NTNU – Trondheim
Norwegian University of
Science and Technology

Application types

Jan.Christian.Meyer@ntnu.no

Recent history

- Parallel computing went mainstream in 2003
  - A broad panel of researchers held a series of meetings from 2004 through 2006, to figure out what to do about it
- Their final report deservedly received a lot of attention

  ***"The Landscape of Parallel Computing Research: A View from Berkeley"*, **

  K. Asanovic *et al.*, Dec. 2006

  https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf
- We have actually been repeating several of its points throughout the semester

  They have been commonly accepted since their publication

NTNU – Trondheim
Norwegian University of Science and Technology

My favorite part

- The whole report covers *a lot* of ground
- In my opinion, one of the highlights is a section that pinpoints computational problems that occur frequently
  – We’ve solved the advection equation in the auditorium
  – You’ve solved the diffusion equation as homework
  – Hopefully, you’ve noticed that they belong to a class of programs that have many technical challenges in common
- We have a number of matching problem classes, with their own performance characteristics
  – It’s impossible to make an exhaustive and eternally complete list, but it’s very useful to have an approximate attempt

The seven dwarves of Berkeley

- The Berkeley panel arrived at 7 different computation-classes they felt were most important:
  - Dense linear algebra
  - Sparse linear algebra
  - Structured grids
  - Unstructured grids
  - N-body problems
  - Monte Carlo methods
  - Spectral methods
- It's "seven" because that's how many there are (and it evokes associations with a fairy tale that is easy to remember)
- It's "dwarves" because they refer to miniature versions of larger problems

Simplified representations

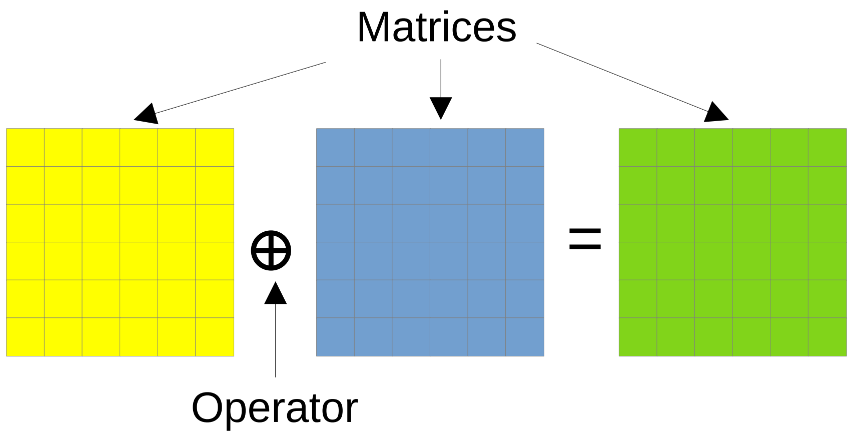
(*aka.* “proxy applications”, “mini-apps”, “kernels”...)

- No useful program *only* does one of those seven things
- When a program does *one* of them, however, that is probably going to be the most time-consuming thing it does
- If it does *several* of them, it will perform differently while it is working on each of its stages

- The Big Idea:
  - If we can parallelize an example of *e.g.* dwarf #3 on computer X, we will know that similar applications can be adapted equally well for computer X
  - When you’re evaluating whether or not to invest in a new machine, it is <u>much</u> faster/easier to try out a performance proxy than to rewrite a full application

Dense linear algebra

- We've touched upon this one, in the matrix multiplication examples
- Key ingredient:
  - Two or more matrices/vectors full of numbers that are mostly different from zero
  - Some kind of operator you want to combine them with



NTNU – Trondheim
Norwegian University of Science and Technology

Dense linear algebra

Characteristics

- This type of computation packs long arrays of consecutive values tightly together in memory
- The operator often consists of somewhat complicated calculations that require many instructions per element

Consequently,

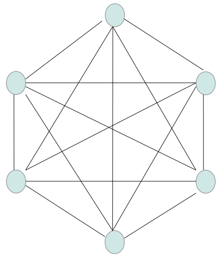
- Cache utilization is of great importance to speed
- Data structures are 1D and 2D contiguous arrays

Dense linear algebra

Origins

```
111111
111111
111111
111111
111111
111111
```

Adjacency

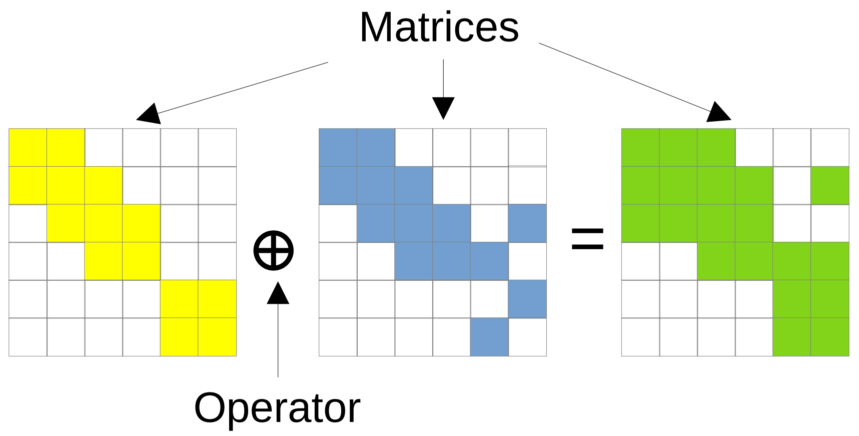


Graph

- When you solve A*x* = *b* for a dense matrix A and vectors *x*, *b,* every value in *x* contributes to every value in *b*
- This sort of thing appears when the elements of x represent a set of things that all affect each other directly
- Some use cases
  - Quantum mechanics (physics of very small things)
  - Homogeneous systems of linear equations (Eigenvalue problems)
  - Data analytics (dimensionality reduction)

Sparse linear algebra

- This is the same kind of problem as its dense cousin, we're combining matrices and vectors
- The difference is that many/most of the matrix elements are equal to zero
- This makes it meaningless to read and write them

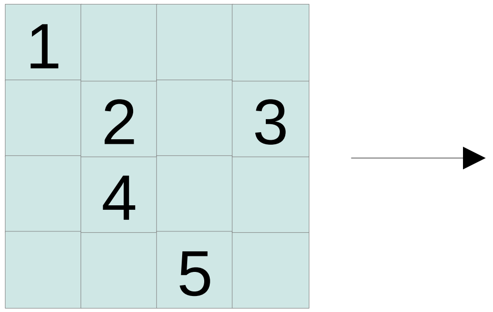


(The illustration isn't *very* sparse, but bigger matrices tend to bring out the effect)

Sparse linear algebra

Data structures

- Since we only need a small subset of the (i,j) indices in each matrix, the data structure turns into a list of indices with non-zero values instead of reserving a memory location per element
  - Simple, popular format: CSR (Compressed Sparse Row)



Values:	[1,2,3,4,5]	The elements
Rows:	[0,1,3,4]	Start of each row in the element list
Cols:	[0,1,3,1,2]	Col. of each number in the element list

(Swap the roles of rows/cols to obtain CSC format)

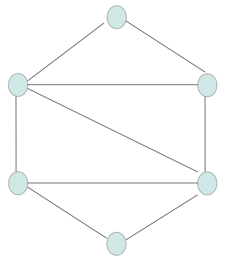
Sparse linear algebra

Characteristics

- Two levels of indirection for every access
  - You have to look up indices in lists before you can get to the element that they index
  - The memory access pattern is semi-unpredictable
    - In general, it depends on patterns found in arbitrary matrices
    - If you know something about the patterns in your matrices before you start, you can customize the indexing mechanism for them
- Operations tend to have less exploitable work per element
  - Applications often become memory-bound

```
111000
111110
111010
010111
011111
000111
```

Adjacency



Graph

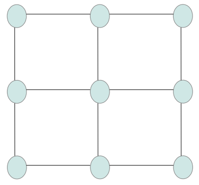
Sparse linear algebra

Origins

- When you solve A*x* = *b* for a sparse matrix A and vectors *x*, *b,* only a few values in *x* contribute to a few values in *b*
- We get this if we split the geometry of a physical thing into sub-things where near neighbors affect each other, and remote parts don’t
- Some use cases:
  - Fluid dynamics, mechanical engineering, climate simulations, … (physics of everyday-size things)
  - Inhomogeneous systems of linear equations (implicit time-integration)
  - Search engines, social networks, machine learning...

Structured grids

```
110100000
111010000
011101000
101110100
010111010
001011101
000101110
000010111
000001011
Adjacency
```

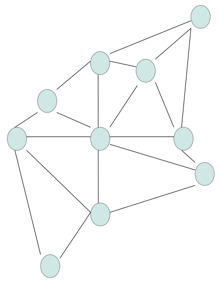

Graph

- This is what we've been doing with the advection equation, and in the problem sets
  - Divide the problem into equally-sized and equally-shaped pieces
  - Near neighbor points affect each other
- Data structures become 2D, 3D, 4D, … arrays
  - You can look at this as a special case of sparse lin. alg.
  - The matrix pattern becomes so regular that we don't even have to represent the matrix in memory, and just connect neighbor elements directly in the code
  - Application areas are similar/same as with sparse lin. alg.
  - Performance characteristics are also similar (mostly memory-bound problems)
  - Main difference: no indirect indexing, so we avoid the extra lookup cost

Unstructured grids

<matrix redacted>
BUT
typically, same number of entries in every row

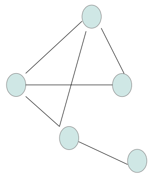
Adjacency



Graph

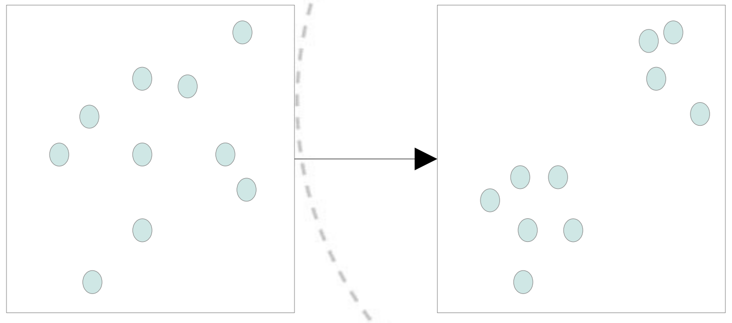
- These are like the structured grids, but without the assumption that neighbors are evenly spaced
  - If all the sub-parts have the same shape, however, it's still not necessary to explicitly represent any matrices in the code
- Data structures depend on the shape of elements
  - Using triangles, we get lists of points + lists of 2 neighbors / vertex
  - Using hexagons, we get lists of points + lists of 3 neighbors / vertex
  - ...and so on… data structures must be adapted to problem geometry, but they become regular

Point	Neighbors
1	2,3
2	1,4
3	1,2
4	1,5
...	...



...

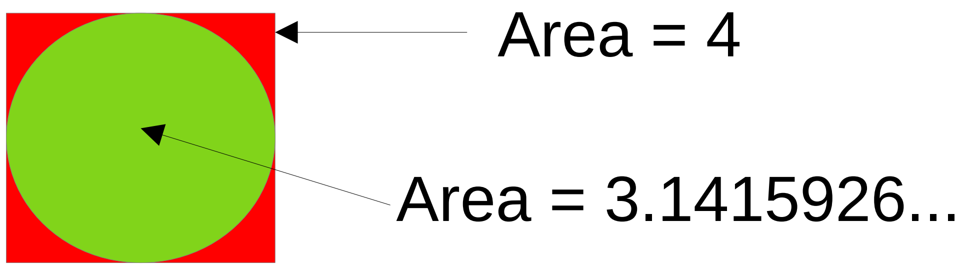
N-body problems



- Problems consist of a list of coordinates for things that push each other around
  - The bodies can be atoms, stars, planets, raindrops, billiard balls…
  - Their coordinates change frequently
- Bottleneck: finding neighbors
  - If every body can affect every other, we get N*(N-1) / 2 pairs
  - If only nearby bodies can affect each other we get a search problem instead, because their neighbor-relations change often
- The performance challenge:
  - Invent data structures that sort nearby bodies into nearby memory quickly (often using some details that come from what the bodies represent)

Monte Carlo methods

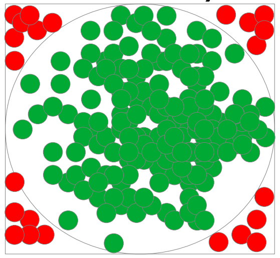
- Monte Carlo methods are calculations that approach their solution by accumulating random numbers
  - Since we’ve already been doing this masterclass in Pi estimation, we can make one more approximation :)
- Imagine a perfectly circular dartboard (radius 1), inscribed in a square (2x2):



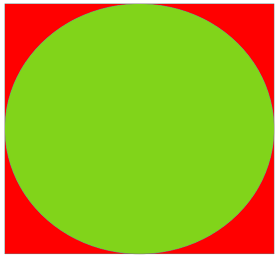
NTNU – Trondheim
Norwegian University of
Science and Technology

Monte Carlo methods

- Throw a bunch of darts at it, randomly:



- Some will hit and some will miss, but each additional point brings the ratio of hits to darts closer to Pi / 4:



- The trick is to formulate the problem in such a way that additional numbers contribute to the solution no matter what their values are

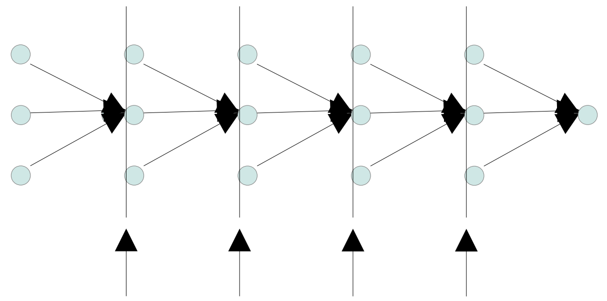
Monte Carlo methods

Characteristics

- MC methods are arguably *embarrasingly parallel*
  - The outcome of each dart-toss is independent of every other
  - *Hooray,* this will be super-parallel!
- Performance challenges:
  - Most pseudo-random number generators have a sequential dependence between one random number and the next
  - When pseudo-random isn't good enough, *true* random number generators rely on harvesting noise from some slow, physical process
  - Even when individual samples are independent, accumulating statistics introduces a need for shared/locked locations to store the overall statistic in

Spectral methods

- Most popularly, these employ Fast Fourier Transforms (FFTs)
  - A couple of other transforms are available, notably Laplace and Wavelet
- We don't have time to discuss these in detail
- We can do a (very) simplified summary from a parallel computing point of view, though
- Our familiar difference/volume/element methods develop values in neighbor points through combining them incrementally:

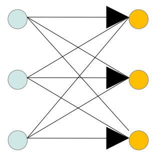


Synchronization (barriers)

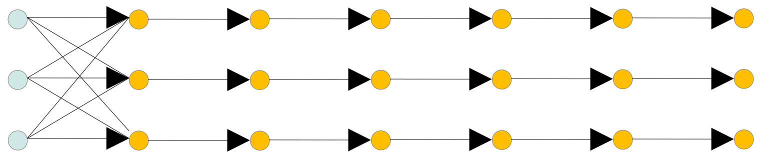
NTNU – Trondheim
Norwegian University of
Science and Technology

Spectral methods

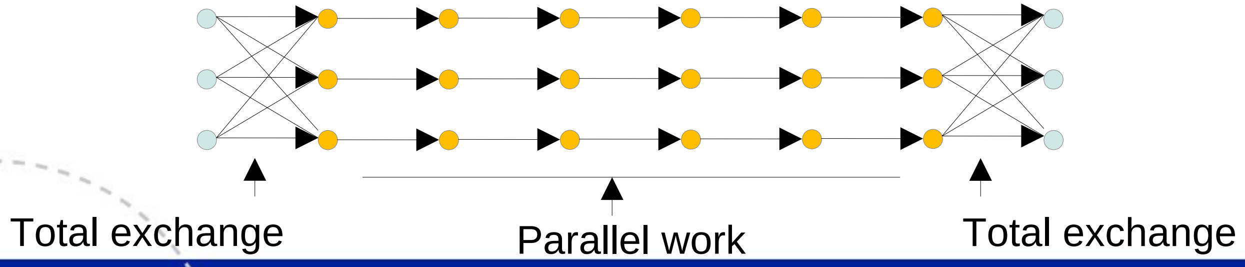
- Transform calculations start by obtaining point-wise functions after looking at the state of the whole problem



- These functions can be developed independently



- Finally, you need the whole problem again in order to reverse the transformation



NTNU – Trondheim
Norwegian University of Science and Technology

In summary

- Those were some super-quick walkthroughs of the original Berkeley kernels
  - Hopefully, enough to give you an idea about how each presents different challenges to effective parallel solutions
- Since 2006, people have come up with many more classes & representative problems
  - These are enough to start on an overview, though
  - I’m mostly trying to make the argument that it’s valuable to look for familiar patterns in parallel programs that do completely different things