CS 294-73

Software Engineering for Scientific Computing

http://www.eecs.berkeley.edu/~colella/CS294Fall2017/
colella@eecs.berkeley.edu
pcolella@lbl.gov

Lecture 1: Introduction

Grading

- 5-6 homework assignments, adding up to 60% of the grade.
- The final project is worth 40% of the grade.
  - Project will be a scientific program, preferably in an area related to your research interests or thesis topic.
  - Novel architectures and technologies are not encouraged (they will need to run on a standard Mac OS X or Linux workstation)

Hardware/Software Requirements

- Laptop or desktop computer on which you have root permission
- Mac OS X or Linux operating system
  - Cygwin or MinGW on Windows *might* work, but we have limited experience there to help you.
- Installed software
  - gcc (4.7 or later) or clang
  - GNU Make
  - gdb or lldb
  - ssh
  - gnuplot
  - VisIt
  - Doxygen
  - emacs
  - LaTex

Homework and Project submission

- Submission will be done via the class source code repository (git).
- On midnight of the deadline date the homework submission directory is made read-only.
- We will be setting up times for you to get accounts.

What we are not going to teach you in class

- Navigating and using Unix
- Unix commands you will want to know
  - ssh
  - scp
  - tar
  - gzip/gunzip
  - ls
  - mkdir
  - chmod
  - ln
- Emphasis in class lectures will be explaining what is really going on, not syntax issues. We will rely heavily on online reference material, available at the class website.
- Students with no prior experience with C/C++ are **strongly** urged to take CS9F.

What is Scientific Computing ?

We will be mainly interested in scientific computing as it arises in simulation.

The scientific computing ecosystem:

- A science or engineering problem that requires simulation.
- Models – must be mathematically well posed.
- Discretizations – replacing continuous variables by a finite number of discrete variables.
- Software – correctness, performance.
- Data – inputs, outputs. Science discoveries ! Engineering designs !
- Hardware.
- People.

What will you learn from taking this course ?

The skills and tools to allow you to understand (and perform) good software design for scientific computing.

- Programming: expressiveness, performance, scalability to large software systems (otherwise, you could do just fine in matlab).
- Data structures and algorithms as they arise in scientific applications.
- Tools for organizing a large software development effort (build tools, source code control).
- Debugging and data analysis tools.

Why C++ ?

- Strong typing + compilation. Catch large class of errors at compile time, rather than run time.
- Strong scoping rules. Encapsulation, modularity.
- Abstraction, orthogonalization. Use of libraries and layered design.

C++, Java, some dialects of Fortran support these techniques to various degrees well. The trick is doing so without sacrificing performance. In this course, we will use C++.

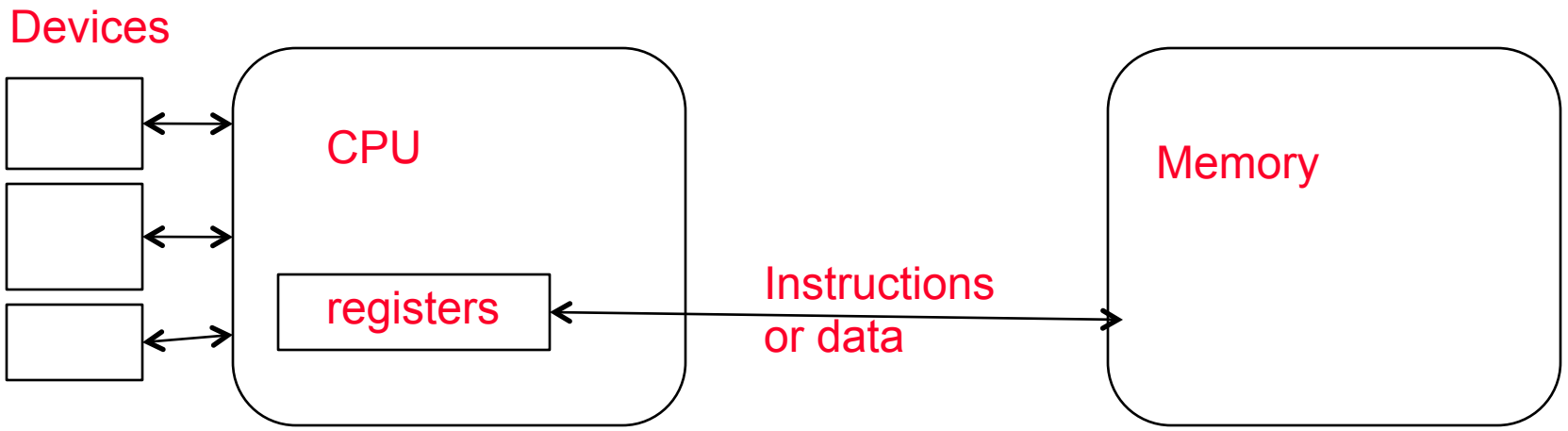
- Strongly typed language with a mature compiler technology.
- Powerful abstraction mechanisms.

A Cartoon View of Hardware

What is a performance model ?

- A “faithful cartoon” of how source code gets executed.
- Languages / compilers / run-time systems that allow you to implement based on that cartoon.
- Tools to measure performance in terms of the cartoon, and close the feedback loop.

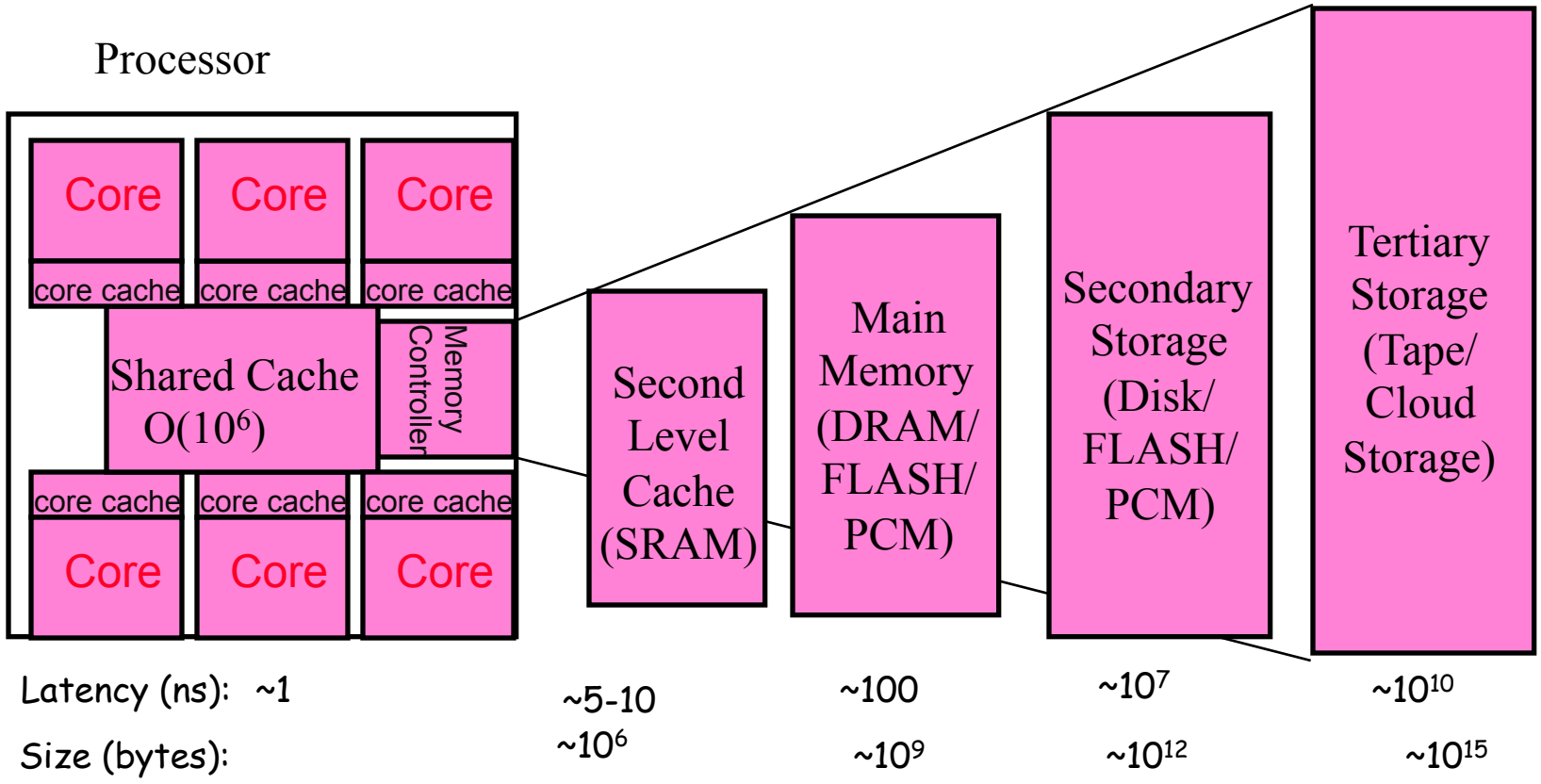
The Von Neumann Architecture



- Data and instructions are equivalent in terms of the memory. Up to the processor to interpret the context.

Memory Hierarchy

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



The Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
    - so, keep a copy of recently read memory in cache.
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
    - Guess where the next memory reference is going to be based on your access history.
- Processors have relatively lots of bandwidth to memory, but also very high latency. Cache is a way to hide latency.
  - Lots of pins, but talking over the pins is slow.
  - DRAM is (relatively) cheap and slow. *Banking* gives you more bandwidth

Programs with locality cache well ...

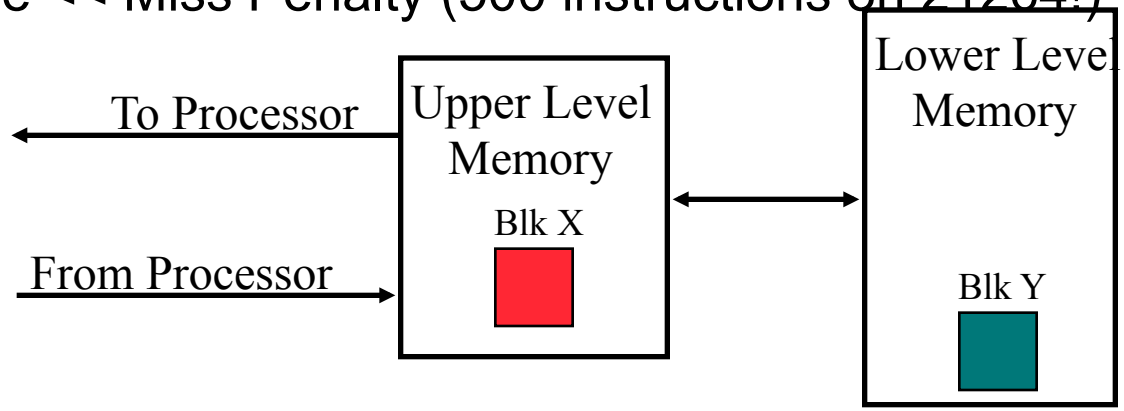
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Memory Hierarchy: Terminology

- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of

    RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level +

    Time to deliver the block the processor
- Hit Time << Miss Penalty (500 instructions on 21264!)



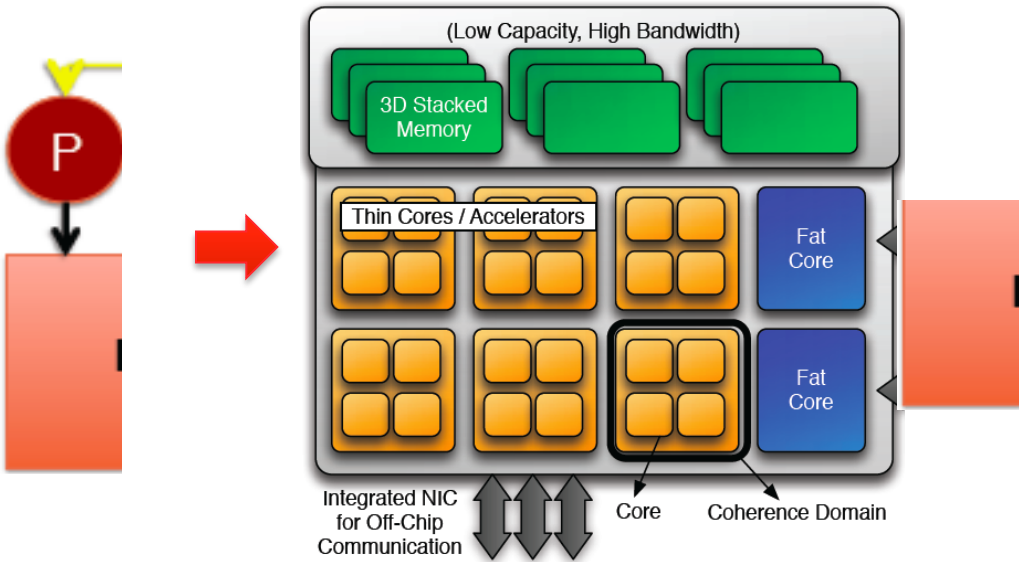
Consequences for programming

- A common way to exploit spatial locality is to try to get stride-1 memory access
  - cache fetches a cache-line worth of memory on each cache miss
  - cache-line can be 32-512 bytes (or more)
- Each cache miss causes an access to the next deeper memory hierarchy
  - Processor usually will sit idle while this is happening
  - When that cache-line arrives some existing data in your cache will be ejected (which can result in a subsequent memory access resulting in another cache miss. When this event happens with high frequency it is called cache thrashing).
- Caches are designed to work best for programs where data access has lots of simple locality.

But processor architectures are changing…

- SIMD (vector) instructions: a(i) = b(i) + c(i), i = 1, … , 4 is as fast as a0 = b0 + c0)
- Non-uniform memory access
- Many processing elements with varying performance

  I will have someone give a guest lecture on this during the semester. Otherwise, not our problem (but it will be in CS 267).



Take a peek at your own computer

- Most UNIX machines
  - >cat /etc/proc
- Mac
  - >sysctl -a hw

Seven Motifs of Scientific Computing

Simulation in the physical sciences and engineering is done out using various combinations of the following core algorithms.

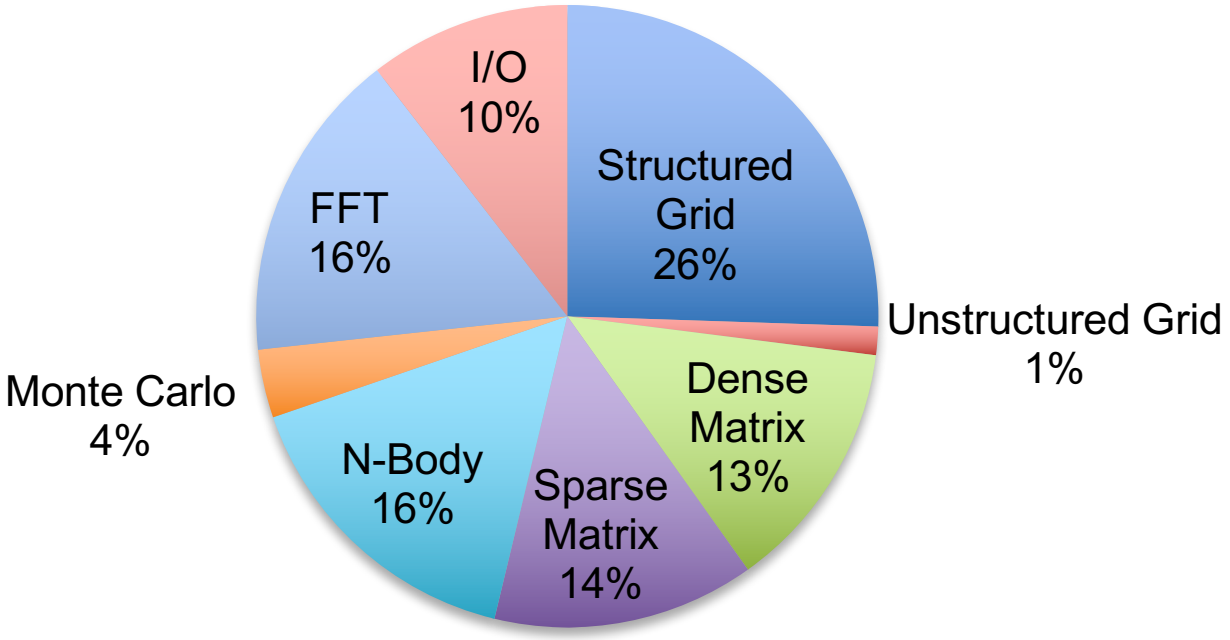
- Structured grids
- Unstructured grids
- Dense linear algebra
- Sparse linear algebra
- Fast Fourier transforms
- Particles
- Monte Carlo (We won't be doing this one)

Each of these has its own distinctive combination of computation and data access.

There is a corresponding list for data (with significant overlap).

Seven Motifs of Scientific Computing

- Blue Waters usage patterns, in terms of motifs.



A "Big-O, Little-o" Notation

If $f = f(x)$, $g = g(x)$ real-valued functions, we say that, if x is near x_0

- $f = O(g)$ as $x \to x_0$ if $|f(x)| \leq C|g(x)|$ for C independent of x.
- $f = o(g)$ if $lim_{x \to x_0} \frac{|f(x)|}{|g(x)|} = 0$.

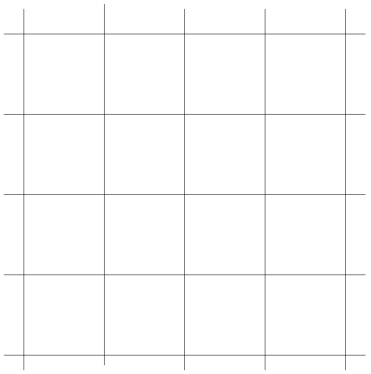
Examples:

- $sin(\frac{1}{n}) = O(\frac{1}{n})$ as $n \to \infty$
- $cos(\frac{1}{n}) = O(1)$ as $n \to \infty$
- $\int_1^n \frac{1}{x} dx = O(log(n))$

$f = \Theta(g)$ if $f = O(g)$, $g = O(f)$

Structured Grids

Used to represent continuously varying quantities in space in terms of values on a regular (usually rectangular) lattice.



$$\Phi = \Phi(\boldsymbol{x}) \rightarrow \phi_{\boldsymbol{i}} \approx \Phi(\boldsymbol{i}h)$$

$$\phi : B \rightarrow \mathbb{R} \ , \ B \subset \mathbb{Z}^D$$

If *B* is a rectangle, data is stored in a contiguous block of memory.

$$B = [1, \ldots, N_x] \times [1, \ldots, N_y]$$

$$\phi_{i,j} = chunk(i + (j-1)N_x)$$

Typical operations are stencil operations, e.g. to compute finite difference approximations to derivatives.

$$L(\phi)_{i,j} = \frac{1}{h^2}(\phi_{i,j+1} + \phi_{i,j-1} + \phi_{i+1,j} + \phi_{i-1,j} - 4\phi_{i,j})$$

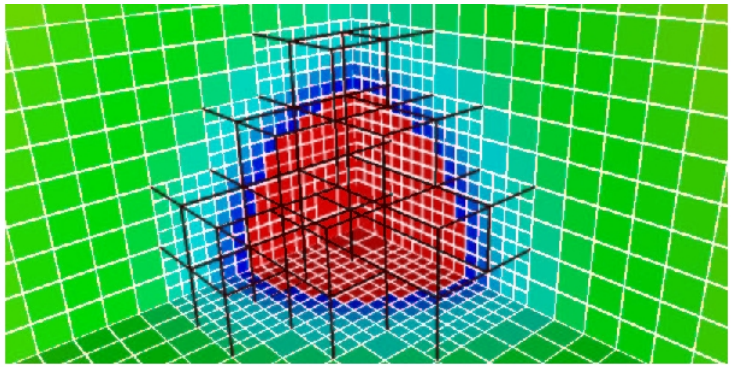
Small number of flops per memory access, mixture of unit stride and non-unit stride.

Structured Grids

In practice, things can get much more complicated: For example, if B is a union of rectangles, represented as a list.

$$\Phi = \Phi(\boldsymbol{x}) \rightarrow \phi_{\boldsymbol{i}} \approx \Phi(\boldsymbol{i}h)$$

$$\phi : B \rightarrow \mathbb{R} \ , \ B \subset \mathbb{Z}^D$$



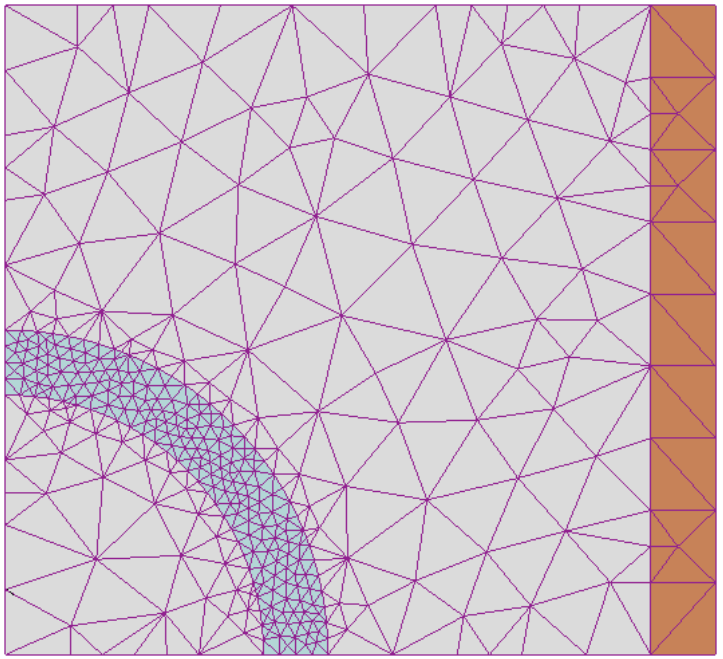
To apply stencil operations, need to get values from neighboring rectangles.

$$L(\phi)_{i,j} = \frac{1}{h^2}(\phi_{i,j+1} + \phi_{i,j-1} + \phi_{i+1,j} + \phi_{i-1,j} - 4\phi_{i,j})$$

Can also have a nested hierarchy of grids, which means that missing values must be interpolated.
Algorithmic / software issues: sorting, caching addressing information, minimizing costs of irregular computation.

Unstructured Grids

• Simplest case: triangular / tetrahedral elements, used to fit complex geometries. Grid is specified as a collection of nodes, organized into triangles.



$$\mathcal{N} = \{\boldsymbol{x}_n : n = 1, \ldots, N_{nodes}\}$$

$$\mathcal{E} = \{(\boldsymbol{x}^e_{n_1}, \ldots, \boldsymbol{x}^e_{n_{D+1}}) : e = 1, \ldots, N_{elts}\}$$

• Discrete values of the function to be represented are defined on nodes of the grid.

$$\Phi = \Phi(\boldsymbol{x}) \text{ is approximated by } \phi : \mathcal{N} \to \mathbb{R} \ , \ \phi_n \approx \Phi(\boldsymbol{x}_n)$$

• Other access patterns required to solve PDE problems, e.g. find all of the nodes that are connect to a node by an element. Algorithmic issues: sorting, graph traversal.

Dense Linear Algebra

Want to solve system of equations

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

Dense linear algebra

Gaussian elimination:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix} \longrightarrow \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ 0 & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ 0 & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & a_{n,2} & a_{n,3} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

$$a_{k,l} := a_{k,l} - a_{1,l}\frac{a_{k,1}}{a_{1,1}}$$

$$b_l := b_l - b_1\frac{a_{k,1}}{a_{1,1}}$$

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\ 0 & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\ 0 & 0 & a_{3,3} & \cdots & a_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & a_{n,3} & \cdots & a_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{pmatrix}$$

$$a_{k,l} := a_{k,l} - a_{2,l}\frac{a_{k,2}}{a_{2,2}}$$

$$b_l := b_l - b_2\frac{a_{k,2}}{a_{2,2}}$$

The p^{th} row reduction costs $2(n-p)^2 + O(n)$ flops, so that the total cost is

$$\sum_{p=1}^{n-1} 2(n-p)^2 + O(n^2) = O(n^3)$$

Good for performance: unit stride access, and O(n) flops per word of data accessed. *But,* if you have to write back to main memory...

Sparse Linear Algebra

$$\mathbf{A} = \begin{pmatrix} 1.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2.3 & 0 & 1.4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3.7 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1.6 & 0 & 2.3 & 9.9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5.8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7.4 & 0 & 0 \\ 0 & 0 & 1.9 & 0 & 0 & 0 & 4.9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3.6 \end{pmatrix}$$

Want to store only non-zeros, so use compressed storage format.

JA	1	2	4	3	2	4	5	5	6	3	7	8
StA	1.5	2.3	1.4	3.7	–1.6	2.3	9.9	5.8	7.4	1.9	4.9	3.6

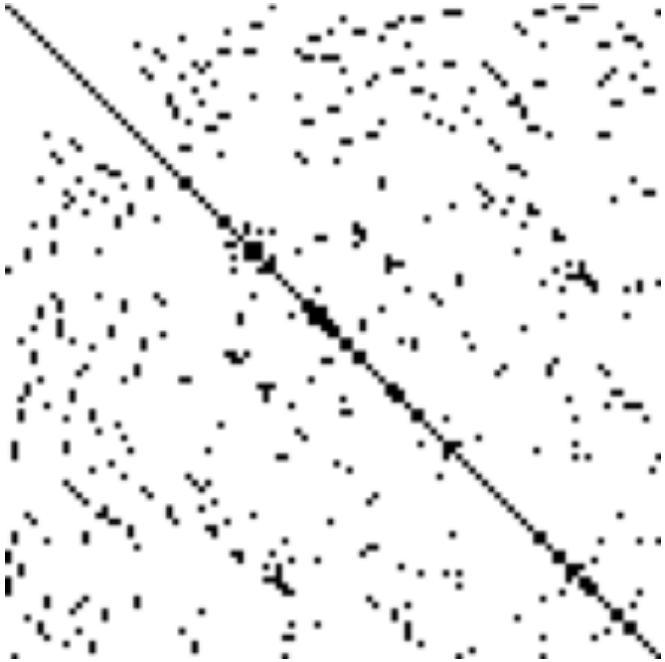
IA	1	2	4	5	8	9	10	12	13

Sparse Linear Algebra

• Matrix multiplication: indirect addressing. Not a good fit for cache hierarchies.

$$(A\boldsymbol{x})_k = \sum_{j=IA_k}^{IA_{k+1}-1} (StA)_j x_{JA_j} \ , \ k = 1, \ldots, 8$$

• Gaussian elimination: fills in any columm below a nonzero entry all the way to the diagonal. Can attempt to minimize this by reordering the variables.



• Iterative methods for sparse matrices are based on applying the matrix to the vector repeatedly. This avoids memory blowup from Gaussian elimination, but need to have a good approximate inverse to work well.

Fast Fourier Transform

$$\mathcal{F}_k^N(\boldsymbol{x}) \equiv \sum_{n=0}^{N-1} x_n z^{nk} \ , \ z = e^{-2\pi\iota/N} \ , \ k = 0, \ldots, N-1$$

$$= \sum_{\tilde{n}=0}^{N/2-1} x_{2\tilde{n}} z^{2\tilde{n}k} + z^k \sum_{\tilde{n}=0}^{N/2-1} x_{2\tilde{n}+1} z^{2\tilde{n}k}$$

$$= \mathcal{F}_k^{N/2}(\mathcal{E}(\boldsymbol{x})) + z^k \mathcal{F}_k^{N/2}(\mathcal{O}(\boldsymbol{x}))$$

We also have

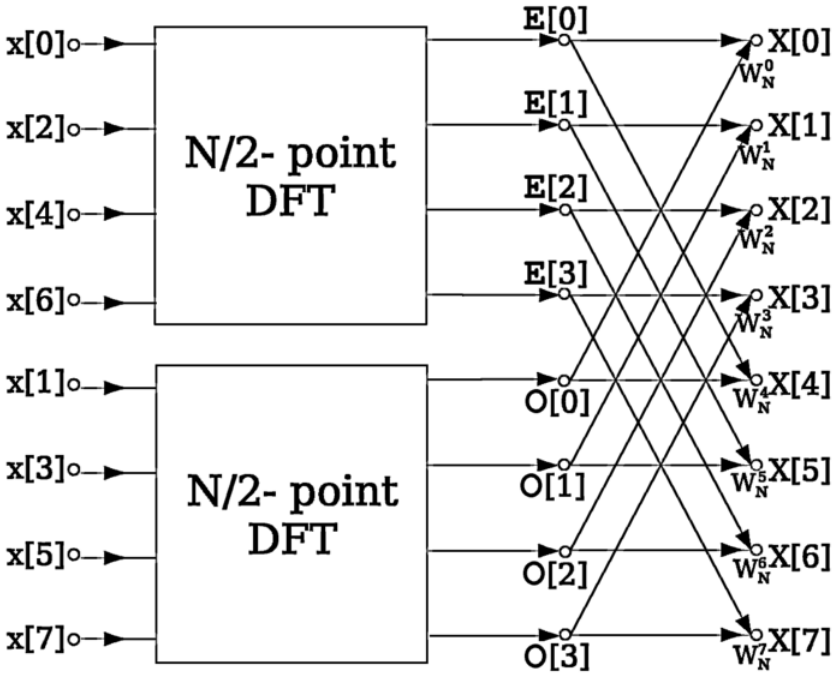
$$\mathcal{F}_k^P(\boldsymbol{x}) = \mathcal{F}_{k+P}^P(\boldsymbol{x})$$

So the number of flops to compute $\mathcal{F}^N(\boldsymbol{x})$ is 2 N, given that you have

$$\mathcal{F}^{N/2}(\mathcal{E}(\boldsymbol{x})) \ , \ \mathcal{F}^{N/2}(\mathcal{O}(\boldsymbol{x}))$$

Fast Fourier Transform

$$\mathcal{F}_k^{N/2}(\mathcal{E}(\boldsymbol{x})) + z^k \mathcal{F}_k^{N/2}(\mathcal{O}(\boldsymbol{x}))$$



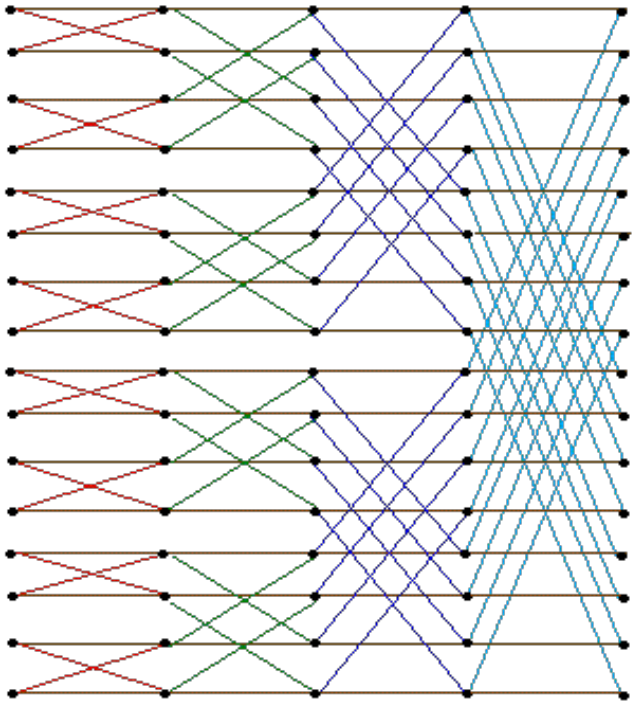
Fast Fourier Transform

If N = 2^M , we can apply this to $\mathcal{F}^{N/2}(\mathcal{E}(\boldsymbol{x}))$, $\mathcal{F}^{N/2}(\mathcal{O}(\boldsymbol{x}))$:

$$\mathcal{F}_k^{N/2}(\mathcal{E}(\boldsymbol{x})) = \mathcal{F}_k^{N/4}(\mathcal{E}(\mathcal{E}(\boldsymbol{x}))) + (z^2)^k \mathcal{F}_k^{N/4}(\mathcal{O}(\mathcal{E}(\boldsymbol{x})))$$

$$\mathcal{F}_k^{N/2}(\mathcal{O}(\boldsymbol{x})) = \mathcal{F}_k^{N/4}(\mathcal{E}(\mathcal{O}(\boldsymbol{x}))) + (z^2)^k \mathcal{F}_k^{N/4}(\mathcal{O}(\mathcal{O}(\boldsymbol{x})))$$

The number of flops to compute these smaller Fourier transforms is is also 2 x 2 x (N/2) = 2 N, given that you have the N/4 transforms. Can continue this process until computing 2^{M-1} sets of $\mathcal{F}^2$, each of which costs O(1) flops. So the total number of flops is O(M N) = O(N log N). The algorithm is recursive, and the data access pattern is complicated.



Particle Methods

Collection of particles, either representing physical particles, or a discretization of a continuous field.

$$\{\boldsymbol{x}_k, \boldsymbol{v}_k, w_k\}_{k=1}^{N}$$

$$\frac{d\boldsymbol{x}_k}{dt} = \boldsymbol{v}_k$$

$$\frac{d\boldsymbol{v}_k}{dt} = \boldsymbol{F}(\boldsymbol{x}_k)$$

$$\boldsymbol{F}(x) = \sum_{k'} w_{k'} (\nabla\Phi)(\boldsymbol{x} - \boldsymbol{x}_{k'})$$

To evaluate the force for a single particle requires N evaluations of $\nabla\Phi$, leading to an $O(N^2)$ cost per time step.

Particle Methods

To reduce the cost, need to localize the force calculation. For typical force laws arising in classical physics, there are two cases.

- Short-range forces (e.g. Lennard-Jones potential).

$$\Phi(\boldsymbol{x}) = \frac{C_1}{|\boldsymbol{x}|^6} - \frac{C_2}{|\boldsymbol{x}|^{12}}$$

$$\nabla\Phi(\boldsymbol{x}) \equiv 0 \text{ if } |\boldsymbol{x}| > \sigma$$

The forces fall off sufficiently rapidly that the approximation introduces acceptably small errors for practical values of the cutoff distance.

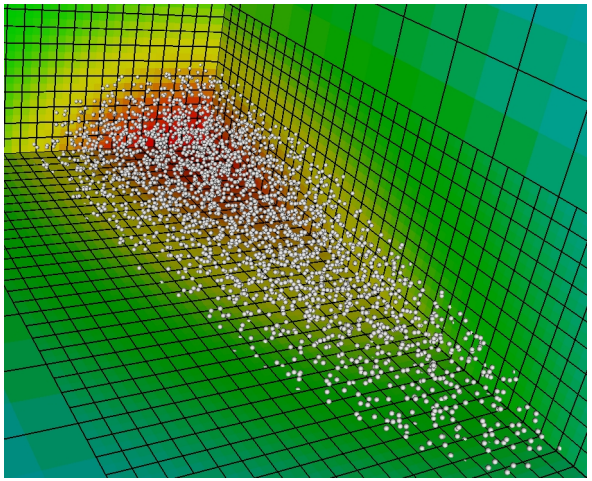
σ

Particle Methods

- Coulomb / Newtonian potentials

$$\Phi(\boldsymbol{x}) = \frac{1}{|\boldsymbol{x}|} \text{ in 3D}$$
$$= log(|\boldsymbol{x}|) \text{ in 2D}$$

cannot be localized by cutoffs without an unacceptable loss of accuracy. However, the far field of a given particle, while not small, is smooth, with rapidly decaying derivatives. Can take advantage of that in various ways. In both cases, it is necessary to sort the particles in space, and organize the calculation around which particles are nearby / far away.



Options: “Buy or Build?”

- “Buy”: use software developed and maintained by someone else.
- “Build”: write your own.
- Some problems are sufficiently well-characterized that there are bulletproof software packages freely available: LAPACK (dense linear algebra), FFTW. You still need to understand their properties, how to integrate it into your application.
- “Build” – but what do you use as a starting point ?
  - Programming everything from the ground up.
  - Use a framework that has some of the foundational components built and optimized.
- Unlike LAPACK and FFTW, frameworks typically are not “black boxes” – you will need to interact more deeply with them.

Fast Fourier Transform