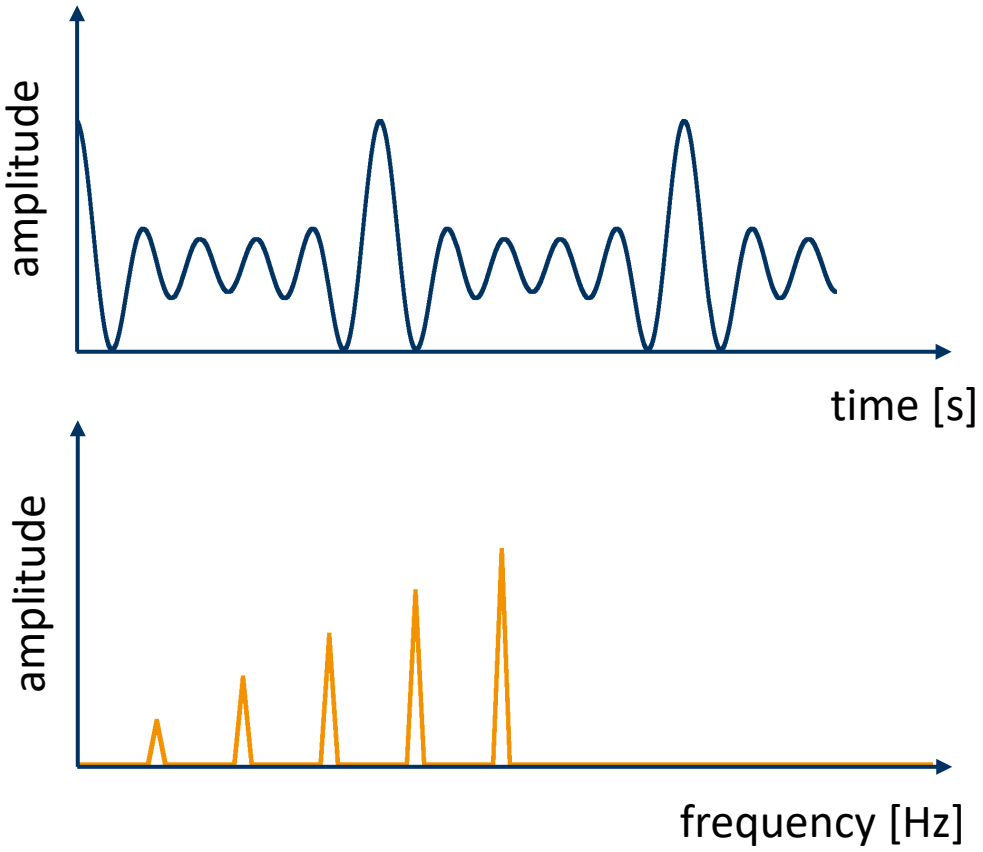
- compressed row storage (CRS)
  - two arrays of size N
  - one holds a_{ij}, the other j
  - third array points to start of row i in j
  - smaller memory footprint than COO
    - $2N + (m + 1)$ vs. $3N$

- variants
  - column-major variant (CCS)
  - store small blocks (2x2 or 4x4) instead of single elements, improves SIMDness
  - compressed diagonal storage (CDS)
    - use domain-specific knowledge!



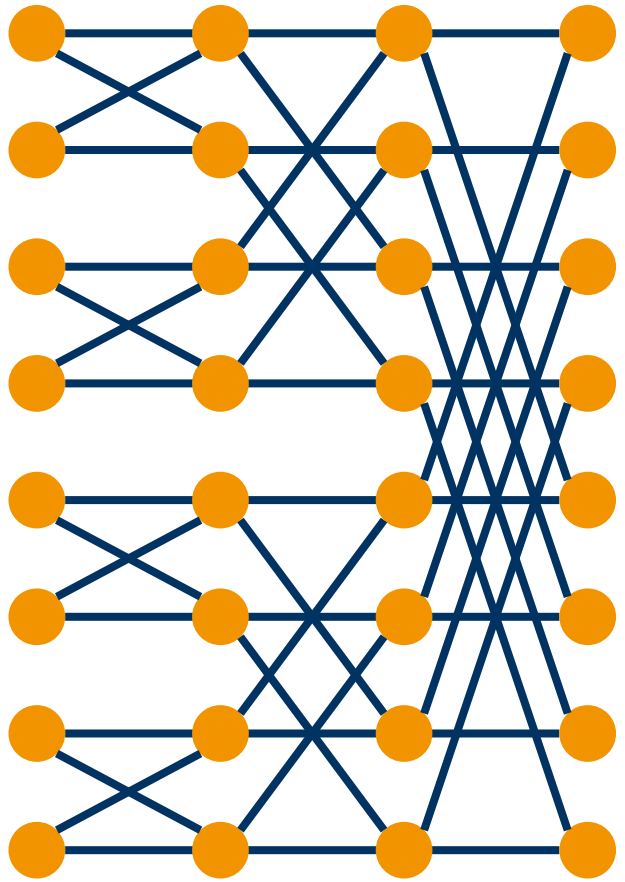
Spectral methods

- data in frequency domain, not time or space
  - can include multiple stages of computations alternating between local and global communication
- e.g. fast Fourier transform (FFT), audio and video signal processing
  - e.g. “focus hunting” in contrast-based autofocus of photo/video cameras



Spectral methods: characteristics

- usually implemented using butterfly patterns
  - multiple stages of multiply-add
  - often latency limited due to global communication patterns (e.g. all-to-all)
- often resemble structured or unstructured grid methods after transformation to frequency domain

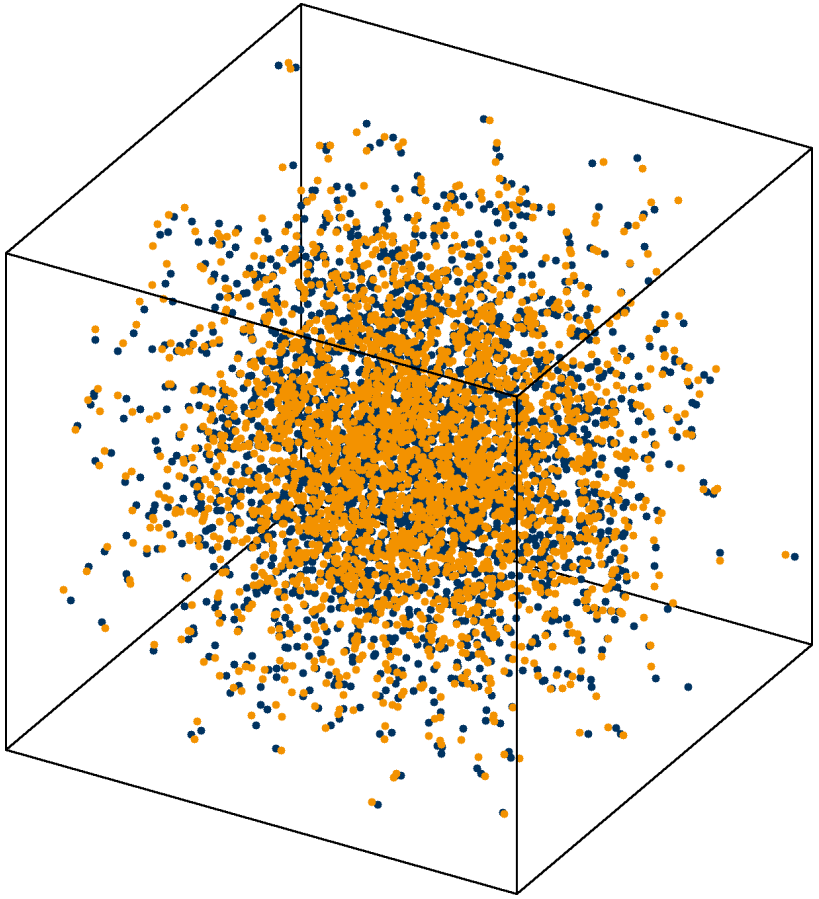


Spectral methods: optimizations

- requires optimization of the transformation to frequency domain
  - relies on transposing data efficiently
- afterwards, consider similar optimizations as for (un)structured grids
- severely restricted scalability on larger HPC systems
  - ongoing research
  - lots of it in math

N-body methods

- models interactions between discrete, moving points
  - often requires dynamic data structures
  - varying spatial locality
  - movement affects load balance and data access costs
- e.g. galaxy collision simulations, molecular dynamics, protein folding



N-body Methods: Characteristics

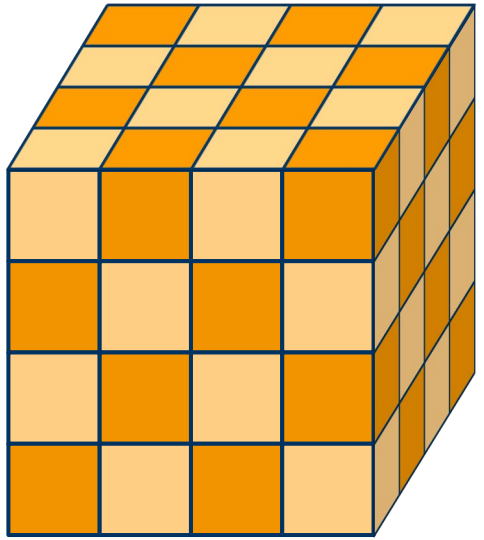
- don’t want to spoil this week’s assignment…

N-body Methods: Optimizations

- don’t want to spoil this week’s assignment…

Structured grids

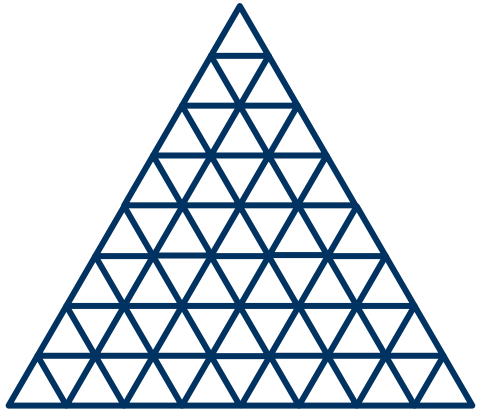
- model interactions between discrete, fixed points
  - grid structure described by pattern
  - topological information easily derived
  - usually high spatial locality
  - may be subdivided into finer grid (“adaptive”)
- e.g. heat transfer (stencil), computational fluid dynamics (CFD), octrees

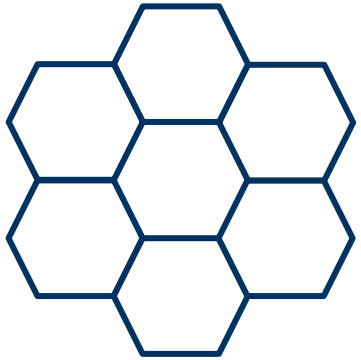


Structured grids: characteristics

- structured nature is a key aspect
  - similar characteristics compared to dense linear algebra (e.g. row-major vs. column-major)
  - memory access patterns & addresses often predictable, facilitates e.g. prefetching
  - adaptive grids and multi-grids possible
- typically memory bound
  - e.g. 7-point stencil in 3D: load 7 data points for computing a new one
  - local communication only (ghost cell exchange with direct neighbor)

- structured grid does not imply rectangular!

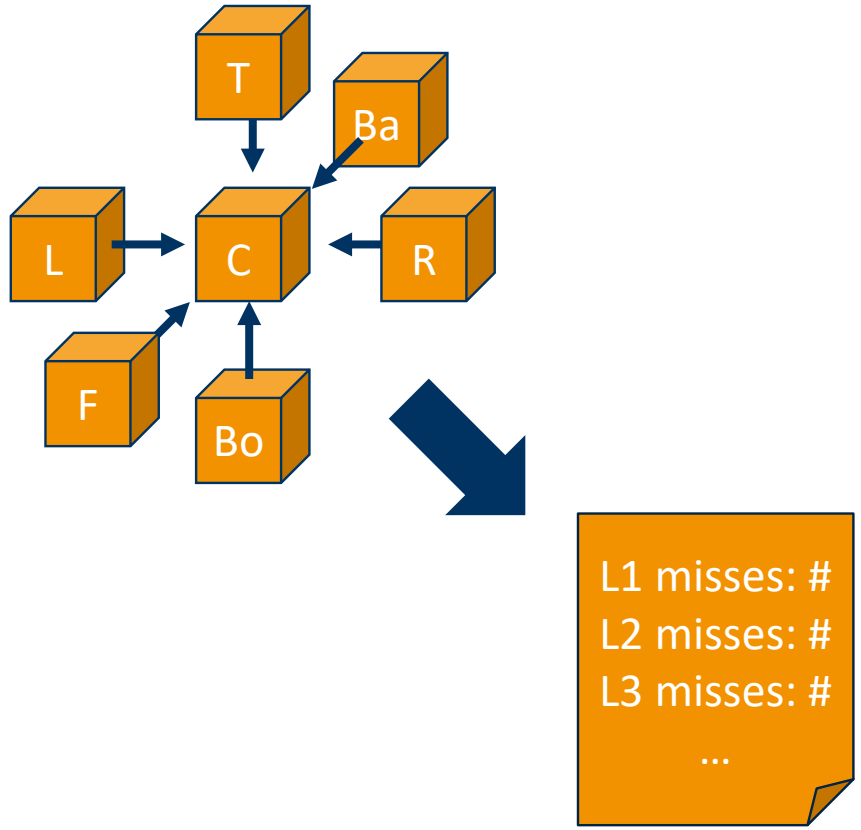




Structured grids: optimizations

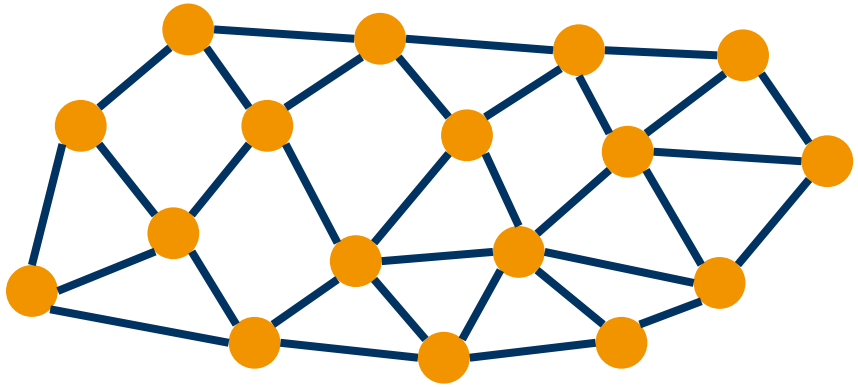
- decomposition, decomposition, decomposition
  - highly dependent on type of grid and specific use case

- structured nature of the problem makes analytic prediction possible
  - performance models and simulators for simple stencil kernels (e.g. Kerncraft)



Unstructured grids

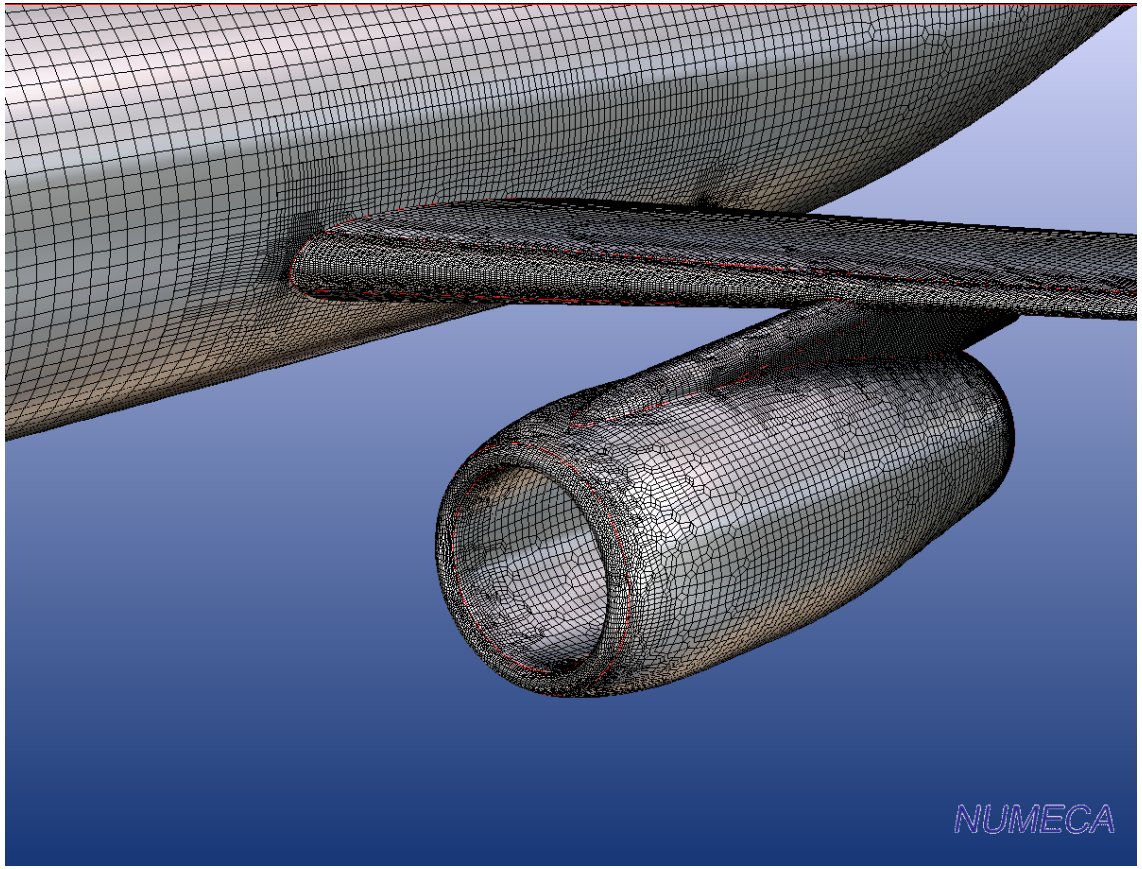
- model interactions between discrete, fixed points
  - grid pattern described explicitly by individual connections
  - irregular geometry and topology
  - usually involves multiple levels of indirection when accessing data



- e.g. computational fluid dynamics (CFD)

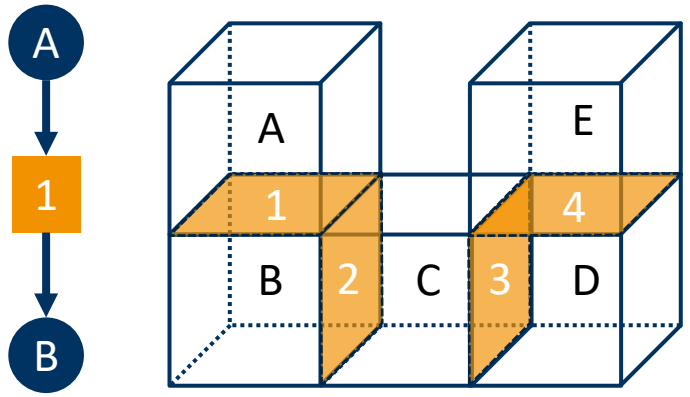
Unstructured grids: characteristics

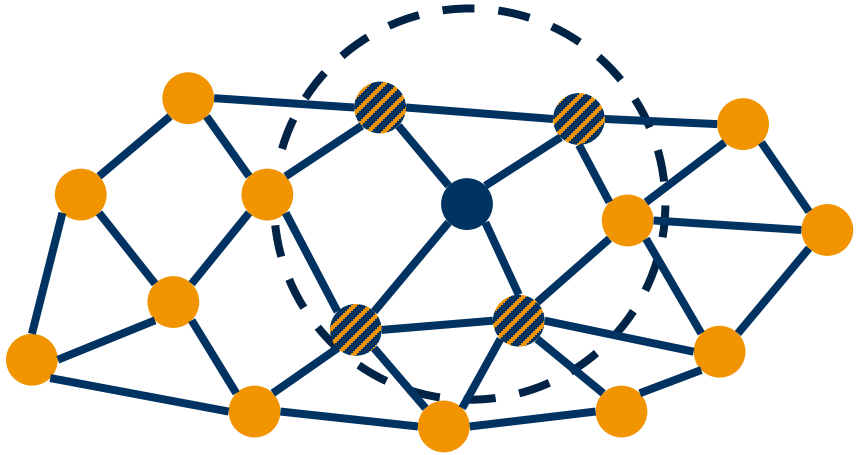
- usually heavily latency bound due to indirect access
  - `… = cells[neighbors[i]]`
  - `… = cell.getNeighbor(i)`
  - also known as “pointer chasing”
- similar problems compared to structured grids, e.g.
  - domain decomposition / adaptivity
  - topological information
  - ghost cell exchange
  - but hardly analytically predictable



Unstructured grids: optimizations

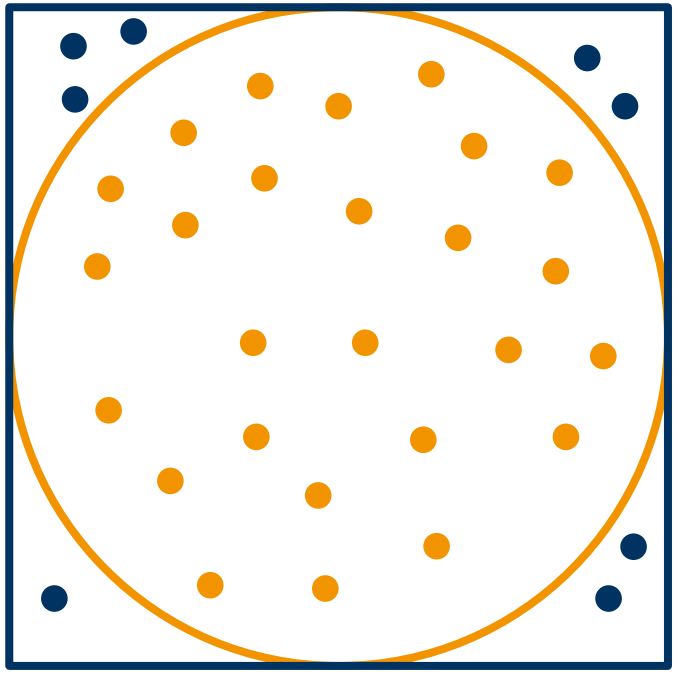
- problem space discretization and topology
  - which types of grid elements?
  - which types of connections?
  - cells, faces, vertices, edges, …
  - should be efficient to load/store, navigate, and compute
- decomposition, decomposition, decomposition
  - efficient ghost cell exchange required
  - efficient grid navigation e.g. to access neighbors





Monte Carlo methods

- also known as map-reduce
  - process data independently and merge the results
- models statistical evaluation of repeated random trials
  - communication usually insignificant
  - embarrassingly parallel, multiple copies of sequential method
- e.g. numerical integration, quantum many-body problems, ray tracing



Monte Carlo methods: characteristics

- parallelization almost a no-factor
  - similar to multiple sequential programs sharing some resources (e.g. L3 cache, random number generator, thermal envelope of CPU)
  - relatively inexpensive reduction
- depends heavily on sequential speed and shared bottlenecks

Monte Carlo methods: optimizations

- not much to do beyond sequential optimization
  - ILP
  - prefetching
  - vectorization
  - reduce resource contention
    (read: fast random number generation)

- consider different hardware
  - GPUs
  - FPGAs
  - …

- try to decrease cost of evaluating a sample
  - increase number of samples if required

Additional Dwarfs

Additional dwarfs

- 8. Combinational Logic
- 9. Graph Traversal
- 10. Dynamic Programming
- 11. Backtrack & Branch+Bound
- 12. Graphical Models
- 13. Finite State Machine

- slightly different focus compared to first 7 dwarfs
  - more (but not exclusively) on integer-heavy applications, machine learning, theoretical problems
  - less on physical processes

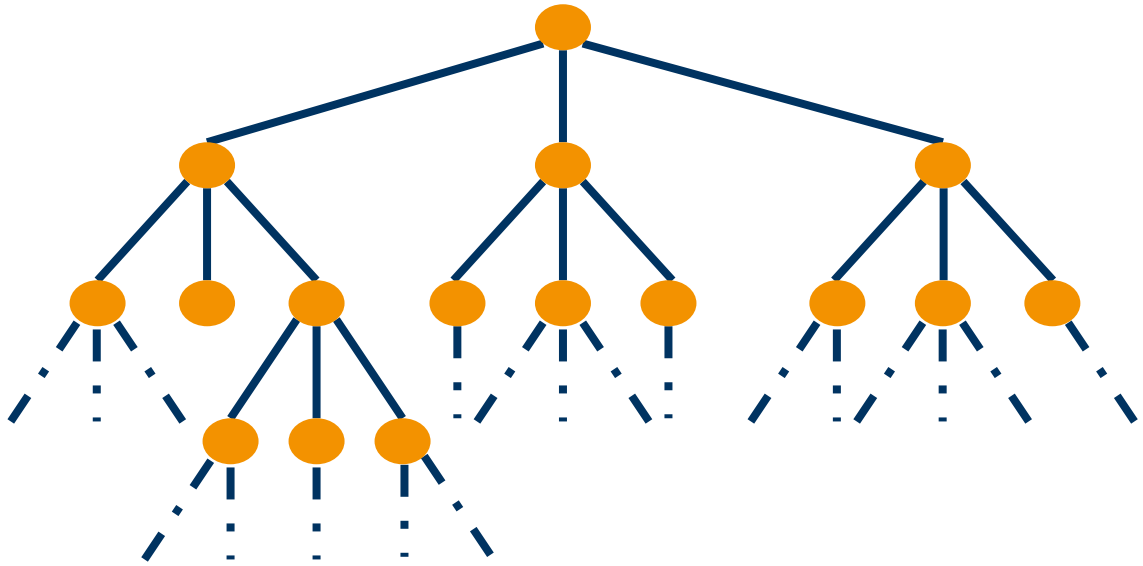
Combinational logic

- generally involves performing simple operations on large amounts of integer data
  - e.g. computing cyclic redundancy codes (CRC)
- often parallelizable on multiple levels
  - bit-level parallelism (e.g. x86 popcnt)
  - block-level parallelism

```
uint8_t compute(uint8_t const msg[], int n) {
    uint8_t rem = 0;
    for (int byte = 0; byte < n; ++byte) {
        rem ^= (msg[byte] << (WIDTH - 8));
        for (uint8_t bit = 8; bit > 0; --bit) {
            if (rem & TOPBIT) {
                rem = (rem << 1) ^ POLYNOMIAL;
            } else {
                rem = (rem << 1);
            }
        }
    }
    return (rem);
}
```

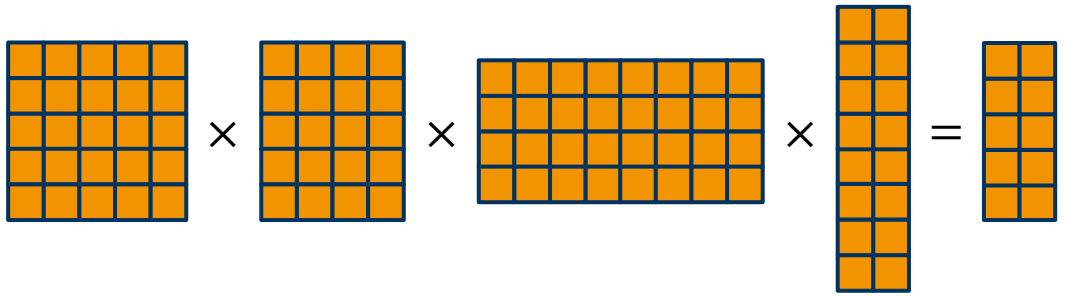
Graph traversal

- traverse a number of objects in a graph and examine characteristics
  - e.g. searching, sorting, collision detection, decision trees, …
  - usually heavy on data reads and lookups, very little computation and output
- parallelizable over different paths in the graph
  - but indirect accesses are heavily latency-bound (c.f. unstructured grids)



Dynamic programming

- method of computing solutions by solving simpler, overlapping sub-problems
  - applicable to problems where optimal result is composed of optimal results of sub-problems
  - e.g. matrix-chain-multiplication
- usually based on memoization
  - solve each sub-problem exactly once
  - store and re-use the result



$A \times \big(B \times (C \times D)\big) = X$ 154 ops

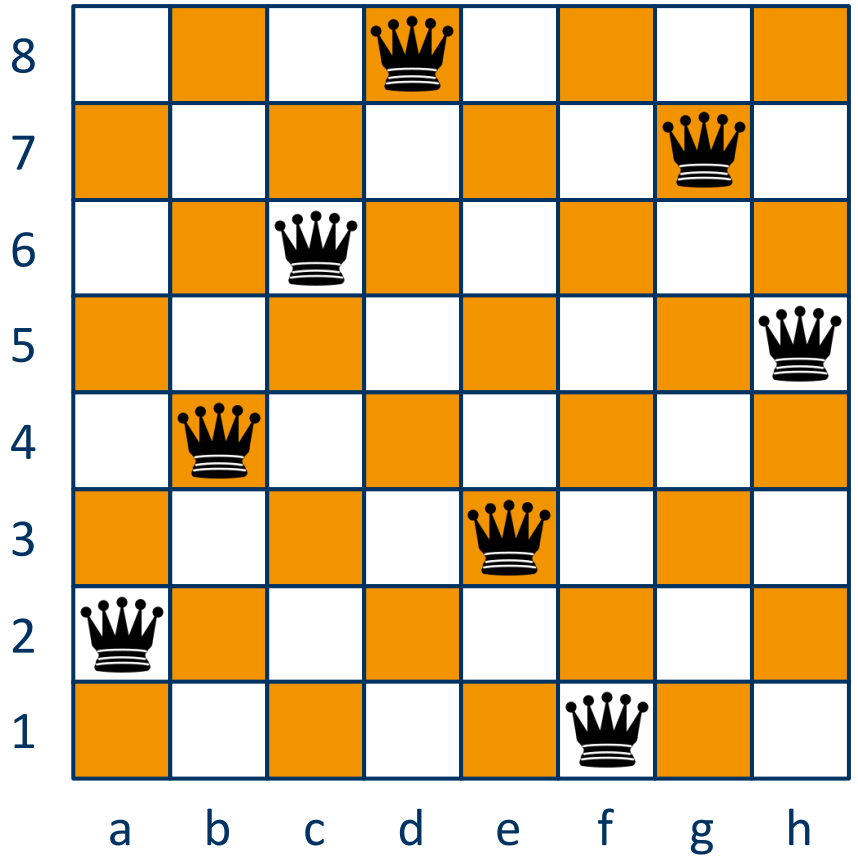
$(A \times B) \times (C \times D) = X$ 204 ops

$\big((A \times B) \times C\big) \times D = X$ 340 ops

Backtrack & Branch+Bound

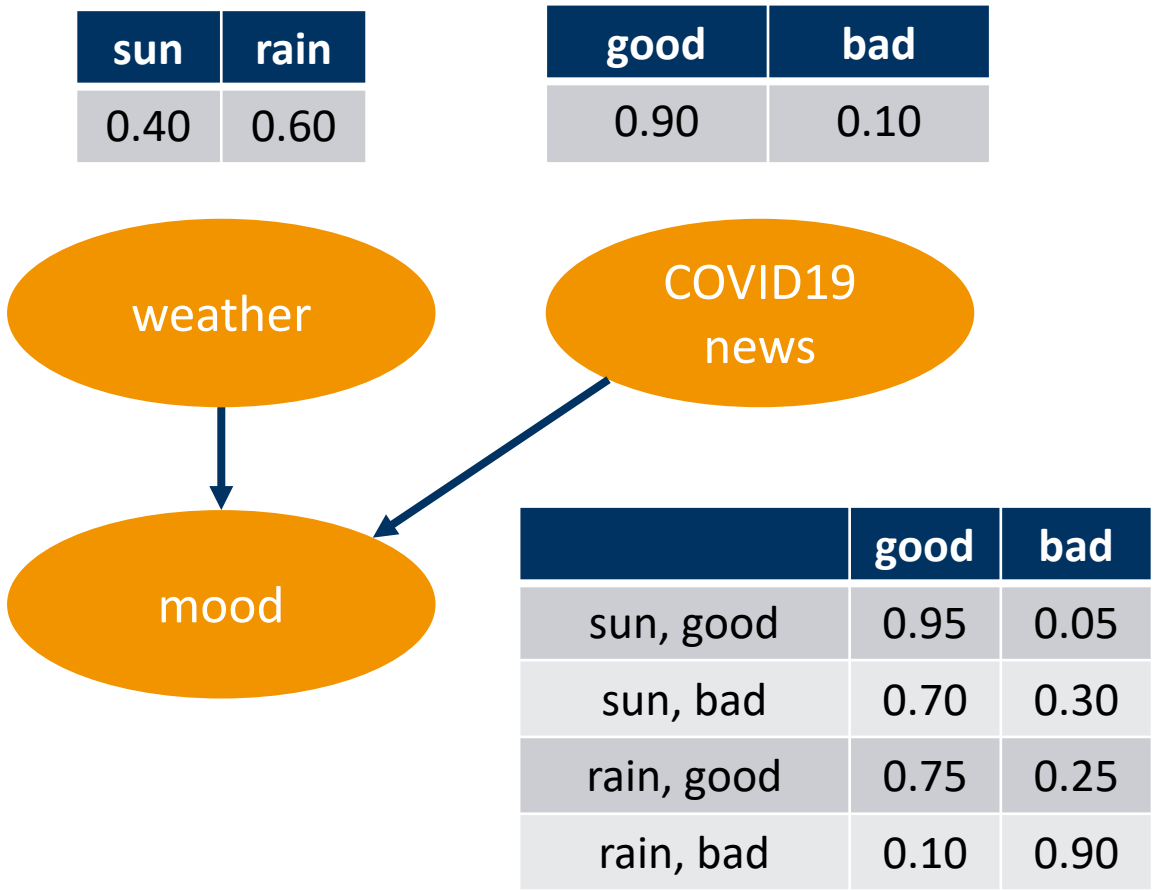
- search and optimization problems for very large problem spaces
  - incrementally build solution but discard if determined unsuitable
  - e.g. n-queens problem

- use divide & conquer strategy:
  - break down complex problem into smaller sub-problems until they become solvable
  - solve sub-problems in parallel



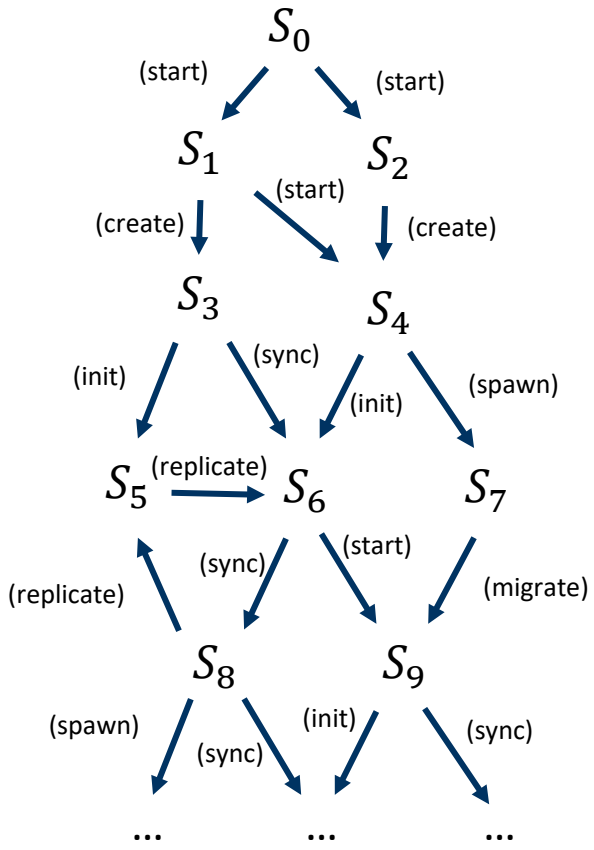
Probabilistic graphical models

- represent graphs consisting of random variables as nodes and dependencies as edges
  - e.g. Bayesian networks, Hidden Markov models
- ongoing research in math and computer science regarding parallelization and optimization



Finite state machines

- represent interconnected set of states to be moved among
  - e.g. parsers
- can sometimes be decomposed into multiple state machines that act in parallel
  - ongoing research



Literature material

- White Paper by Berkeley University: „The Landscape of Parallel Computing Research: A View from Berkeley” from 2006: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf
- Holds more detailed descriptions and related aspects

Note the focus on scientific problems

- MPI can be (and is!) used to implement also
  - distributed-memory runtime systems
  - emulate shared memory runtime systems on distributed memory (e.g. PGAS)
  - provide the connecting parallelism layer for shared-memory or sequential systems
    - e.g. use multiple accelerators in separate compute nodes (Celerity project @ UIBK)
    - extend shared memory parallelism to distributed memory (MPI+X)
    - …
- still, the majority of codes is of scientific computing nature

Side note: Portability

- functional portability
  - program runs on different hardware and produces correct results
  - hardware architecture, compiler, libraries, etc.
  - usually not hard to achieve (mostly x86/ARM + GPUs)

- performance (also “non-functional”) portability
  - program runs efficiently on different hardware and is considered “fast”
  - no exact definition, topic of ongoing research

Summary

- 13 Dwarfs of HPC
  - abstract application categories
  - facilitate cross-platform reasoning and cross-application component reuse
  - 7 older ones, most well-studied physics problems
  - 6 newer ones, partially of theoretical nature, subject to ongoing research
  - give a broad perspective on HPC potential and limitations

Image Sources

- Dwarfs: http://pngimg.com/download/47261, http://rpgmke.wikidot.com/moonshae-isles-campaign
- Barnes-Hut: http://portillo.ca/nbody/barnes-hut/
- Unstructured Mesh: https://resourcearea.cpu-24-7.com/en/numeca_welcome
- CRC: https://barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code