Parallel Programming Concepts
What are the (computationally) demanding problems / applications of the future?

Dr. Peter Tröger
M.Sc. Frank Feinbube
What are the (computationally) demanding applications of the future?

- Computer Vision (for medicine) + as computer input
- Machine Learning
- Compilers + Runtime Environments
- (Climate) Simulation
- Personalized Medicine + Genome Data
- Fail-Safe Long-Running Applications (Infrastructures)
- Cryptography
- Sensor Networks
- Needle-in-a-haystack (Protein Folding)
- Holo-Decks
- Big Data (Analyzing Social Networks)
- Cloud Computing for Mobile Devices (PlayStation Now)
- Raytracing (3D Rendering)
- Compression (HD TV)
- AI + Pattern Matching
- Voice Recognition
- Games (+ Physics + AI)
Parallel Algorithms of this lecture = Algorithms, we considered to be important (and easy enough to be implemented in an exercise)

- **Algorithms?**
  Parallel Sum, Dining Philosophers, Heat Map, Parallel Grep, Decrypt, Worley Noise, Julia Fractal, Collective (Averages), Wator, Nqueens, Petri Net Simulation, Simulation with Time-Warp, Tensor Products, Cycles of War, ...

- **Algorithm Classes?**
  Reductions, Concurrency, Convolution, Scan, Pattern Matching, Encryption, Noise, Fractals, Simulation, Backtracking Search, Model checking, Matrix Multiplication, ...
What are the (computationally) demanding problems / applications of the future?

Benchmarks
Power Wall 2.0 = Dark Silicon + Algorithms, Esmaeilzadeh et. al considered to be important in the future

"Dark Silicon and the End of Multicore Scaling"
by Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, Doug Burger
The next slide shows algorithms from these sources:
(Algorithms that occurred again and again and thus seem to be important)

- PPC (this course)
- UPCRC Summer School on Parallel Computing
- Intel ArBB examples

- Parallel and Hybrid Benchmarks (Suites):
  - SHOC, HPC Challenge, PLASMA, Parboil, ALPBench, ParkBench, BioBench, BioParallel, UPC Standard

- Books & Papers
  - CUDA by Example
  - Parallel Benchmarks Inspired by Berkeley Dwarfs
  - State-of-the-art in heterogeneous computing
  - Rodinia: A Benchmark Suite for Heterogeneous Computing
  - An Introduction To Programming With X10
  - GPU Programming in a High Level Language - Compiling X10 to CUDA
Sudoku Validator, Sudoku Validator 2D, Crypt 3, Game Of Life, Merge, Par Grep, Wator, Heat Map, Parallel Sum, Matrix Multiplication 0, Matrix Multiplication 1, Matrix Multiplication 2, Matrix Multiplication 3, Matrix Multiplication 4, Matrix Multiplication 5, Convolution, Matrix Vector Multiplication, Minimum Spanning Tree, Prefix Scan, Quicksort, Vector Average, Dot Product, Heat Transfer, Histogram, Julia Set, Ray Tracing, Ripple, Summing Vectors, Breadth-first Search, FFT (forward and reverse, 1D), Molecular Dynamics, Reduction, SGEMM (Matrix Multiplication), Scan, Sort (Radix), Sparse Matrix-Vector Multiplication, Stencil 2D, STREAM Triad benchmark, S3D turbulent combustion simulation, Black Scholes Partial Differential Equation (PDE), bodytrack, Cache-aware annealing, Data deduplication, facesim, ferret (search server), Fluid animate, freqmine, raytrace, streamcluster, swaptions, vips (image processing), x264 (H.264 video encoding), HPL (Linpack TPP), DGEMM (double precision Matrix Multiplication), STREAM, PTRANS (parallel matrix transpose), FFT, Weighted Averages, 2D FFT, Basic Linear Algebra Subprograms (BLAS), Matrix Multiplications, FFT, Stencil Computations, Random Number Generation (Monte Carlo), Data-parallel primitives and sorting, Image processing and computer vision, Linear Systems of Equations (Cholesky, LDLT, LU with partial pivoting), Matrix Inversion (Cholesky, LU with partial pivoting), Least Squares (QR and LQ), Mixed Precision Iterative Refinement (linear systems using Cholesky or LU, least squares using QR or LQ), Symmetric Eigenvalue Problem (eigenvalues only), Singular Value Problem (singular values only), Level 3 Tile BLAS (GEMM, HEMM, HER2K, HERK, SYMM, SYR2K, SYRK, TRMM, TRSM), In-Place Layout Translation (CM, RM, CCRB, CRRB, RCRB, RRRB), K-Means, Needleman-Wunsch, Hot Spot, Back Propagation, SRAD, Leukocyte Tracking, Breadth-First Search, Stream Cluster, Similarity Scores, Heart Wall, MUMmerGPU, CFD Solver, LU Decomposition, Particle Filter, Path Finder, Gaussian Elimination, k-Nearest Neighbors, LavaMD, SQLite Select, 3D Stencil, Hybrid Sort, Myocyte, B+ Tree, Breadth-first Search (BFS), Distance-Cutoff Coulombic Potential (CUTCP), Fast Fourier Transform (FFT), Saturating Histogram (HISTO), Lattice-Boltzmann Method Fluid Dynamics (LBM), Dense Matrix-Matrix Multiply (MM), Magnetic Resonance Imaging - Gridding (MRI-GRIDDING), Magnetic Resonance Imaging - Q (MRI-Q), Sum of Absolute Differences (SAD), Spare-Matrix Dense-Vector Multiplication (SPMV), 3-D Stencil Operation (STENCIL), Two Point Angular Correlation Function (TRACF), SpeechRec, FaceRec, RayTrace, MPGenc, MPGdec, Dense Matrix Multiply, Transpose, Dense LU factorization with partial pivoting, QR Decomposition, Matrix tridiagonalization, 3D FFT, Multigrid Kernel, Embararassingly parallel, Conjugate gradient, Large integer sort, Input / output, PSTSWM Compact Application, NAS Parallel Benchmark CFD Codes, Sequence Similarity Searching, Phylogenetic Analysis, Multiple Sequence Alignment, Sequence Profile Searching, Genome-level Alignment, Sequence Assembly, GeneNet, Single Nucleotide Polymorphisms (SNP), SEMPHY, Support Vector Machines (SVM), Parallel Linear Space Alignment (PLSA), Sobel Edge Detection, Nqueens, Pi via Monte Carlo, Distributed Version of Lloyd’s Least squares quantization in PCM, Kmeans, Black Scholes, 3D Finite Differences, Dense Matrix Multiply (sgemm from BLAS), montecarlo, sobelconvolution, mandelbrot, kirchhoff, matrix vector, sparse matrix vector, fft, Histogram backprojection
Algorithms that occurred again and again and thus seem to be important
These Algorithms belong to the following domains:
So, what are the important aspects of those algorithms?

- Domain
- Algorithm Class

- Algorithm
  - Input Data
  - Output Data
  - Transformation
  - Atomics / Barriers / Synchronization Techniques?
  - Data Types / Precision

- Diversity, Parallelization and Speedup
- Computation vs. Communication

- Distinct Computing Bottleneck?
What are the (computationally) demanding problems / applications of the future?

Berkeley Dwarfs

Based on „The Parallel Comptuing Landscape: A View from Berkeley 2.0“
The Parallel Computing Landscape: A View from Berkeley 2.0

Krste Asanovic, Ras Bodik, Jim Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowick, Edward Lee, Nelson Morgan, George Necula, Dave Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Kathy Yelick
4 Themes of View = Par Lab

1. Applications
   - Compelling Apps driving the research agenda top-down

2. Identify Computational Bottlenecks
   - Breaking through disciplinary boundaries

3. Developing Parallel Software with Productivity, Efficiency, and Correctness
   - 2 Layers + C&C Language, Autotuning

4. OS and Architecture
   - Composable Primitives, not Packaged Solutions
   - Deconstruction, Fast Barrier Synchronization, Partition
Par Lab Research Overview

- Our industry has bet its future on parallelism (!)
  - Recruit best minds to help?
- Try Apps-Driven vs. CS Solution-Driven Research
- Dwarfs as lingua franca, anti-benchmarks, ...
- Efficiency layer for $\approx 10\%$ today’s programmers
- Productivity layer for $\approx 90\%$ today’s programmers
- C&C language to help compose and coordinate
- Autotuners vs. Parallelizing Compilers
- OS & HW: Composable Primitives vs. Solutions

Easy to write correct programs that run efficiently on manycore

- Apps
  - Personal Health
  - Image Retrieval
  - Hearing, Music
  - Speech
  - Parallel Browser
- Composition & Coordination Language (C&CL)
- C&CL Compiler/Interpreter
- Parallel Libraries
- Parallel Frameworks

Efficiency
- Efficiency Languages
- Legacy Code
- Schedulers
- Communication & Synch. Primitives

Arch.
- OS
  - Legacy OS
  - OS Libraries & Services
  - Hypervisor
- Multicore/GPGPU
- RAMP Manycore

Static Verification
- Type Systems
- Directed Testing
- Dynamic Checking
- Debugging with Replay

Correctness
1. Applications

■ „Who needs 100 cores to run MS Word?“
  □ We need compelling apps that use 100s of cores

■ Applications:
  □ Conorary Artery Disease Modelling + Health Coach
  □ Content-Based Image Retrieval + Meeting Diarist
  □ Face Recognizer / Name Whisperer
  □ Teleconference speaker identifier, speech helper
  □ Music Enhancer + Hearing Augmenter + New Instrument UI
  □ Parallel Browser for Handhelds

■ Applications cover the most important
  □ Platforms: Handheld, Laptop, Games
  □ Markets: Consumer, Business, Health
2. Computational Bottleneck instead of conventional benchmark

- How to invent parallel systems of the future when we are tied to old code, programming models, and CPUs of the past?

- Look for common patterns of communication and computation
  - Embedded Computing (42 EEMBC benchmarks)
  - Desktop / Server Computing (28 SPEC2006)
  - Data Base / Text Mining Software
  - Games / Graphics / Vision
  - Machine Learning
  - High Performance Computing (Original „7 Dwarfs“)

- Result: 13 Dwarfs
Dwarf Popularity
= How compelling apps relate to dwarfs
Berkeley Dwarfs

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

http://view.eecs.berkeley.edu/wiki/Dwarf_Mine
1. Dense Linear Algebra

- Classic vector and matrix operations: VxV, MxV, MxM

- Example:

```latex
\begin{verbatim}
do i=1,n
  do j=1,n
    do k=1,n
      a(i,j) = a(i,j) + b(i,k)*c(k,j)
    enddo
  enddo
enddo
\end{verbatim}
```

- Data layout: continues array
- Computation: on elements, rows, columns or matrix blocks
- Uniprocessor Mapping: block algorithms to exploit cache
- Parallel Mapping:
  - Issues: memory hierarchy, data distribution for load balancing critical
  - Best: 2d block cyclic distributions + comp./comm. overlap
Matrix Multiplication 3

```csharp
int ITILE2 = 32;
int JTILE2 = 32;

Parallel.For(ExecuteOn, 0, sizeX, delegate(int ii)
{
    for (int jj = 0; jj < sizeY; jj += JTILE2)
    {
        int il = Math.Min(ii * ITILE2 + ITILE2, sizeX);
        int jl = Math.Min(jj + JTILE2, sizeY);
        for (int i = (ii * ITILE2); i < il; i++)
            for (int j = jj; j < jl; j++)
                { c[i, j] = 0;
                    for (int k = 0; k < sizeZ; k++)
                        c[i, j] += a[i, k] * b[k, j];
                }
    }
});
```
2. Sparse Linear Algebra

- **Matrix with lots of zeros**
- **Compressed data structures**: only non-zero entries + indices

**Example:**

```plaintext
do i=1,n
   do j=row_start(i),row_start(i+1)-1
      y(i) = y(i) + val(j)*x(col_index(j))
   enddo
endo
```

- **Uniprocessor Mapping**: graph algorithms
  - integer operations + indexed accesses
  - dense blocks to exploit caches
- **Parallel Mapping**:
  - **Issues**: complex dependence structure
  - **Best**: Scatter-gather vector architecture HW
3. Spectral Methods

- **Data**: is operated on in the spectral domain (often transformed from temporal or spatial domain).
- During a transformation, spectral methods typically use multiple stages, where the dependencies within a stage form butterfly patterns (2 Inputs, 2 Outputs).
- Example: Fast Fourier Transform

- **Uniprocessor Mapping**: 
  - Highly vectorizable using strided accessing
  - Multiple simultaneous runs
  - Exploit scatter-gather

- **Parallel Mapping**: 
  - **Best**: A multidimensional transform is partitioned across multiple processors → A number of 1D spectral transforms are performed locally, according data partitioning and transposing.
4. N-Body Methods

- Calculations on interactions between many discrete points

- Variations:
  - Particle-particle $O(N^2)$,
  - Hierarchical: Barnes-Hutt $O(N \log N)$, Fast Multipole $O(N)$
  - Particle-in-cell codes belong to 5th Dwarf: Structured Grid

- **Uniprocessor Mapping:**
  - cache-blocking
  - divide-and-conquer (many points as one)

- **Parallel Mapping:**
  - **Issues:** load balancing, no fix hierarchy for moving particles
5. Structured Grid

- **Data**: a regular multidimensional grid; access is regular and statically determinable (strided)
- **Computation**: sequence of grid updates (all points are updated using values from a small neighborhood); updates are logically concurrent
- In practice implemented as sequential sweep through computation domain (in place or two grid versions)

- **Uniprocessor Mapping**: highly vectorizable, points can be visited in any order
  - Spatial locality to use of long cache lines
  - Temporal locality to allow cache reuse (small grids)

- **Parallel Mapping**: subgrid per processor
  - communication and synchronization for boundaries (=ghost cells, surface to volume ratio important)
  - Latency hiding: increased number of ghost zones + exchanging more data less frequently
5. Structured Grid
Variant: Adaptive Mesh Refinement

- Overlaying higher-resolution grids across areas of interest
- Probably recursive and adaptive over time

**Challenges:**
- Complex indexing
- Dynamic load balancing to distribute work evenly while minimizing communication

**Problems:**
- Reduced spatial locality
- Overhead to interpolate between coarse and fine grids
6. Unstructured Grid

- Elements update neighbors in irregular mesh/grid
  - Static or dynamic structure
- Problematic data distribution (usually indirection through tables) and access requirements
- **Domain**: modeling (think physics engine)
  - Mesh represents surface or volume
  - Entities are points, edges, faces, volumes, ...
  - Applying tension, temperature, pressure
  - Computations involve numerical solutions or differential equations
- **Computations**: sequence of mesh update steps
  - Like structured grid, but neighbors could be points or edges or faces or volumes ...
  - Poor spatial locality. Example:
6. Unstructured Grid

- Trend towards finer meshes and smaller time-steps
  -> larger datasets and more computations
- Massively data parallel, but irregularly distributed data and comm.

**Solutions**

- Data Distribution: graph; edges = nearest-neighbors
  - Partitioning to minimize number of cross-partition edges
    = communication-minimizing data distribution
  - Equivalent sub-meshes (= number of grid points) per processor
- Mesh granularity: maintain mesh stability for highly dynamic
  computations using an Adaptive Mesh Refinement (AMR) scheme
  - Very complex -> high development and verification cost
  - Complicates data distribution and load balancing problem

**Example:** design of airplanes (very accurate at wing-tips)
6. Unstructured Grid
Uniprocessor Mapping

**Locality**

- Highly vectorizable, but scatter-gather memory access required
- Memory hierarchies (and caching) are less effective

- *Spatial locality* is limited to one of the table entry types, as the others are accessed indirectly depending on the mesh
- Some *temporal locality*, since values are used by neighbors

- Reducing *memory traffic*
  - Increasing locality = Placing neighboring points nearby in memory: Graph partitioners (like Metis and Zoltan) or other reordering algorithms (like space-filling curves)
  - Reduce metadata overhead by encoding the data structure
6. Unstructured Grid Uniprocessor Mapping

**Prefetching**

- Indirection makes *compile-time analysis* difficult
  - Pattern is difficult to encode more efficiently than the index structures themselves

- Although memory access patterns repeat, each round is too large for *hardware pre-fetching*

- **Software pre-fetch** mechanisms to execute the index stream
  - Address resolution typically not data-dependent for static calculations
  - The order in which the data items are visited is static within a given iteration
  - Due to limited memory bandwidth, prefetch operations (or “speculative thread”) must lead the computations significantly
  - In a conventional cache-hierarchy such mechanism can still suffer from undesired cache-line evictions due to cache-line aliasing
Parallel Mapping

- Divide mesh structure -> neighboring entities on same node
- Communication/synchronization of nodes mostly only to neighbors (neighborhood is less regular than in structured grid)
- Graph partitioners
  - Equal mesh portions for each processor
  - Reducing the number of edges that cross between processors

Best:
- Allocate ghost regions around each processors portion of the graph -> fill with data from neighboring nodes before computation step
- Mesh partitioning as a preprocessing step
- Maybe additional mesh partitioners that operate incrementally in response to load imbalances at runtime
7. MapReduce
7. MapReduce

- Originally called “Monte Carlo”
- Repeated independent execution of a function with results aggregated at the end
- Nearly no communication between processes

- **Uniprocessor Mapping:**
  - pattern of computation depends on problem being run

- **Parallel Mapping:**
  - Embarrassingly parallel: independent, no communication
  - Issue: may require different amounts of computation

**Examples:**
- Monte Carlo Pi, Quantum Monte Carlo
- Optimization Protein Structure Prediction
- BOINC
8. Combinational Logic

- AND, OR, XOR, ...
- Exploit bit-level parallelism for high throughput
- Simple operations on very large amounts of data

Examples:
- Networks and file systems: checksums, RAID
- Data mining: population count, finding the number of ‘1’s in a word

Uniprocessor Mapping:
- Issues: lack of support for bit-level operations or variable-word-size operations in the ISA or the programming language

Parallel Mapping:
- Best: algorithms may be broken into data pipelines, where each processor executes part of the pipeline and then passes the data to the next processor
9. Graph Traversal

- Traverse a number of objects and examine their characteristics
- Usually indirect lookups and little computations
- Variation: searching

**Uniprocessor Mapping:**
- Pointer chasing without much chance for more efficient processing
- There may be locality in accesses to the graph
- There may be some processing per node that can reduce the effective cost of finding later nodes

**Parallel Mapping:**
- Issue: many searching algorithms limited by memory latency only
10. Dynamic Programming

- Compute solutions by solving simpler overlapping subproblems
- Optimization problems where the optimal result is built up of the optimal solutions of the subproblems
- Variations of subproblems:
  - Number of subproblems used in optimal solution to original problem
  - Number of choices in determining which subproblem to use
- Usually: table -> no entry = recomputed / entry = reuse

- Examples: circuit design, DNA sequence matching, finding most likely sequence of hidden states (with the Viterbi algorithm), Knapsack, ...

- **Uniprocessor Mapping**: lookup table
  - Issue: table lookup slow due to memory hierarchies
  - Structure grid: regular data access pattern -> reordering for better spatial and temporal locality (caching). Very hard for dynamic grids
10. Dynamic Programming

Parallel Mapping:

- Issue: table lookup slow due to communication delays
- Overlap of subproblems = data dependency
  - Communication vs. (re)computation overhead
  - Number of subproblems determines communication needs
  - Number of choices in a problem and optimization complexity determines computation needs

- Structured Problems:
  - Subproblems grouped into blocks to increase comp. granularity
  - Assumption: no communication delay within the same PE

- Unstructured Problems:
  - Dependency tree + ghost zones
  - Communication vs. computation trade-off is hard to balance
11. Backtrack and Branch-and-Bound

- Various search and global optimization problems
- Very large search space -> rule out some regions (this is the strength of this approach)
- Divide and Conquer principle
  -> branching into subdivisions
  + bounds are found on everything in subregion

- *Examples:* Integer Linear Programming, Boolean Satisfiability, Combinatorial Optimization; Traveling Salesman

**Uniprocessor Mapping:** explore search space

- Problem-specific heuristics to guide search to productive regions
- Problem-specific bounding methods at each node of the search

**Parallel Mapping:** subregion for each processor

- Issue: useful invariants about the search space: duplicated calculations vs. inter-process communication overhead
- Issue: dynamic load balancing (load prediction impossible; see picture above)
12. Graphical Models

- A graph in which nodes represent variables, and edges represent conditional probabilities
- Bayesian networks, Hidden Markov models, neural networks
- Examples: AI, machine learning, speech and image recognition

**Uniprocessor Mapping:**
- Evaluation is a form of Graph Traversal (9th dwarf)
- Probabilistic aspect -> small amount of computation per node
- Processing many observations and updating variables accordingly

**Parallel Mapping:**
- May be evaluated multiple times for a single problem
  - Update conflicts possible
- Simple: many graphical models can be evaluated for a single input
13. Finite State Machines

- Interconnected set of states, initial state, input, transitions (based on current inputs and state), output (based on current inputs and state)

- Parallelism in the computation of state transitions

- Decomposing into multiple state machines possible
  - Smaller and simpler
  - Combined states and outputs functionally mimic the original
  - Communication/synchronization required

- **Uniprocessor Mapping**: usually case statements

- **Parallel Mapping**:  
  - Issue: only one active state = only one thread of execution + parallelism must justify communication overhead
  - Best: decomposition, multiple FSM at the same time
Valuable Roles of Dwarfs

1. „Anti-benchmark“
   - Dwarfs are not tied to code or language artifacts
   - Encourage innovation in algorithms, languages, data structures, and/or hardware

2. Universal, understandable vocabulary
   - To talk across disciplinary boundaries

3. Bootstrapping: Parallelize parallel research
   - Allow analysis of HW & SW design without waiting years for full apps

4. Targets for libraries
The 13-11 Benchmark
The 13-11 Benchmark

- Matrix Multiplication
- Quadratic Sieve
- Othello

MPI

SMP + CUDA

Global Master

Node 1

Node 2

Node N-1

Node N

Machine 1

Machine 2

Machine k

SMP Threads

CUDA Threads

SMP Threads

CUDA Threads
OpenDwarfs

- A non-commercial OpenCL compute benchmark suite
- Performance data for CPUs, MICs, NVIDIA and AMD GPUs
- Other runtimes are planned for the future

- https://github.com/opendwarfs

- Stable:
  - gem

- Beta:
  - bfs, cfd, crc, fft, kmeans, lud, nw, spmv, srad, swat, bwa_hmm, nqueens

- Alpha:
  - astar, tdm
What are the (computationally) demanding problems / applications of the future?

Data Access Patterns
Dwarf Popularity
= How compelling apps relate to dwarfs
<table>
<thead>
<tr>
<th>Best Practices in Performance Tuning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm Design</strong></td>
</tr>
<tr>
<td>- Asynchronous, Recompute, Simple</td>
</tr>
<tr>
<td><strong>Memory Transfer</strong></td>
</tr>
<tr>
<td>- Chaining, Overlap Transfer &amp; Compute</td>
</tr>
<tr>
<td><strong>Control Flow</strong></td>
</tr>
<tr>
<td>- Divergent Branching, Predication</td>
</tr>
<tr>
<td><strong>Memory Types</strong></td>
</tr>
<tr>
<td>- Local Memory as Cache, rare resource</td>
</tr>
<tr>
<td><strong>Memory Access</strong></td>
</tr>
<tr>
<td>- Coalescing, Bank Conflicts</td>
</tr>
<tr>
<td><strong>Sizing</strong></td>
</tr>
<tr>
<td>- Execution Size, Evaluation</td>
</tr>
<tr>
<td><strong>Instructions</strong></td>
</tr>
<tr>
<td>- Shifting, Fused Multiply, Vector Types</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
</tr>
<tr>
<td>- Native Math Functions, Build Options</td>
</tr>
</tbody>
</table>
1. Dense Linear Algebra

- Classic vector and matrix operations: $V \times V$, $M \times V$, $M \times M$

- Example:

```fortran
  do i=1,n
    do j=1,n
      do k=1,n
        a(i,j) = a(i,j) + b(i,k)*c(k,j)
      enddo
    enddo
  enddo
```

- Data layout: continues array
- Computation: on elements, rows, columns or matrix blocks
- Uniprocessor Mapping: block algorithms to exploit cache
- Parallel Mapping:
  - Issues: memory hierarchy, data distribution for load balancing critical
  - Best: 2d block cyclic distributions + comp./comm. overlap
Matrix Multiplication 1

```csharp
Parallel.For(ExecuteOn, 0, sizeX, 0, sizeY, delegate(int i, int j)
{
    c[i, j] = 0;
    for (int k = 0; k < sizeZ; k++)
        c[i, j] += a[i, k] * b[k, j];
});
```
Matrix Multiplication 2

```csharp
Parallel.For(ExecuteOn, 0, sizeX, 0, sizeY, delegate(int i, int j)
{
    double v = 0;

    for (int k = 0; k < sizeZ; k++)
        v += a[i, k] * b[k, j];

    c[i, j] = v;
});
```
Matrix Multiplication 3

```csharp
int ITILE2 = 32;
int JTILE2 = 32;

Parallel.For(ExecuteOn, 0, sizeX, delegate(int ii)
{
    for (int jj = 0; jj < sizeY; jj += JTILE2)
    {
        int il = Math.Min((ii * ITILE2) + ITILE2, sizeX);
        int jl = Math.Min(jj + JTILE2, sizeY);
        for (int i = (ii * ITILE2); i < il; i++)
            for (int j = jj; j < jl; j++)
            {
                c[i, j] = 0;
                for (int k = 0; k < sizeZ; k++)
                    c[i, j] += a[i, k] * b[k, j];
            }
    }
});
```
```csharp
int m = sizeX; int n = sizeY; int p = sizeZ;
int ITILE3 = 32; int JTILE3 = 32; int KTILE3 = 32;

Parallel.For(ExecuteOn, 0, (m + ITILE3 - 1) / ITILE3,
                   0, (n + JTILE3 - 1) / JTILE3, delegate(int idx_ii, int idx jj)

                   {
                       int ii = idx_ii * ITILE3;
                       int il = Math.Min(ii + ITILE3, m);

                       int jj = idx_jj * JTILE3;
                       int jl = Math.Min(jj + JTILE3, n);

                       for (int kk = 0; kk < p; kk += KTILE3) {
                           int kl = Math.Min(kk + KTILE3, p);

                           for (int i = ii; i < il; i++)
                               for (int j = jj; j < jl; j++)
                                   for (int k = kk; k < kl; k++)
                                       c[i, j] += a[i, k] * b[k, j];
                       }
                   });
```
Matrix Multiplication 5

matmultrec(sizeX, sizeY, sizeZ, 0, sizeX, 0, sizeY, 0, sizeZ, a, b, c);

void matmultrec(int m, int n, int p, int mf, int ml, int nf, int nl, int pf, int pl,
                double[,] A, double[,] B, double[,] C)
{
    if ((ml - mf) * (nl - nf) * (pl - pf) < 8 * 32768) /* product size below which matmultleaf is used */
        matmultleaf(m, n, p, mf, ml, nf, nl, pf, pl, A, B, C);
    else
    {
        matmultrec(m, n, p, mf, mf + (ml - mf) / 2, nf, nf + (nl - nf) / 2, pf, pf + (pl - pf) / 2, A, B, C);
        matmultrec(m, n, p, mf, mf + (ml - mf) / 2, nf, nf + (nl - nf) / 2, nl, pf, pf + (pl - pf) / 2, A, B, C);
        matmultrec(m, n, p, mf, mf + (ml - mf) / 2, nf, nf + (nl - nf) / 2, nl, pf, pf + (pl - pf) / 2, pl, A, B, C);
        matmultrec(m, n, p, mf, mf + (ml - mf) / 2, nl, pf, pf + (pl - pf) / 2, A, B, C);
        matmultrec(m, n, p, mf + (ml - mf) / 2, ml, nf, nf + (nl - nf) / 2, pf + (pl - pf) / 2, A, B, C);
        matmultrec(m, n, p, mf + (ml - mf) / 2, ml, nf, nf + (nl - nf) / 2, pl, A, B, C);
        matmultrec(m, n, p, mf + (ml - mf) / 2, ml, nf, nf + (nl - nf) / 2, pl, A, B, C);
        matmultrec(m, n, p, mf + (ml - mf) / 2, ml, nf + (nl - nf) / 2, pf + (pl - pf) / 2, A, B, C);
        matmultrec(m, n, p, mf + (ml - mf) / 2, ml, nf + (nl - nf) / 2, pl, A, B, C);
    }
}

void matmultleaf(int m, int n, int p, int mf, int ml, int nf, int nl, int pf, int pl,
                 double[,] A, double[,] B, double[,] C)
{
    Parallel.For(ExecuteOn, mf, ml, nf, nl, delegate(int i, int j)
    {
        for (int k = pf; k < pl; k++)
            C[i, j] += A[i, k] * B[k, j];
    });
}
Data Access Patterns

Access to Neighbors:
None, One, Two-Dimensional

Access to Neighbors: Simple vs. Ring

Strided Access: 1, 2, 4, 8

[Pattern], [Access], [Frequency], [Stride], [Neighbors], [Borders], [Data Type]

map

Constant  Texture  Local

Global

Host

Private

[Pattern], [Access], [Frequency], [Stride], [Neighbors], [Borders], [Data Type]
Data Access Patterns

Workflow
1. Programmers declare (mark and describe) data access patterns in code
2. Automatic mapping onto hybrid hardware
3. Apply auto optimizations:
   - Reduce copy overhead while maintaining correctness

Algorithm Characteristics
- **Pattern**: linear, complex, arbitrary
- **Access**: read only, write only, read write
- **Frequency**: never, once, exactly N times, rarely, frequent
- **Stride**: vector of non-negative integers; default 1
- **Neighbors**: vector of non-negative integers; default 0
- **Borders**: ignore, ring fashion, next dimension, repeat last, fix value
- **Data Type**
Examples of Data Access Patterns

<table>
<thead>
<tr>
<th>Example</th>
<th>Pattern</th>
<th>Access</th>
<th>Frequency</th>
<th>Stride</th>
<th>Neighbors</th>
<th>Borders</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot Product</td>
<td>Linear</td>
<td>Read Only, Write Only</td>
<td>Once</td>
<td>Huge</td>
<td>Many</td>
<td>Ignore</td>
</tr>
<tr>
<td>Average</td>
<td>Linear</td>
<td>Read Only, Write Only</td>
<td>Exactly 3 Times, Once</td>
<td>0</td>
<td>2 (1D)</td>
<td>Ring Fashion</td>
</tr>
<tr>
<td>Summing Vectors</td>
<td>Linear</td>
<td>Read Only, Write Only</td>
<td>Once</td>
<td>0</td>
<td>0</td>
<td>Ignore</td>
</tr>
<tr>
<td>Ripple</td>
<td>Linear</td>
<td>Write Only</td>
<td>Never, Once</td>
<td>0</td>
<td>0</td>
<td>Ignore</td>
</tr>
<tr>
<td>Heat Transfer</td>
<td>Linear</td>
<td>Read Only, Write Only</td>
<td>Exactly 5 Times</td>
<td>0</td>
<td>4 (2D)</td>
<td>Ignore</td>
</tr>
<tr>
<td>Histogram</td>
<td>Linear, Arbitrary</td>
<td>Read Only, Write Only</td>
<td>Once, Frequent</td>
<td>0</td>
<td>0</td>
<td>Ignore</td>
</tr>
</tbody>
</table>

The diagrams illustrate the process of data access, where read and write operations are performed before and after calculation.
Data Access Pattern Transformations

Filter

Rotate

Partition
Flow of Transforms

Filter → Partition → Rotate → Read

CPU → Read → MIC

GPU → Read → MIC
What are the (computationally) demanding applications of the future?

- Many domains
- Many applications
- Many algorithms

- Same bottlenecks again and again

- Coarse-grain view: Berkeley Dwarfs
  - Encourage innovation in algorithms, languages, data structures, and/or hardware; target for libraries

- Fine-grained view: Data Access Patterns
  - Enable innovation in algorithms, languages, data structures and/or hardware; target for libraries and compilers.