



703308 VO High-Performance Computing WS2021/2022

The 13 Dwarfs of HPC

Philipp Gschwandtner

Overview

- ▶ The 13 Dwarfs of HPC
 - ▶ abstract application categories
- ▶ “Tales from the Proseminar”
 - ▶ Daniel’s Weird `int` Problem
 - ▶ g++ vs. gcc when compiling C code

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 applications and discuss them
- ▶ this lecture presents the most prominent classes of HPC applications
 - ▶ many new applications you encounter will fit into these categories

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 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 to discuss?

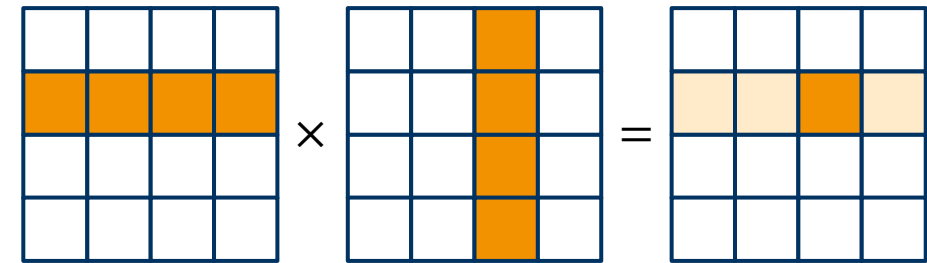
- ▶ 1. Dense Linear Algebra
- ▶ 2. Sparse Linear Algebra
- ▶ 3. Spectral Methods
- ▶ 4. N-body Methods
- ▶ 5. Structured Grids
- ▶ 6. Unstructured Grids
- ▶ 7. Monte Carlo Methods

▶ What have you already heard?

- ▶ matrix mul (first MPI lecture)
- ▶ 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 often 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]  
...
```

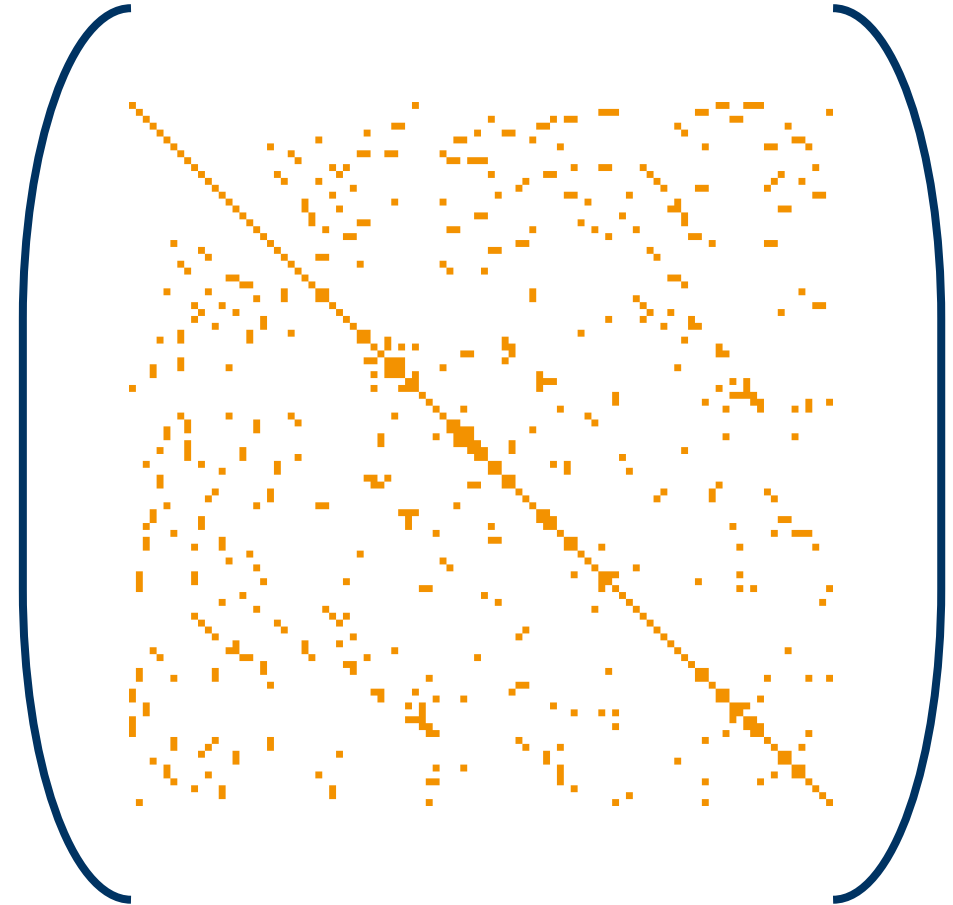
Dense Linear Algebra: Optimizations

- ▶ 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 Orders 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;
```

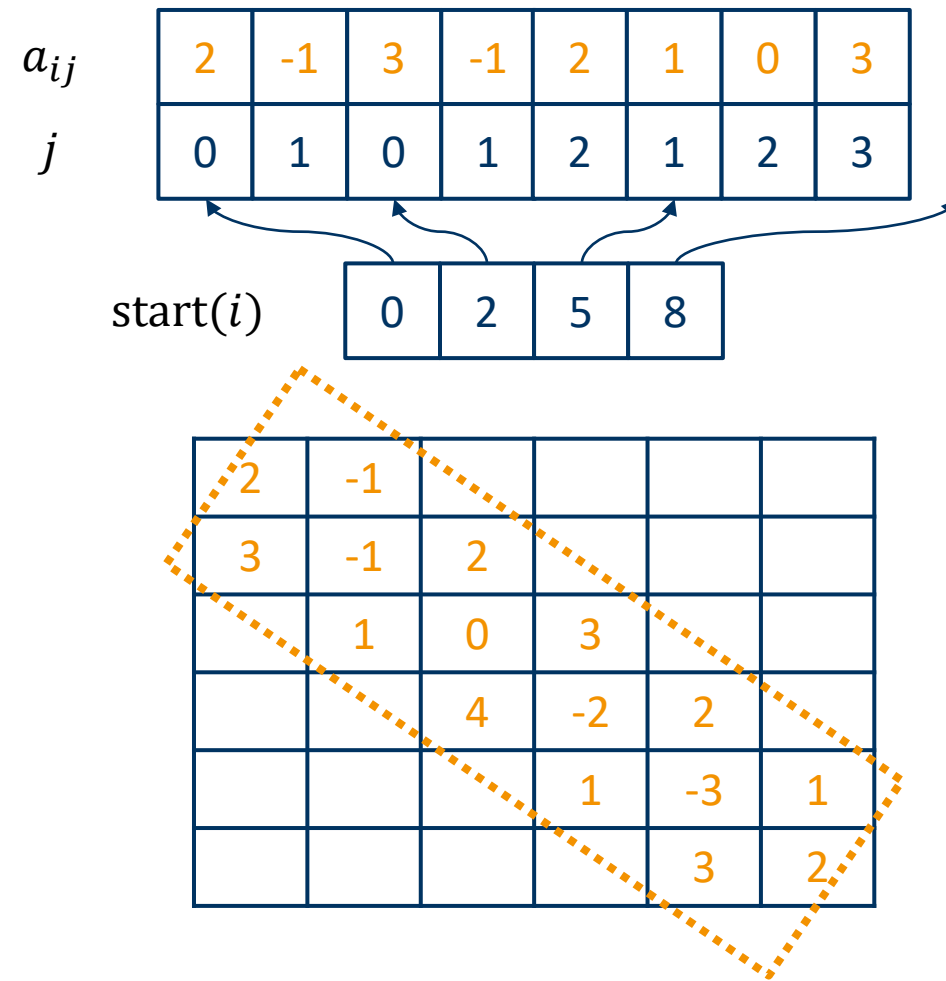
Sparse Linear Algebra: Optimizations

► 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. $3N$

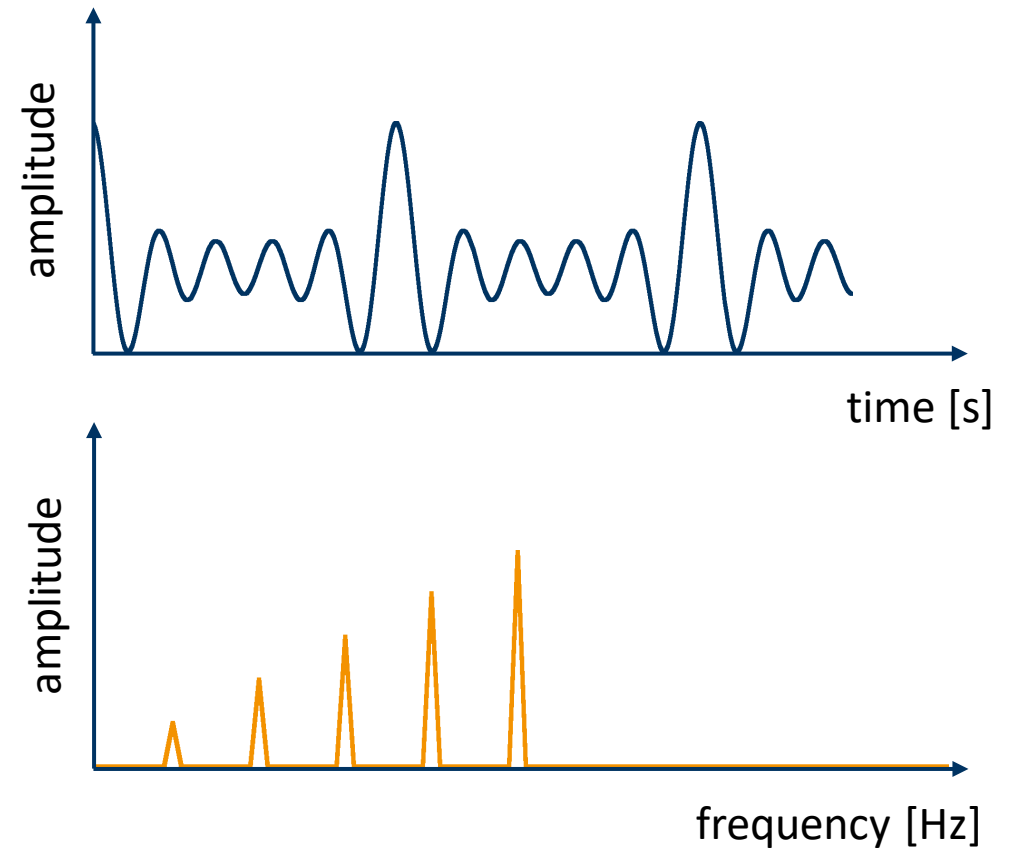
► variants

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



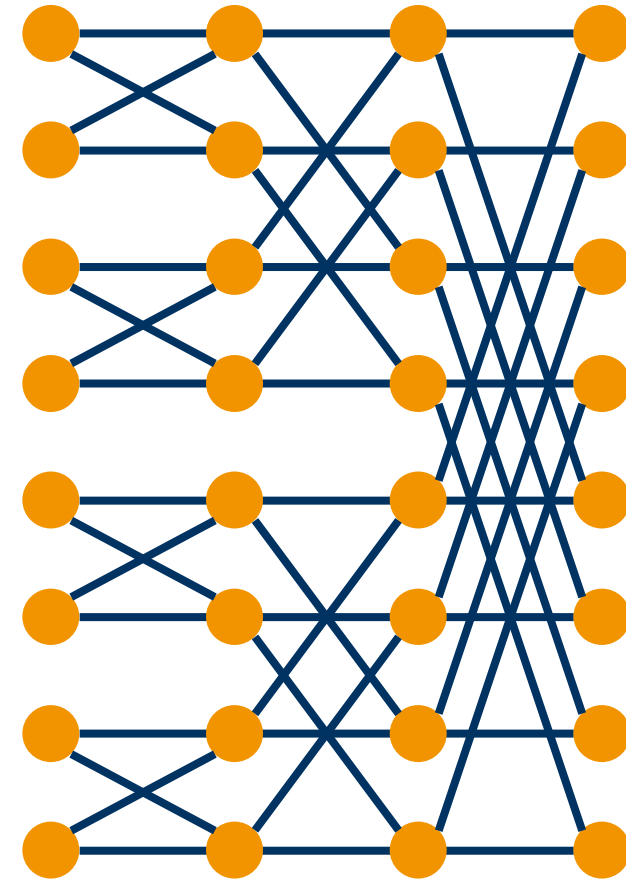
Spectral Methods

- ▶ 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

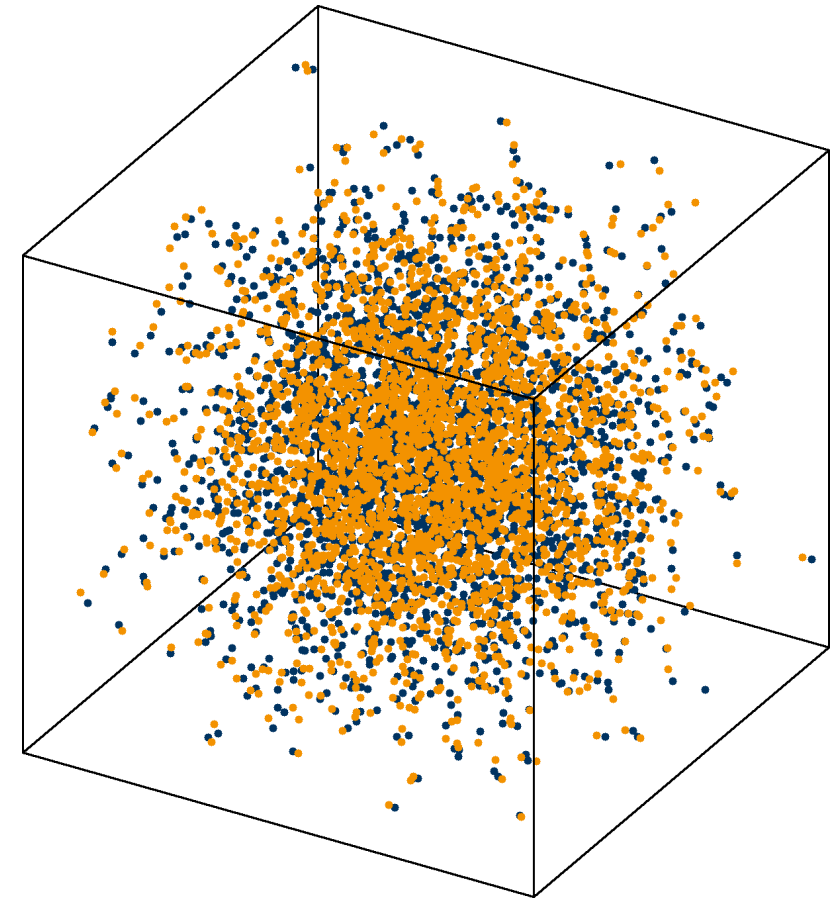


Spectral Methods: Optimizations

- ▶ 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 to 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

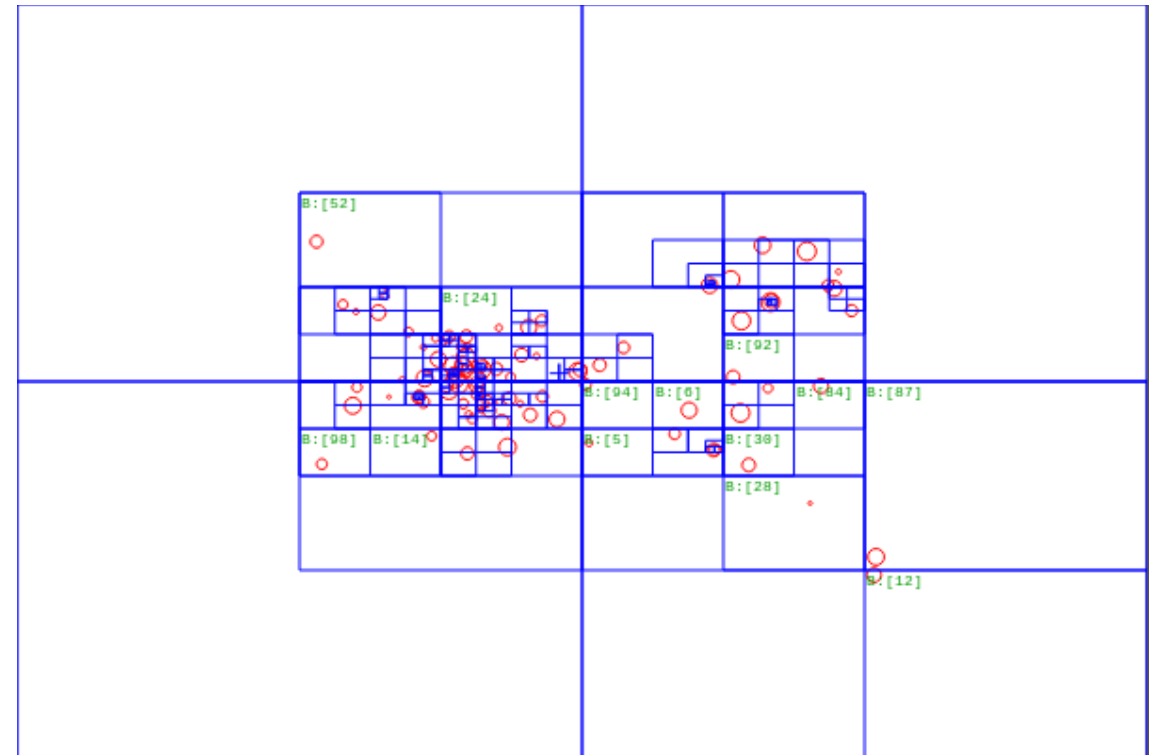


N-body Methods: Characteristics

- ▶ computational effort is an issue
 - ▶ $\mathcal{O}(N^2)$ for N particles
 - ▶ but also global communication
- ▶ lots of hierarchical optimization studies
 - ▶ domain decomposition!
 - ▶ e.g. Barnes-Hut or fast multipole optimization
- ▶ alternative approaches rely on domain-specific knowledge, e.g.
 - ▶ ignore long-distance particle interactions
 - ▶ store particles in Cartesian topology & only consider particles in neighboring grid cells

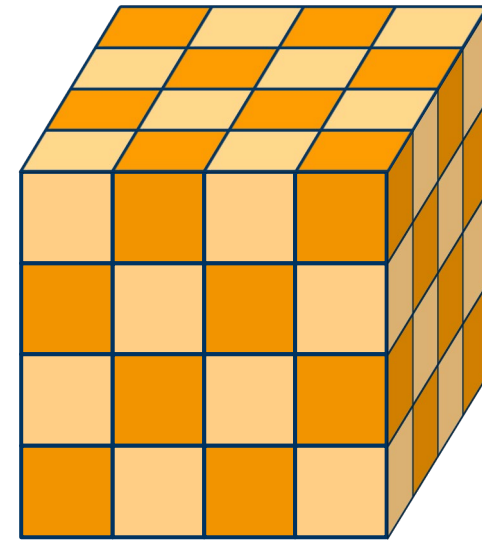
N-body Methods: Optimizations

- ▶ Barnes-Hut optimization:
 - ▶ decompose domain into quadtree (for 2D)
 - ▶ aggregate particle effect (e.g. gravitation from mass) for each cell into a single (hypothetical) particle at the center of gravity per cell
 - ▶ reduces complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \cdot \log N)$



Structured Grids

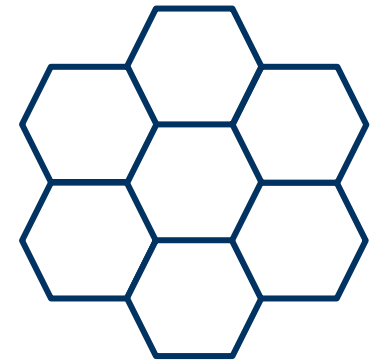
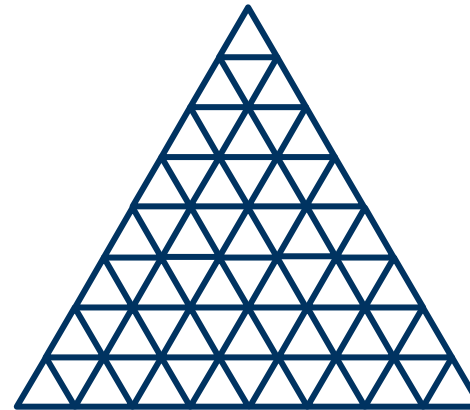
- ▶ model interactions between discrete, fixed points
 - ▶ grid structure described by pattern
 - ▶ topological information easily derived
 - ▶ usually high spatial locality
 - ▶ may be subdivided into finer grid (“adaptive”)
- ▶ e.g. heat transfer (stencil), computational fluid dynamics (CFD), octrees



Structured Grids: Characteristics

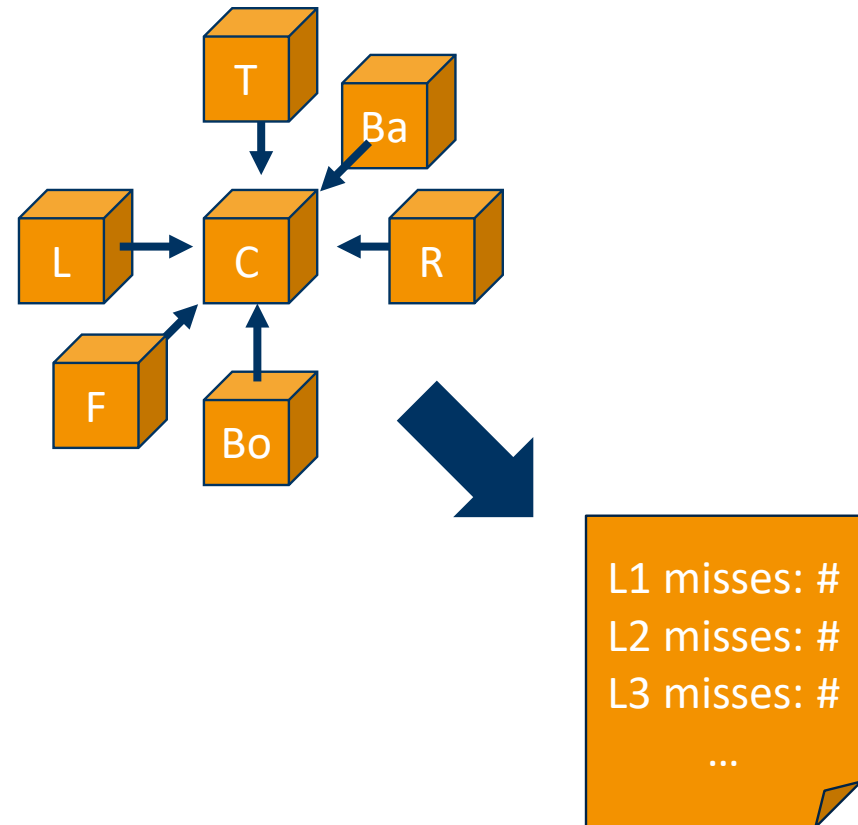
- ▶ **structured nature is a key aspect**
 - ▶ similar characteristics compared to dense linear algebra (e.g. row-major vs. column-major)
 - ▶ memory addresses often predictable, facilitates e.g. prefetching
 - ▶ adaptive grids and multi-grids possible
- ▶ **typically memory bound**
 - ▶ e.g. 7-point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

- ▶ **structured grid does not imply rectangular!**



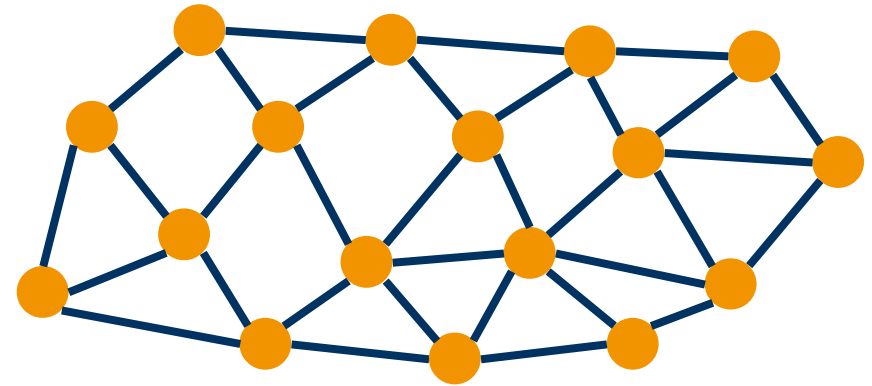
Structured Grids: Optimizations

- ▶ 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)



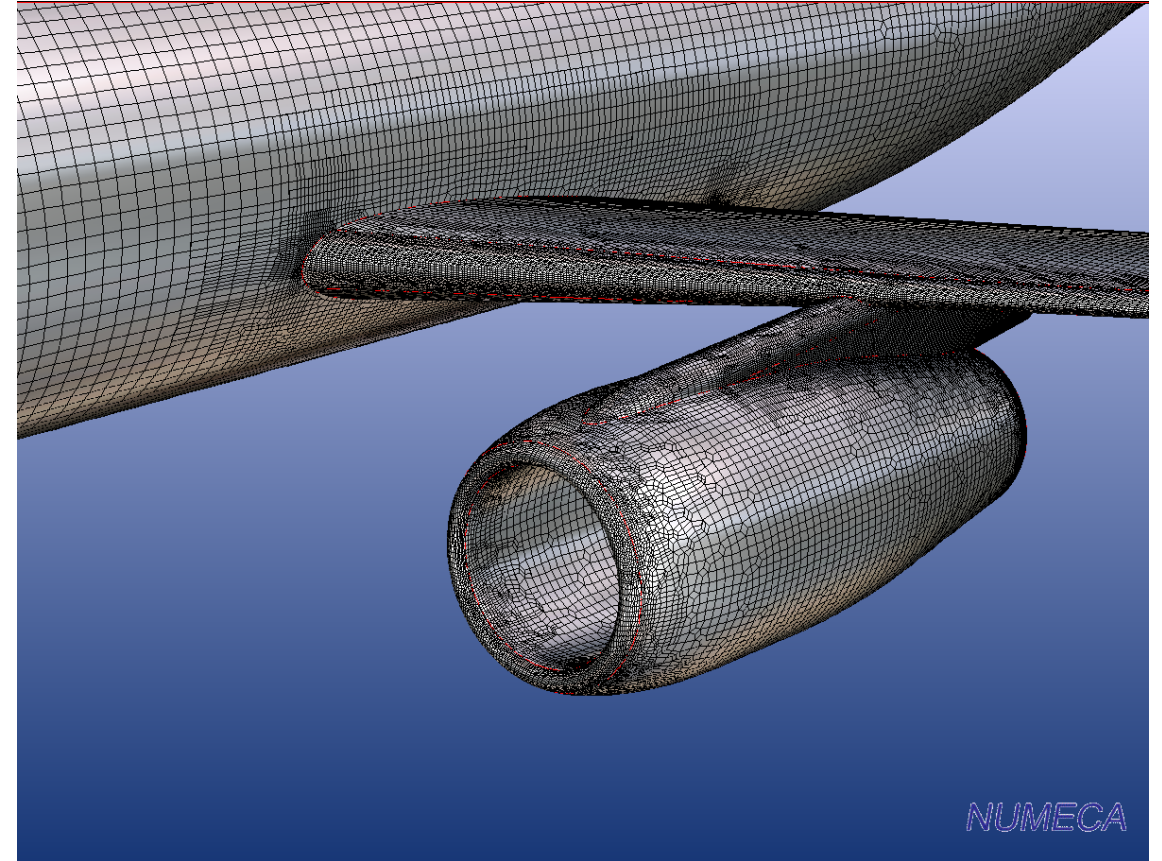
Unstructured Grids

- ▶ 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)



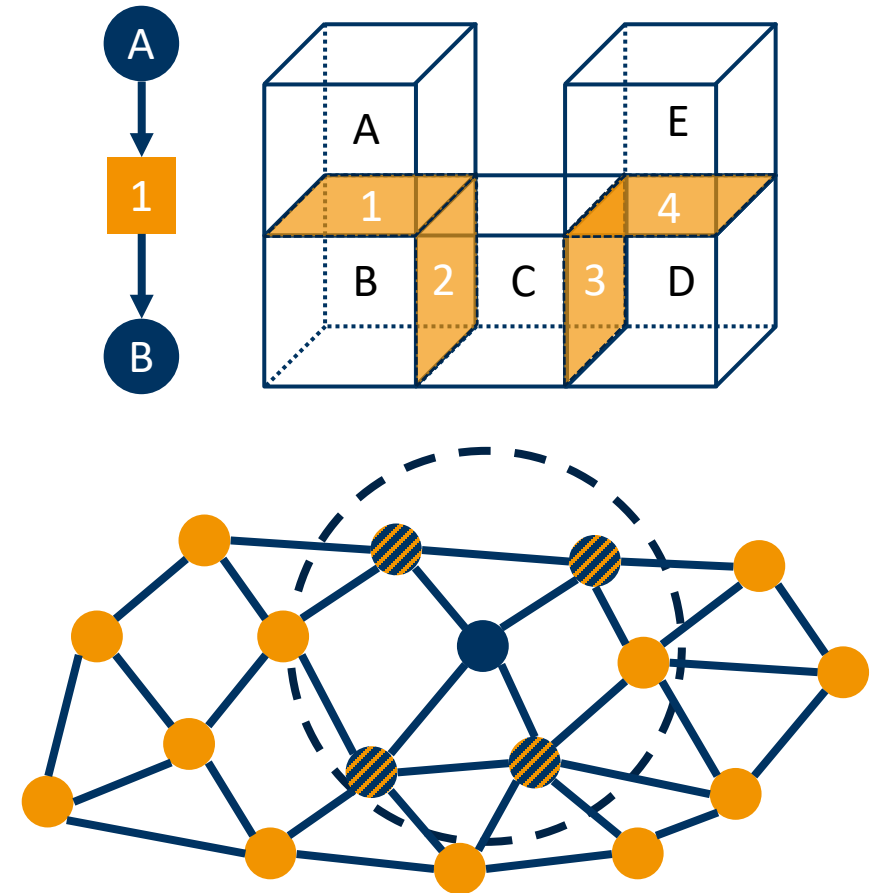
Unstructured Grids: Characteristics

- ▶ 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



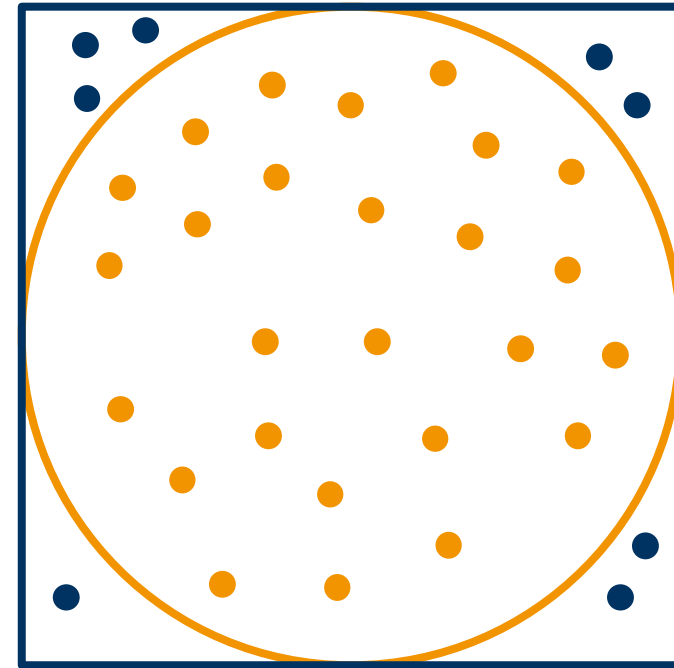
Unstructured Grids: Optimizations

- ▶ 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)
 - ▶ relatively inexpensive reduction
- ▶ depends heavily on sequential optimizations

Monte Carlo Methods: Optimizations

- ▶ not much to do beyond sequential optimization

- ▶ ILP
- ▶ prefetching
- ▶ vectorization
- ▶ 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 and machine learning
 - ▶ less on physical processes

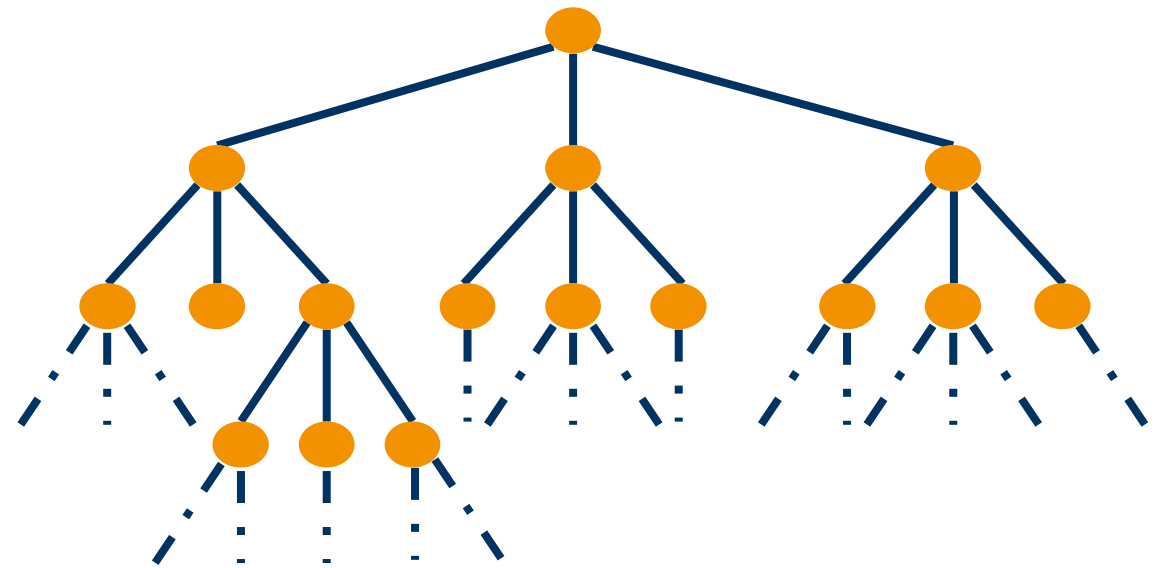
Combinational Logic

- ▶ 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);  
}
```

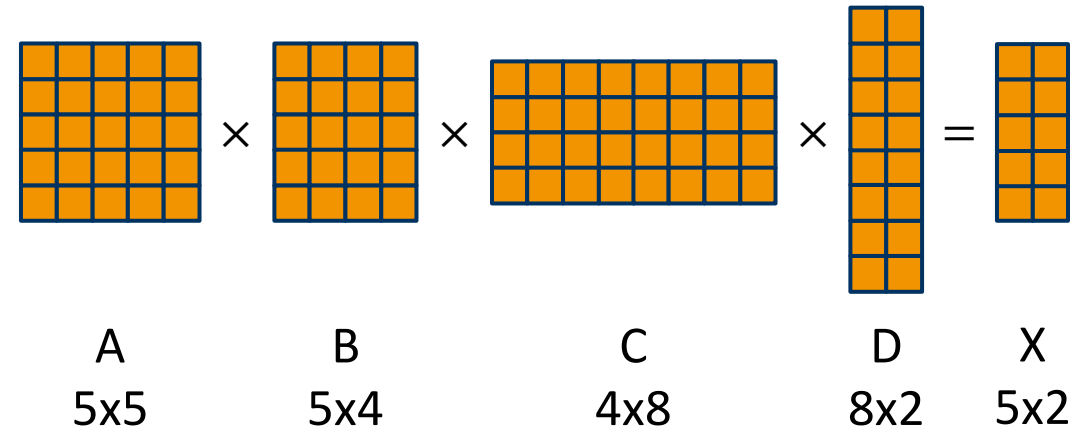
Graph Traversal

- ▶ 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



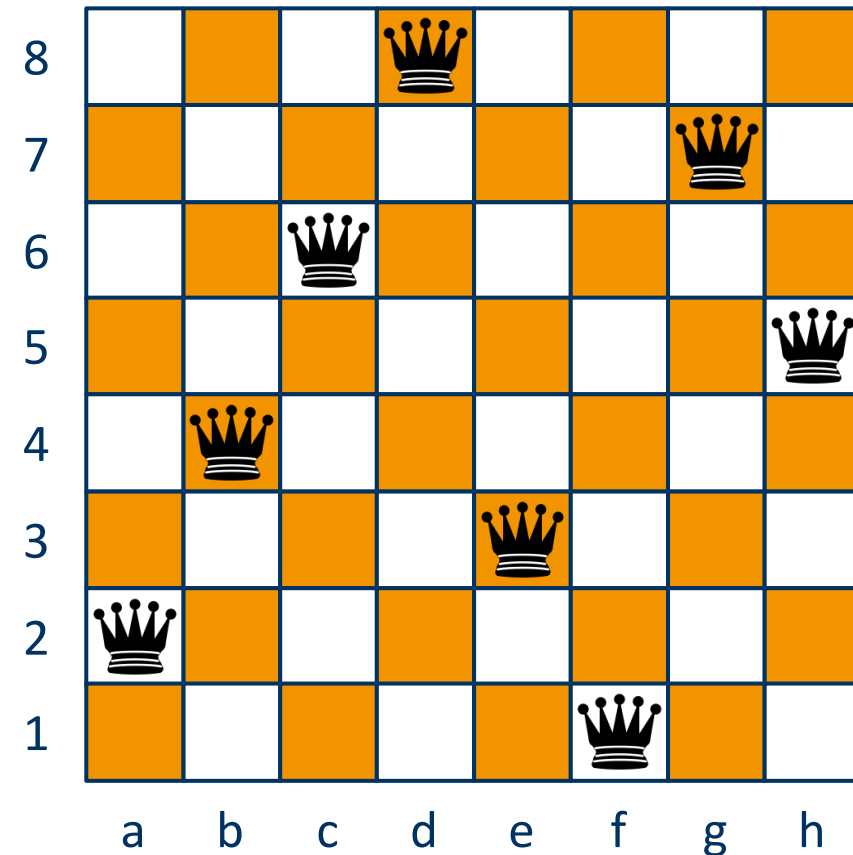
$$A \times (B \times (C \times D)) = X \quad 154 \text{ ops}$$

$$(A \times B) \times (C \times D) = X \quad 204 \text{ ops}$$

$$((A \times B) \times C) \times D = X \quad 340 \text{ ops}$$

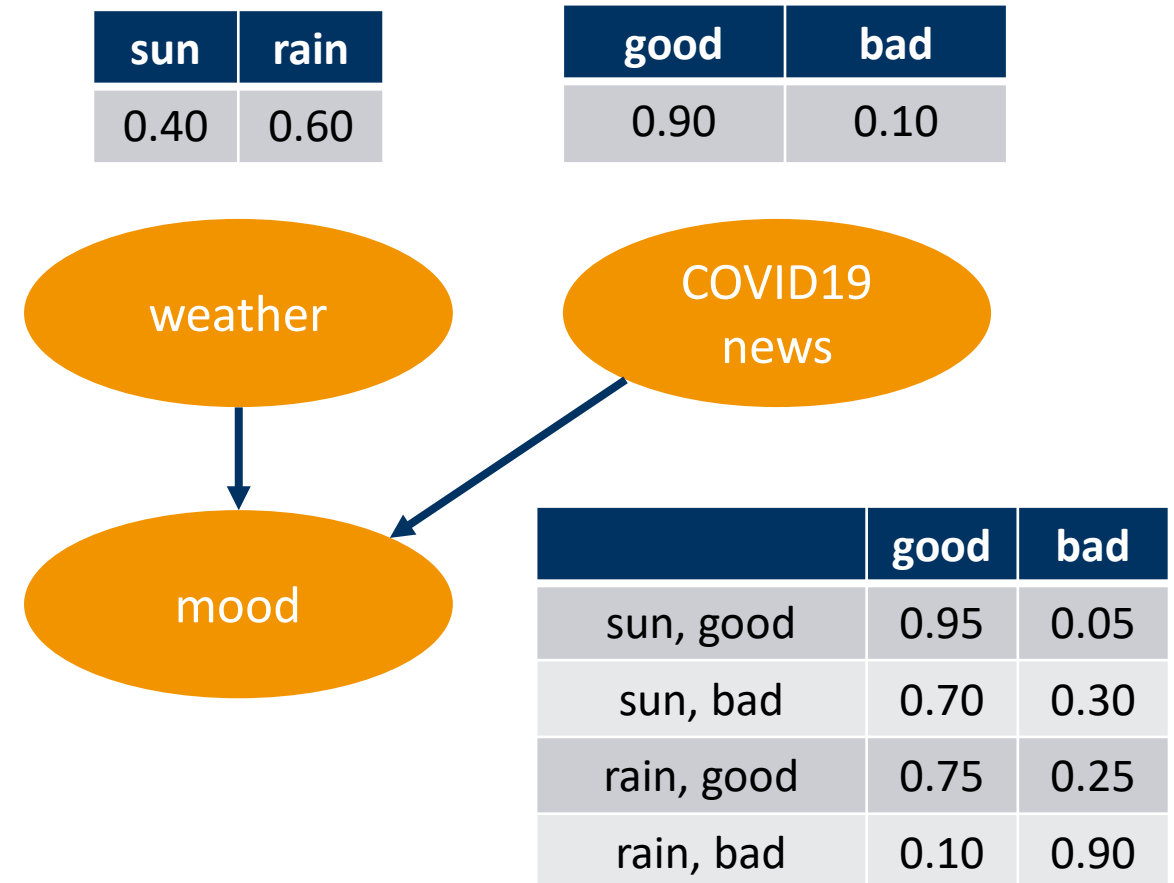
Backtrack & Branch+Bound

- ▶ 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



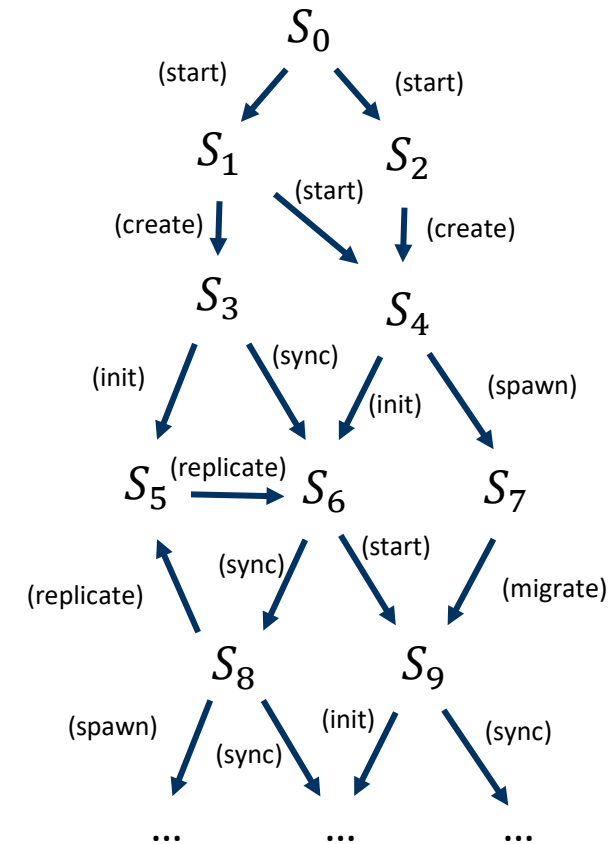
Probabilistic Graphical Models

- ▶ 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



Tales from the Proseminar



Tales from the Proseminar: Daniel's Weird `int` Problem

- ▶ sequential 2D heat stencil
 - ▶ 100x100 problem size
 - ▶ -O0 (also with -O2) on LCC2
 - ▶ gcc 4.8.5 (but also 9.2 and MSVC 2019)
 - ▶ switching from `long` `long` loop iterators to `int` **halves** execution time

```
$ /usr/bin/time -f %E ./stencil_long 100 100
0:04.60
$ /usr/bin/time -f %E ./stencil_int 100 100
0:02.31
```

- ▶ What the heck is going on?

Tales from the Proseminar: Daniel's Weird `int` Problem

- ▶ profiled with gprof to find “hot spots”, left is `int`, right is `long long`

% time	cumulative seconds	self seconds	name
32.92	0.76	0.76	int.c:93
20.06	1.22	0.46	int.c:78
14.83	1.56	0.34	int.c:79
10.47	1.80	0.24	int.c:87
9.59	2.02	0.22	int.c:86

% time	cumulative seconds	self seconds	name
36.84	1.69	1.69	long.c:79
31.61	3.15	1.45	long.c:78
16.35	3.90	0.75	long.c:93
5.34	4.15	0.25	long.c:86
3.27	4.30	0.15	long.c:87

Tales from the Proseminar: Daniel's Weird `int` Problem cont'd

```
74 // get temperature at current position
75 value_t tc = A[i];
76
77 // get temperatures of adjacent cells
78 value_t txl = (i % Nx != 0) ? A[i - 1] : tc;
79 value_t txr = (i % Nx != Nx - 1) ? A[i + 1] : tc;
    // ..... snip .....
90 if (Ny > 1)
91     B[i] = tc + 0.165 * (txl + txr + tyl + tyr + tzl + tzt + (-6 * tc));
92 else
93     B[i] = tc + 0.2 * (txl + txr + tzl + tzt + (-4 * tc));
94 // if ((int)B[i] < (int)A[i])
```


Tales from the Proseminar: Daniel's Weird `int` Problem

- ▶ far-fetched idea, but maybe branch (miss-)predictions? compared with `perf stat`:

```
2328.650634 task-clock:u (msec) #    0.996 CPUs
           0 context-switches:u #    0.000 K/sec
           0 cpu-migrations:u   #    0.000 K/sec
          186 page-faults:u      #    0.080 K/sec
5,739,935,525 cycles:u          #    2.465 GHz
10,671,532,880 instructions:u   #    1.86  IPC
1,196,623,336 branches:u       # 513.870 M/sec
    1,030,678 branch-misses:u  #    0.09%
```

2.338387625 seconds time elapsed

```
4639.728721 task-clock:u (msec) #    0.998 CPUs
           0 context-switches:u #    0.000 K/sec
           0 cpu-migrations:u   #    0.000 K/sec
          186 page-faults:u      #    0.040 K/sec
11,444,184,736 cycles:u         #    2.467 GHz
10,972,976,005 instructions:u   #    0.96  IPC
1,196,977,569 branches:u       # 257.984 M/sec
    1,030,347 branch-misses:u  #    0.09%
```

4.650327844 seconds time elapsed

Tales from the Proseminar: Daniel's Weird `int` Problem cont'd

```
value_t tx1 = ( i % Nx != 0 ) ? A[i-1] : tc;
```

```
mov    eax, DWORD PTR [rbp-20]
cdq


idiv   DWORD PTR [rbp-24]


mov    eax, edx
test   eax, eax
je     .L2
mov    eax, DWORD PTR [rbp-20]
cdqe
```

```
sal    rax, 3
lea    rdx, [rax 8]
mov    rax, QWORD PTR [rbp-32]
add    rax, rdx
movsd  xmm0, QWORD PTR [rax]
jmp    .L3
.L2:   ...
.L3:   ...
```

Tales from the Proseminar: Daniel's Weird `int` Problem cont'd

- ▶ Found instruction information on Agner Fog's blog for Intel Skylake architecture:
 - ▶ https://www.agner.org/optimize/instruction_tables.pdf
- ▶ if you want to study compiler output: <https://godbolt.org/>

inst.	operands	μops	latency
<code>idiv</code>	r32	10	26
<code>idiv</code>	r64	57	42-95

Tales from the Proseminar: Daniel's Weird `int` Problem cont'd

- ▶ root cause of the issues: single loop for multi-dimensional problem space
 - ▶ requires % operator to get boundaries (which are strided in linearized space)
 - ▶ replace with loop nest and comparison operators (>, <, ==, !=)
- ▶ potentially premature optimization and violates step 1 of “*Four Steps to Creating an Optimized Parallel Program*”

Tales from the Proseminar: g++ vs. gcc for C Code

g++

```
void foo() {  
    for(int i = 0; i < 10; ++i) { }; return;  
}
```

gcc

```
push    rbp  
mov     rbp, rsp  
mov     DWORD PTR [rbp-4], 0
```

```
.L3:  
    cmp     DWORD PTR [rbp-4], 9  
    jg      .L2  
    add     DWORD PTR [rbp-4], 1  
    jmp     .L3
```

```
.L2:
```

```
    nop  
    pop     rbp  
    ret
```

```
push    rbp  
mov     rbp, rsp  
mov     DWORD PTR [rbp-4], 0
```

```
    jmp     .L2
```

```
.L3:  
    add     DWORD PTR [rbp-4], 1  
.L2:
```

```
    cmp     DWORD PTR [rbp-4], 9  
    jle     .L3
```

```
    nop  
    pop     rbp  
    ret
```

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

▶ “Tales from the Proseminar”

- ▶ Daniel’s Weird `int` Problem: Don’t consider all `int` instructions to be cheap!
- ▶ `g++` vs. `gcc` when compiling C code

Image Sources

- ▶ Dwarfs: <http://pngimg.com/download/47261>, <http://rpgmke.wikidot.com/moonshae-isles-campaign>
- ▶ CRC: <https://barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code>
- ▶ Barnes-Hut: https://en.wikipedia.org/wiki/Barnes%E2%80%93Hut_simulation
- ▶ Unstructured Mesh: https://resourcearea.cpu-24-7.com/en/numeca_welcome