Software Engineering for Scientific Computing

Phillip Colella

Elements of Scientific Simulation

We are mainly interested in scientific computing as it arises in simulation that requires either (1) large software systems, and (2) high-performance computing. Increasingly, (2) $\longrightarrow$ (1). However, want to remind you of the larger context.

- 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 are the Tools for HPC?

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

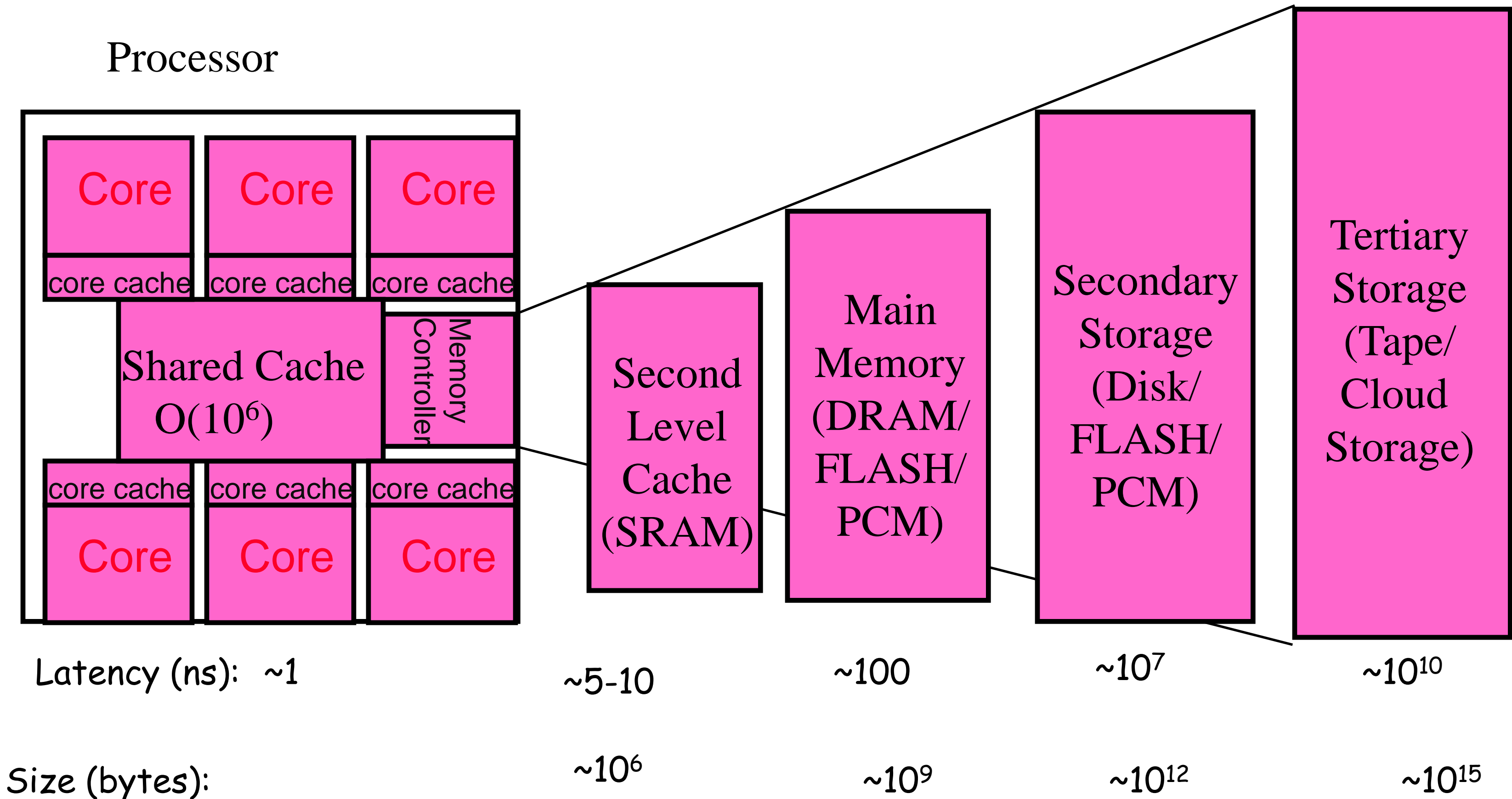
- Programming: expressiveness, performance, scalability to large software systems.
- Data structures and algorithms as they arise in scientific applications.
- Tools for organizing a large software development effort (build tools, source code control).
- Debugging and data analysis tools.

Outline of the Talk

- A little bit about hardware
- Motifs of scientific simulation.
- Programming and software design.
- A little bit about plumbing: source code control, build systems, debugging, data analysis and visualization.

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 cheap and slow. *Banking* gives you more bandwidth

Consequences for programming

- A common way to exploit spatial locality is to assume 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 soon)
- 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.
- All of this becomes more complicated as we introduce more processors on a chip.

Seven Motifs of Scientific Computing

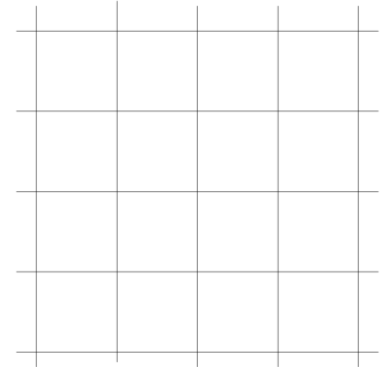
Simulation in the physical sciences is done out using various combinations of the following core algorithms.

- Structured grids
- Unstructured grids
- Dense linear algebra
- Sparse linear algebra
- Fast Fourier transforms
- Particles
- Monte Carlo

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

Structured Grids

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



$$\Phi = \Phi(x) \rightarrow \phi_i \approx \Phi(ih)$$

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

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.

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

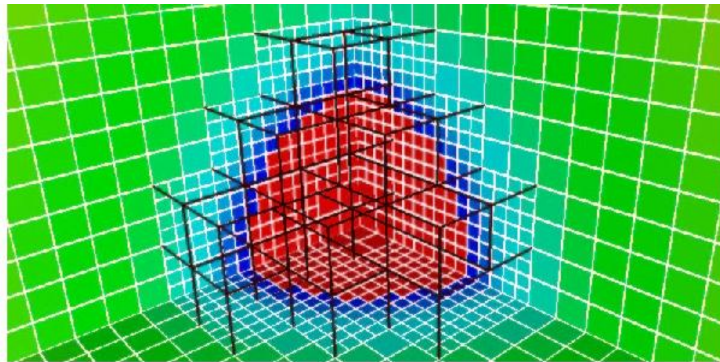
Small number of flops per memory access, mixture of unit stride and non-unit stride.

Structured Grids

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

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

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



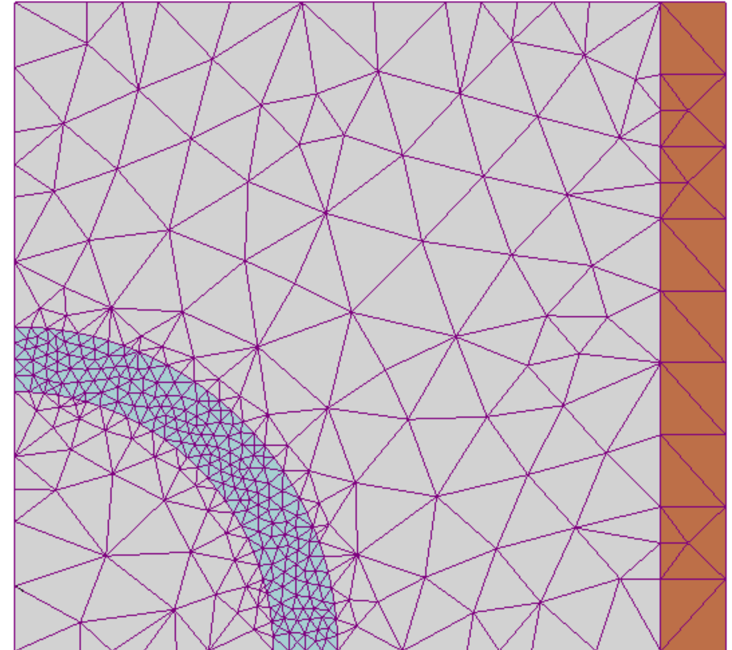
To apply stencil operations, need to get values from neighboring rectangles.

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

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.

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

• 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 pth row reduction costs 2 $(n-p)^2$ + O(n) flops, so that the total cost is

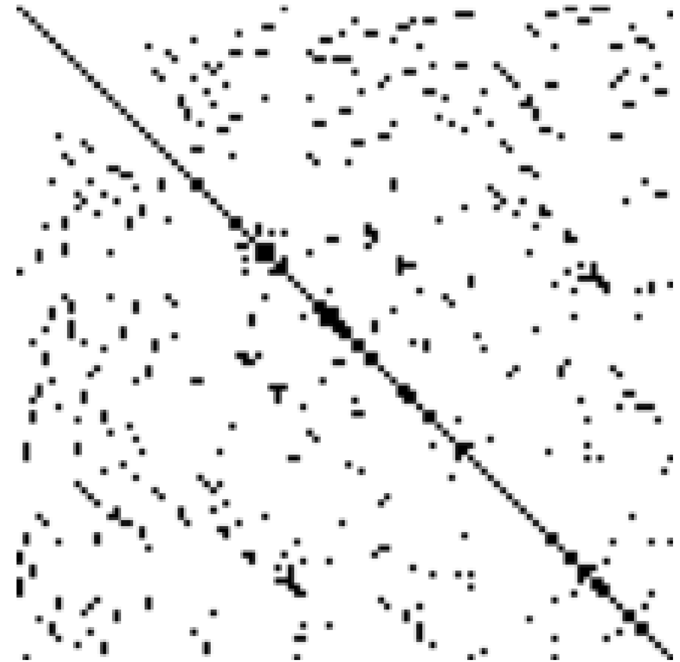
$$\sum_{p=1}^{n-1} 2(n-p)^2 + O(n^2) = O(n^3)$$

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

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

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



• 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.
• Diagonal scaling leads to modified compressed representation.

Fast Fourier Transform

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

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

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

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

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

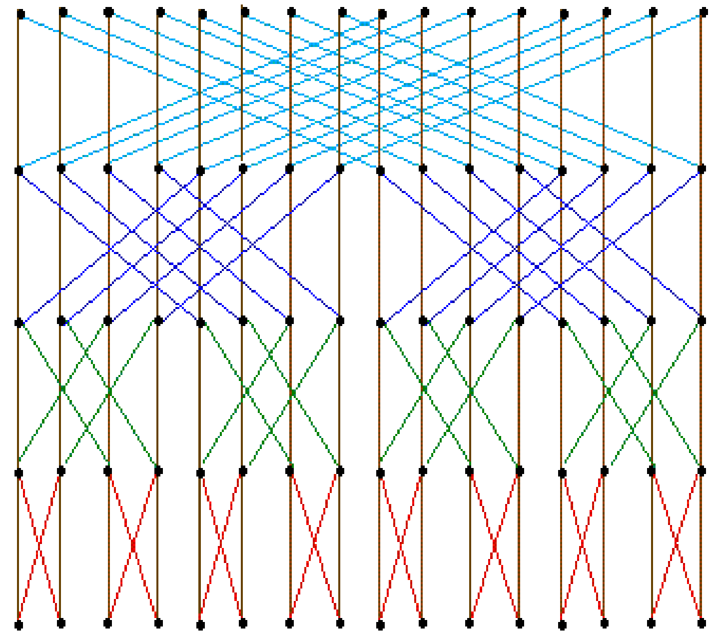
Fast Fourier Transform

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

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

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

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. Can be simplified by sorting (reverse bit sort), leading to an algorithm that can be done as a loop.



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. This can be reduced to $O(N \log N)$ for continuous fields, or to $O(N^2/p)$ for discrete particles by various localization techniques.

Software Design for Large Projects

Competing Concerns:

- Performance.
- Expressiveness - how easy / difficult is it to express what you want the computer to do for you.
- Maintainability.
    - Debugging.
    - Modification to do something different.
    - Ability for the other programmers on the team to pick up where you left off.
    - Porting to new platforms.

Tools and Techniques

- 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 on the motifs given above. In the discussion here, we will focus on C++.

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

The built-in types

- int, float, double, char
- This seems pretty meager. In fact, if that's all you had you would not be writing programs in C.
  - Matlab at least has Matrix and Vector
- Adding some elements to this makes a big difference
  - Array
  - Pointer
  - Functions
  - Looping
  - Conditionals
- Given those you can write very advanced programs. Your operating system, for instance.
- But we're doing scientific computing, where is matrix and vector ?

Making your own types

- But not that much richer. What about a 3D array ? Fancier tensors ? What if you are not doing linear algebra…but perhaps working on one of the other motifs.
- C++ lets you build *User-Defined Types*. They are referred to as a Class. Classes are a combination of member data and member functions. A variable of this type is often referred to as an object.
- Classes provide a mechanism for controlling scope.
  - Private data / functions: There are internal representations for objects that users usually need not concern themselves with, and that if they changed you probably wouldn't use the type any differently.
  - Public data / functions: There are the functions and operations that manipulate this state and present an abstraction for a user.
  - Protected data / functions: intermediate between the first two, but closer to private than to public.

Cut-and-Paste Coding Avoided Using Templates

- Usually you find bugs long after you have cut and pasted the same bug into several source files
- The once uniform interfaces start to diverge and users become frustrated
- Awesome new functions don't get added to all versions of your class
- you end up with a LOT of code you have to maintain, document, and test.

Vector as a Template Class

```
template <class T> class Vector
{
public:
  ~Vector();
  explicit Vector(int a_dim);
  Vector(const Vector<T>& a_rhs);
  Vector<T> operator*(T a_x) const;
  T  operator*(const Vector<T>& a_rhs) const;
  T& operator[](int a_index);
  Vector<T> operator+(Vector<T> a_rhs);
   void push_back(cost T& val);
private:
  T *m_data;
  int m_size;
};
```

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 we use compressed storage format.

- Represent internally as a pair of vector of vectors.

  - $V_I[j]$, $j=0,n_{row}-1$ $V_I[j][m]$ is the column index of $(m+1)^{st}$ nonzero entry in row $j+1$.
  - $V_R[j]$, $j=0,n_{row}-1$ $V_I[j][m]$ is the value of A at $(j+1, V_I[j])$.

$V_I[1] = \{2,3\}$
$V_R[1] = \{2.3,1.4\}$

SparseMatrix Class

```
class SparseMatrix
{
public:
  /// set up an M rows and N columns sparse matrix
  SparseMatrix(int a_M, int a_N);
  /// Matrix Vector multiply.  a_v.size()==N, returns vector of size M
  Vector<float> operator*(const Vector<float>& a_v) const;
  ///accessor functions for get and set operations of matrix elements
  float& operator[](int a_index[2]);
private:
  int m_m, m_n;
  float m_zero;
  Vector<Vector<float> > m_data;
  Vector<Vector<int> >   m_colIndex;
};
```

If necessary, sparse matrix automatically adds a new matrix element when you reference that location, and initializes it to zero.

For each non-zero entry in ‘A’ we keep one float, and one int indicating which column it is in

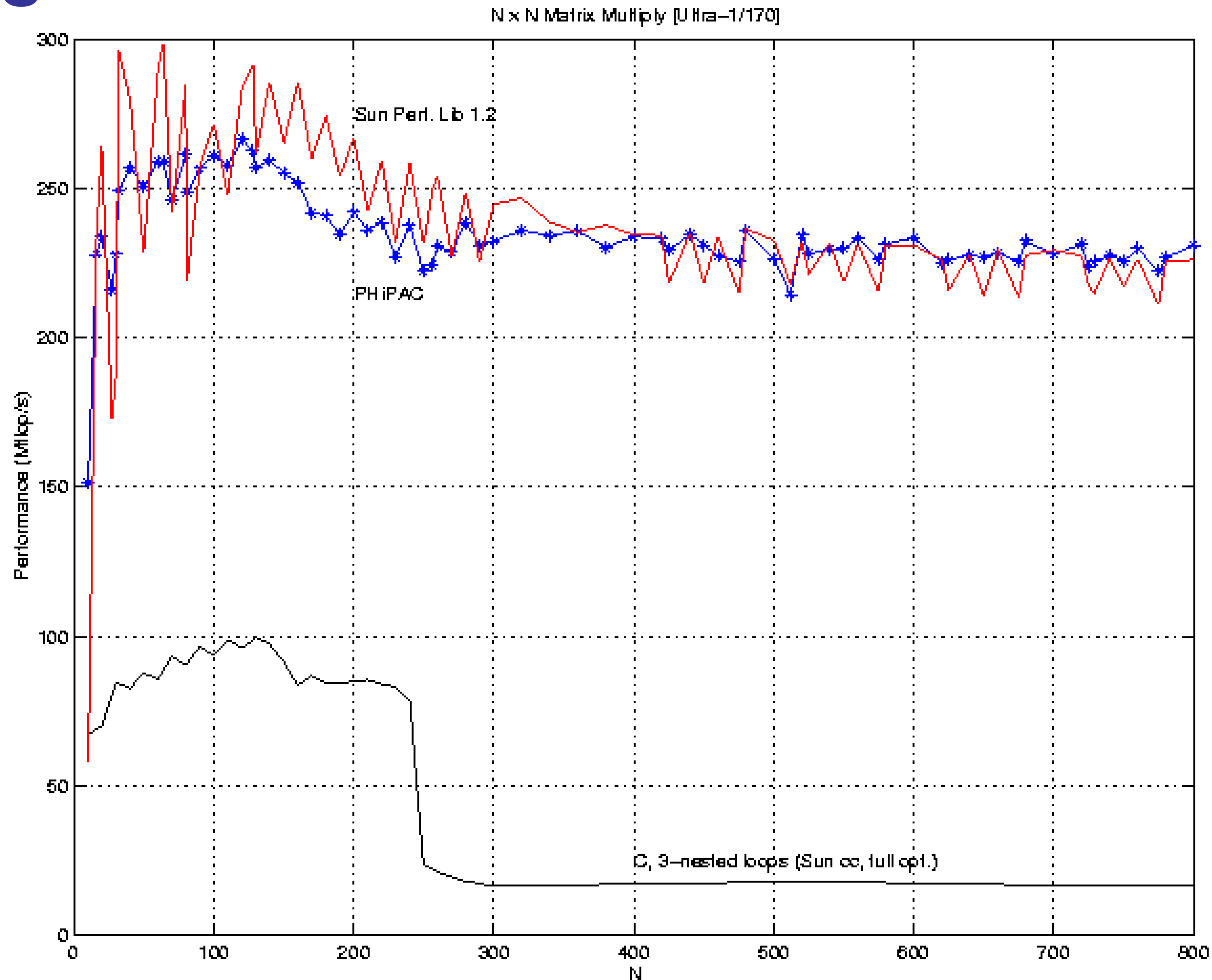
Using a SparseMatrix object

```
SparseMatrix A(cells-2, cells);
int left[2]   = {0,0};
int center[2] = {0,1};
int right[2]  = {0,2};
for(int i=0; i<cells-2; i++, left[0]++, left[1]++, center[0]++,
  center[1]++,right[0]++, right[1]++)
  {
    A[left]=1;
    A[center]=-2;
    A[right]=1;
  }
vector<float> LOfPhi = A*phi;
```

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.

Matrix-multiply, optimized several ways



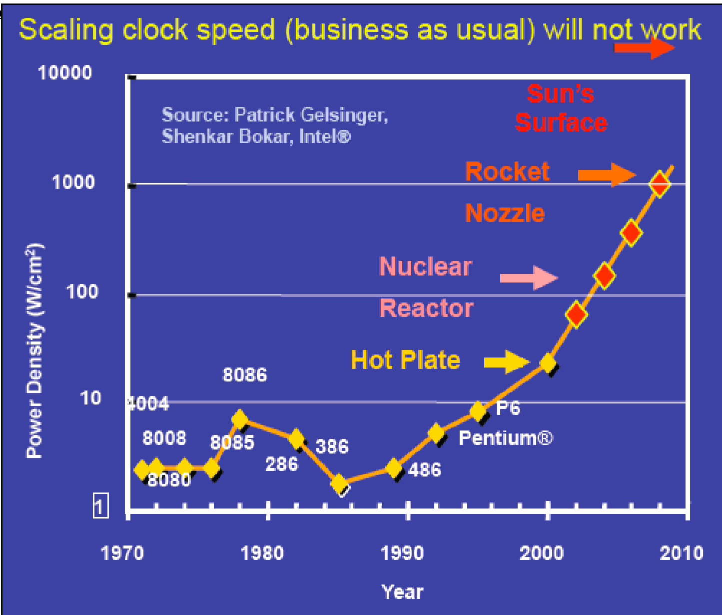
Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

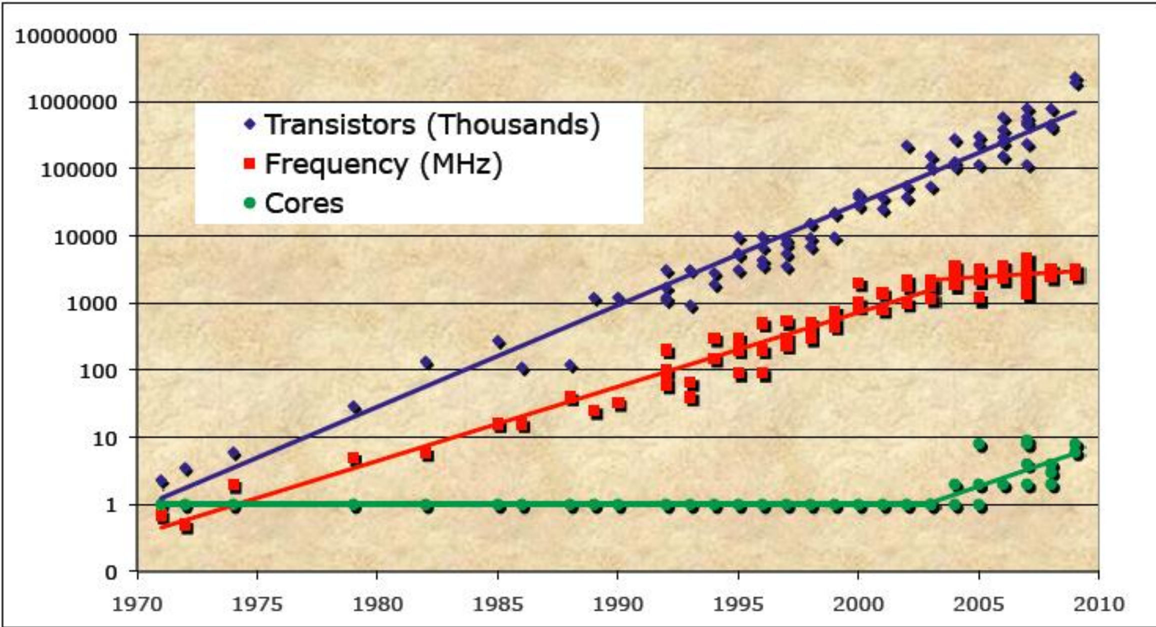
Plumbing

- Build Systems (make/Gmake, config)
- Revision Control (cvs, svn, git,…)
- Debugging tools (gcc + gdb + emacs is the lowest common denominator)
- Visualization and data analysis tools (VisIt, ParaView)

What's Next ?

More Parallelism





NRC study:"The Future of Computer Performance: Game Over or Next Level?"

The power cost of memory

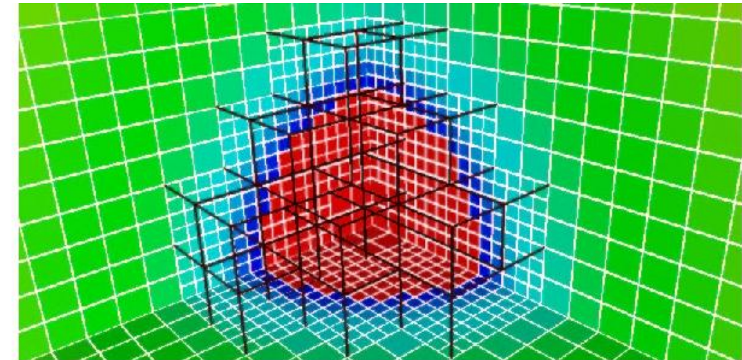
mJ

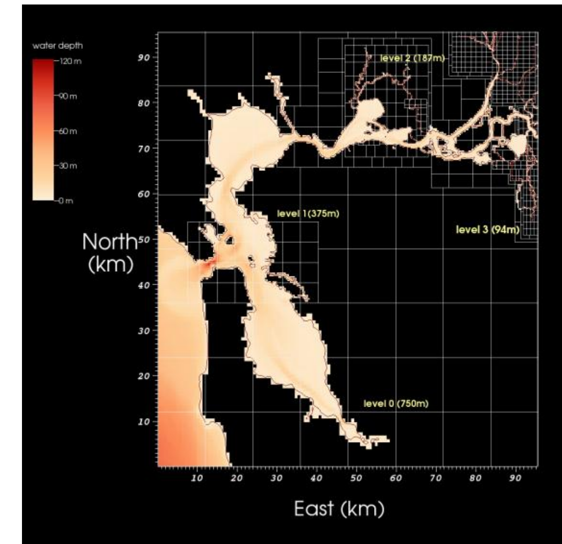
Energy/Word

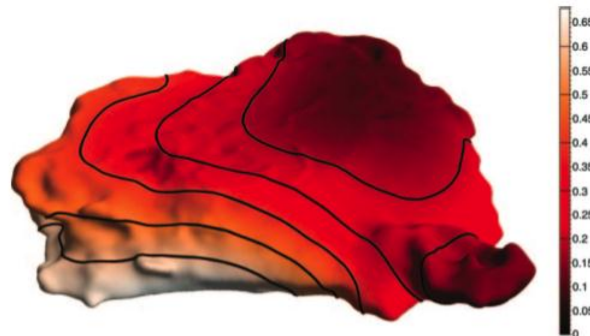
Allan Snavely, SciDAC 2010 meeti

Bytes per flop going down, access costs going up.

- **Adaptive meshes**
- Scalable **matrix-free** methods for sparse linear systems.
- Changing the discretizations to trade bytes for flops (**higher-order methods**).
- More effectively exploit **mathematical locality** to increase **data locality**.
- New landscape may change model choices / tradeoffs (e.g. reduced chemistry vs. detailed chemistry).







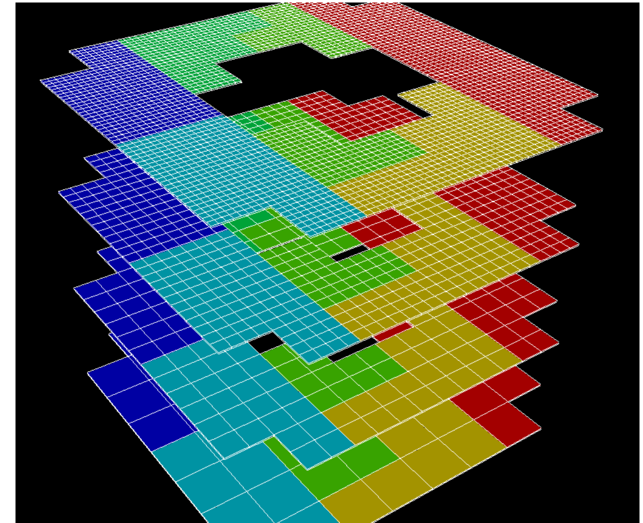
Grid resolution, problem complexity going up.

- Applications users want same throughput on finer grids (computed years / day in climate), but CFL condition on time step for explicit methods says that $dt \sim (dx)^p$, p at least 1.

- **More implicit methods** (without increased memory footprint).

- Reformulation of problem that more effectively **parallelizes across state space**.

- Aggressive use of **subcycling** in time, locally in time, physical space, state space.



Conclusions

- Everything is on the table: models, algorithms, software, (maybe) hardware.
- Hardware: multiple levels of communication: high-bandwidth, and low-bandwidth / low latency.
- Software: Languages / libraries that are sufficiently expressive of both the algorithms and the architecture. Dynamic, time-varying loads, coupled, multiple physics, multiple scales, unpredictably heterogeneous compute nodes. Bulk-synchronous parallelism is no longer viable.
- Applications: continue transition from domain scientists developing low-level components of HPC codes to writing them as a composition of components from professionally developed, high-performance scalable libraries. Reformulation of models, algorithms.
- Increasing cost of data access leads to turning modular code into monolithic code. How do we preserve reusability across applications?
- New models, algorithms and software have long lead times to have scientific impact – need to front-load these activities.
- Iterative process: bootstrap from existing programming models (MPI + X), 2013 platforms (enough threads per node).

Encapsulation

- Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state.
  - In Counter, how might a public data access result in an inconsistent state ?
- A benefit of encapsulation is that it can reduce system complexity, and thus increases robustness, by allowing the developer to limit the interdependencies between software components.
  - The code presents a contract for the programmer
  - If private data has gotten messed up, you don’t have to look over your whole code base to determine where the mistake occurred.
  - The compiler has enforced this contract for you.

Modularity

- Separation of Concerns
  - There is a small number of people that need to know how Counter is implemented.
  - There are a dozen people that need to know to use this class
  - There could be hundreds of people that just need to know this class is doing it's job.
- Improve maintainability by enforcing logical boundaries between components.
- Modules are typically incorporated into the program through interfaces.
  - C++ makes this explicit with the public interface

Compiled vs. Interpreted

- C/C++ is a *compiled* programming language.
  - A compiler turns your source code into
  - The second file (a.out) is them run afterwards to see what your program does.
  - The hallmarks of a compiled language are
    - This two step process.
    - The second file is in machine language (also called “a binary”, or “an executable”).
- This is contrasted with *interpreted* languages.
  - python, matlab scripts, perl, are examples of interpreted languages.
  - All the user works with are source files.
  - Interpreted languages need another program to read and execute them (matlab scripts are run inside matlab, python is run inside the python virtual machine, shell scripts are executed by the shell program)

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).

$$\Phi(x) = \frac{C_1}{|x|} - \frac{C_2}{|x|}$$

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

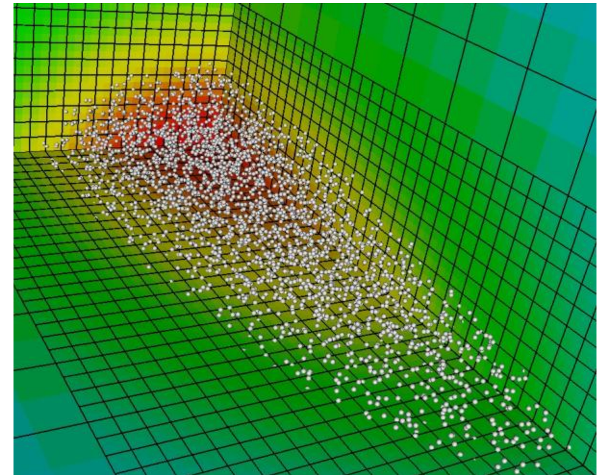
$$\nabla\Phi(x) \equiv 0 \text{ if } |x| > \sigma$$

Particle Methods

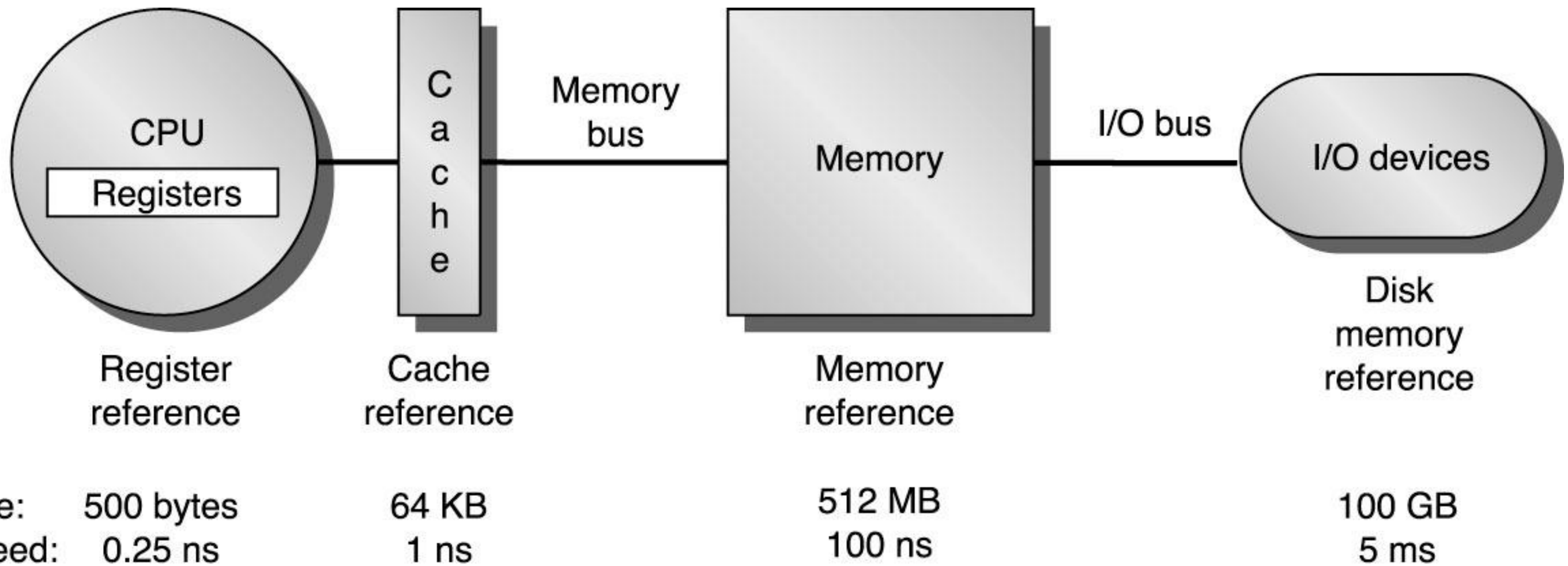
- Coulomb / Newtonian potentials

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

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.



Cache-based Processors



© 2003 Elsevier Science (USA). All rights reserved.

A CPU cache is used by the central processing unit of a computer to reduce the average time to access memory. The cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory.