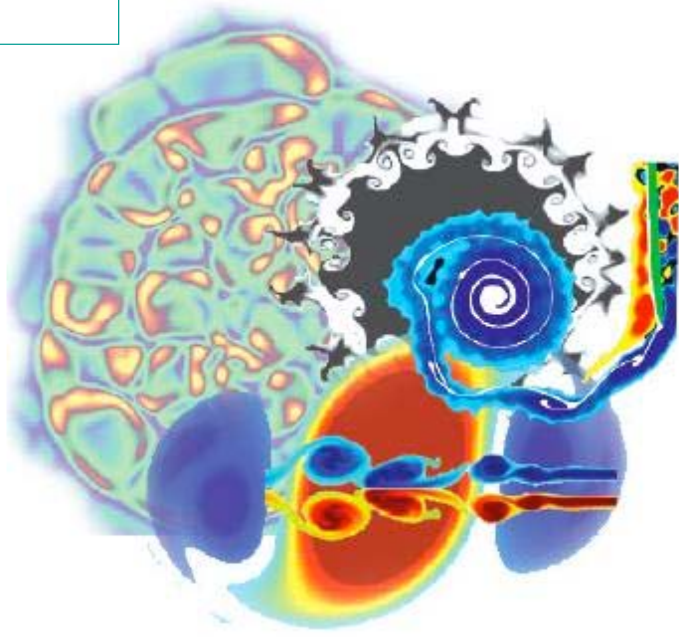
SCIENTIFIC COMPUTING ADVANCED TRAINING

Introduction to Scientific Computing, Part II

Two-lecture series for post-graduates,

Dr. Lorena Barba

University of Bristol



22 and 30 May 2006

Post-graduate lectures
Department of Mathematics

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Previous lecture (#1)

- What is scientific computing?
  - And what it's not … it's not LaTeX, ssh, usual apps …
  - It's about scientific discovery through simulation. A complex workflow …
  - … leading to the often overlooked topic of…
- Verification and validation
  - Importance of grid convergence study (even for grid-free methods!): theoretical vs. *observed* order of convergence
- Basic toolbox of numerical analysis
  - Vector, matrix operations; interpolation; discrete derivatives; integration; systems of equations; Fourier transforms; time stepping of ODEs; stochastic tools … *how to? ⇒ first step: Numerical Recipes*
- Which programming language … ?
  - Concepts of object-orientation
  - Fortran? Lots of legacy code + performance concerns with OOP
- Matlab, and other software packages + Numerical Libraries

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

This, second lecture…

- Most important algorithms of the 20th century
- Parallel implementation of scientific codes
- Revisit the subject of “which programming language?”

DISCLAIMER

- Presentation biased by my own *opinion*.
- There are other opinions, all legitimate
- Sometimes, you have no choice!
  - Supervisor wants you to use X language, non-negotiable
  - Need to work with legacy code: stuck with it

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

The issue of programming language revisited

- Fortran 77 -- I say, *forget it!*
  - Difficult to represent data structures succinctly
  - Lack of dynamic storage means that all arrays must have a fixed size which cannot be exceeded
  - Variable names only 6 characters long
  - Fixed form source format -- *argh!*
- Fortran 90 -- an “abused language”
  - People take little bits to improve their f77 codes, rarely use it how it was intended
- F90 has very powerful array facilities …
  - But f90 is not *just* allocatable arrays! More powerful than that.
  - Free form: the most obvious change from f77
  - Derived types and operator overloading

Fortran 90 arrays

- You can make any array section you can dream of…

`A(:)`	"A" is a rank one array and we are using all of it
`A(2:)`	Now we have from element 2 up
`A(:5)`	Now we only have up to element 5
`A(2:5)`	Elements 2 to 5
`A(2:6:2)`	Elements 2 to 6 in steps of 2, i.e., elements 2, 4, 6
`B(2,:)`	"B" is a rank 2 array; we are taking all the elements of the second dimension of "B" such that dimension 1 has index 2
`B(1:4,2:2)`	A rank 2 array; the matrix composed of elements of "B" with first index going from 1 to 4, and second index going from 2 to 2

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Fortran 90 array arithmetic

- Arrays can be added, subtracted, multiplied, etc., element-wise

`A = B`	Assign all elements of array “A” to the corresponding elements of array “B”
`A = B * C`	Multiply each element in “B” by the corresponding element in “C” and put the result in the corresponding element of “A”

`real :: a(10), b(10,10)`
`real :: d(10)`
`a = a + b(2,:) / d`
	Take the second dimension of “b” corresponding to the first index equal to 2, and use that in the expression. All arrays are rank one.

`A = A**2`	Square each element of “A” and put the result back in “A”

- Plus many intrinsic functions…

```
matmul(), transpose(), sum(), dot_product(), size()
```

Fortran 90 allocatable arrays

- Allocatable arrays are an important addition in Fortran 90, and one that most people know about
  - Allocatable arrays allow the sizing of an array to be postponed until it is known

Declaration	`real, allocatable                    :: a(:,:)`
	`real, dimension(:,:), allocatable :: b`

Allocation	`allocate( a(5,5), b(4,10) )`
	`allocate( a(2:5,10:2:-2), b(0:jmax) )`

Deallocation	`deallocate(a,b)`

Testing status	`if ( allocated(a) ) then`
	`...`

Fortran 90 allocatable arrays

- Limitations
  - It’s not possible to use allocatable arrays to build an expandable data storage structure “on the fly”
  - If your “dynamic array” needs to grow, then the solution is long-winded:
    - Create a temporary array, copy the contents of array that needs to grow
    - Deallocate the old array, and allocate again to desired size
    - Copy back the data held in the temporary…
    - Destroy the temporary array…
  - Can’t put allocatable arrays in abstract data types.
- New standard: Fortran 2003, allows allocatable arrays to “grow” (but not yet available from compiler makers)

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Fortran 90 derived data types

- Derived data types are a major and welcome addition in Fortran 90
  - User-defined type: data structure made up of simple types (real, integer) and other user-defined types, that can be treated like built-in types.
  - The only thing you cannot put in a user-defined type is an allocatable array (f90). However, a pointer can be used to obtain the results.
  - Also called Abstract Data Type (ADT)

Define the type	`type typeName`
	`  ! Variable declaration goes here`
	`end type`

Declare an instance	`type (typeName)  :: variable_name`
Access a component	`variable_name%component`

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

C++ classes

- In C++, abstract data types are built using ‘classes’
  - Key feature of a ‘class’ is the separation of interface and implementation
  - Class: defines a type of object by specifying the <u>data</u> it contains and <u>methods</u> that interact with the data
- Example: abstraction for a ‘particle’
  - Described by:
    - its mass,
    - a pair of 3-vectors for its position and velocity
  - Methods:
    - advancing the particle in phase space
    - computing its kinetic energy
    - construction or initialization function (how to create an *instance* of an object ‘particle’)

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Example: definition of a class 'particle'

```
class Particle {
  private:
    double mass;
    double position[3], velocity[3];
  public:
    Particle(double imass=0.0)  {
      mass = imass;
      for (int i=0; i<3; i++) {
        position[i] = 0.0;
        velocity[i] = 0.0;     }  }
    virtual ~Particle() { }
    virtual double Position(int i) const { return position[i]; }
    virtual double Velocity(int i) const { return velocity[i]; }
    virtual double Kinetic_Energy() const {
      double ke = 0.0;
      for (int i=0; i<3; i++) ke += velocity[i]*velocity[i];
      return mass*ke;                     }
    virtual double Charge() const { return 0.0; }
};
```

constructor

destructor

Encapsulation

- The internal data is accessed only through fixed interfaces.
  - If later the class “Particle” is redesigned to use momentum instead of velocity for its internal representation, no code calling the member functions of this class need be changed. Only the member functions require changing:

```
class Particle {
  private:
    ...
    double position[3], momentum[3];
  public:
    ...
    virtual double Velocity(int i) const {
      return momentum[i]/mass;           }
    virtual double Kinetic_Energy() const {
      double ke = 0.0;
      for (int i=0; i<3; i++) ke += momentum[i]*momentum[i];
      return ke/mass;
    }
    ...
};
```

Class 'Particle' member functions

- A member function Charge() is provided:

  ```
  virtual double Charge() const { return 0.0; }
  ```

  – Here it simply returns 0; the particle has no data describing its charge
- Functions declared as 'virtual':
  – Another class which inherits from 'Particle' may override, or redefine, the behavior of these functions while maintaining the same interface
- Use of the class:

  ```
  Particle  p1, p2;
  ```

  – Creates two concrete objects, "p1" and "p2"

  ```
  double ke_of_p1 = p1.Kinetic_Energy();
  ```

  – Obtains the kinetic energy of particle "p1" using the *dot syntax*.

How to implement the abstract type "particle" in f90?

SCIENTIFIC COMPUTING ADVANCED TRAINING

F90 objects – two features: type and module

- TYPE allows grouping data together, but does not associate methods with the data:

```
TYPE Particle
    REAL mass
    REAL, DIMENSION(0:2) :: position, velocity
END TYPE Particle
```

- Declare an instance and access its data:

```
TYPE(Particle)  :: p1
    REAL :: partsmass = p1%mass
```

- Encapsulation: place the TYPE inside a MODULE

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Abstract object ‘particle’ in f90

```
MODULE ParticleModule
   TYPE Particle
     PRIVATE
     REAL mass
     REAL, DIMENSION(0:2) :: position, velocity
   END TYPE Particle
CONTAINS
   SUBROUTINE Initialize(p,imass)
     TYPE(Particle), INTENT(INOUT) :: p
     REAL, OPTIONAL :: imass
     INTEGER I
     IF (PRESENT(imass)) THEN
        p%mass = imass
     ELSE
        p%mass = 0.0
     ENDIF
```

Resets the default access within the current code block

Abstract object ‘particle’ in f90 - part 2

```
    DO i=0,2
        p%position(i) = 0.0
        p%velocity(i) = 0.0
    END DO
  END SUBROUTINE Initialize
  REAL FUNCTION Position(p,i)
    TYPE(Particle), INTENT(IN) :: p
    INTEGER, INTENT(IN) :: i
    Position = p%position(i)
    RETURN
  END FUNCTION Position
  REAL FUNCTION Velocity(p,i)
    TYPE(Particle), INTENT(IN) :: p
    INTEGER, INTENT(IN) :: i
    Velocity = p%velocity(i)
    RETURN
  END FUNCTION Velocity
```

Abstract object ‘particle’ in f90 - part 3

```
    REAL FUNCTION KineticEnergy(p)
      TYPE(Particle), INTENT(IN) :: p
      INTEGER I
      REAL :: ke = 0.0
      DO i=0,2
      ke = ke + (p%velocity(i))**2
      END DO
      KineticEnergy = p%mass * ke
      RETURN
    END FUNCTION KineticEnergy
    REAL FUNCTION Charge(p)
      TYPE(Particle), INTENT(IN) :: p
      Charge = 0.0
      RETURN
    END FUNCTION Charge
END MODULE ParticleModule
```

Abstract object ‘particle’ in f90 -- usage

- Define particles in a code segment:

```
USE ParticleModule
TYPE(Particle) :: p1, p2
Initialize(p1)
Initialize(p2)
```

- Summary: F90 does allow abstract objects by combining `TYPE`s and procedures (`SUBROUTINE` and `FUNCTION`) in a `MODULE`.
  - `TYPE` - contains the internal data (encapsulated by `PRIVATE`)
  - `MODULE` - provides an interface via procedures that are public and operate on the contained `TYPE`.

SCIENTIFIC COMPUTING ADVANCED TRAINING

Inheritance and polymorphism

- Define a new abstract object that inherits the old data and methods,
  - Can alter the behavior of some functions
  - Can add new data or methods as needed

```
class Nucleus : public Particle {
  private:
    int numProtons, numNeutrons;
    static double elemCharge;
  public:
    Nucleus(int inumProtons, int inumNeutrons, double imass=0.0)
    : Particle(imass) {
      numProtons = inumProtons;
      numNeutrons = inumNeutrons;    }
    ~Nucleus() { }
    double Charge() const {
      return numProtons*elemCharge;    }};
```

Inheritance and polymorphism

- Declare two concrete objects:

```
Particle p;
Nucleus n;
```

  - Then `p.Charge()` returns the charge according to the definition of `Particle` (i.e., returns `0.0`), while `n.Charge()` returns the charge as dictated in `Nucleus`.

- Polymorphism using pointers:

```
Particle *pptr;    declare a pointer to a Particle object
pptr = &p;         asigned a value with the address-of symbol
pptr->Charge();    arrow syntax to get object particle "p" by pointer
...
pptr = &n;         allowed because Nucleus is a kind of Particle
pptr->Charge();    now uses the Nucleus Charge() function!
```

SCIENTIFIC COMPUTING ADVANCED TRAINING

Creating a Nucleus object in f90

```
MODULE NucleusModule
   USE ParticleModule
   TYPE Nucleus
     PRIVATE
     TYPE(Particle) p
     INTEGER numProtons, numNeutrons
   END TYPE Nucleus
   REAL, PRIVATE, PARAMETER :: elemCharge = 1.6e-19
   SAVE
   INTERFACE Initialize
     MODULE PROCEDURE Initialize, NucInitialize
   END INTERFACE Initialize
   INTERFACE Position
     MODULE PROCEDURE Position, NucPosition
   END INTERFACE Position
! Similar interfaces for Velocity, KineticEnergy, Charge
```

SCIENTIFIC COMPUTING ADVANCED TRAINING

Creating a Nucleus object in f90 - part 2

```
CONTAINS
   SUBROUTINE NucInitialize(n,inumProtons,inumNeutrons,imass)
     TYPE(Nucleus), INTENT(INOUT) :: n
     INTEGER, INTENT(IN) :: inumProtons, inumNeutrons
     REAL, OPTIONAL, INTENT(IN) :: imass
     Initialize(n%p,imass)
     n%numProtons = inumProtons
     n%numNeutrons = inumNeutrons
   END SUBROUTINE NucInitialize
   REAL FUNCTION NucPosition(n,i)
     TYPE(Nucleus), INTENT(IN) :: n
     INTEGER, INTENT(IN) :: I
     NucPosition = Position(n%p,i)
     RETURN
   END FUNCTION NucPosition
! Similar functions for NucVelocity, NucKineticEnergy
```

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Creating a Nucleus object in f90 - part 3

```
   ...
   REAL FUNCTION NucCharge(n)
     TYPE(Nucleus), INTENT(IN) :: n
     NucCharge = n%numProtons * elemCharge
     RETURN
   END FUNCTION NucCharge
END MODULE NucleusModule
```

- There is no way to inherit procedures because the two `TYPES` are not interchangeable as arguments
- Must recode `FUNCTION`s of `ParticleModule` for `NucleusModule`
- Cumbersome `INTERFACE` structure
- … much longer code, harder to maintain.

Conclusion on Fortran vs. C++

- Fortran90 allows object-based programming using the elements `TYPE` and `MODULE`.
- C++ is an object-oriented programming language, supporting inheritance and polymorphism
- F90 lacks inheritance and does not permit code reuse to same extent as C++
- C++ does not have a built-in array type with simple array syntax
  - Provided by C++ array class libraries
- Another C++ feature lacking in f90 is “templates” for generic programming
  - No time to discuss it here.

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Modern Algorithms

The development of new numerical algorithms is crucial, and leverages huge hardware investments.

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Top 10 Algorithms of the 20th Century

- 1946: The Monte Carlo method.
- 1947: Simplex Method for Linear Programming.
- 1950: Krylov Subspace Iteration Method.
- 1951: The Decompositional Approach to Matrix Computations.
- 1957: The Fortran Compiler.
- 1959: QR Algorithm for Computing Eigenvalues.
- 1962: Quicksort Algorithms for Sorting.
- 1965: Fast Fourier Transform.
- 1977: Integer Relation Detection.
- 1987: Fast Multipole Method.

Dongarra & Sullivan, IEEE Comput. Sci. Eng., Vol. 2(1):22--23 (2000).

Monte Carlo method

- Also known as the “Metropolis algorithm”
  - Aims to obtain approximate solutions to numerical problems with unmanageably many degrees of freedom and to combinatorial problems of factorial size, by mimicking an random process.
  - PDFs, probability density functions, describe the physical or mathematical system
  - Many simulations (“trials”) are performed
  - Results taken as an average; errors can be predicted
- Applications
  - Graphics (ray tracing)
  - Finance
  - Particle physics
  - Mathematics: integration in many dimensions

Quicksort algorithm

- Put *N* things in numerical or alphabetical order: a mundane problem! Challenge: doing so quickly.
- “Divide and conquer” strategy
- The steps are:
  1. Pick an element, called a *pivot*, from the list.
  2. Reorder the list so that all elements which are less than the pivot come before the pivot and all elements greater than the pivot come after it. After this partitioning, the pivot is in its final position. This is called the **partition** operation.
  3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

– Runs on average with O(*N* log *N*) efficiency
  – *Major* improvement over O(N^2) algorithm

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Fast Multipole Method

- For *N*-body simulations:
  - How to predict the motions of *N* particles interacting via gravitational or electrostatic forces (stars, atoms)?
  - Accurate calculations seem to require O(N^2) calculations
- Gravitational force: two masses $$\mathbf{F} = \frac{GMm\mathbf{r}}{|\mathbf{r}|^3} = \frac{GMm}{r^2}\hat{\mathbf{r}}$$

Inverse square law

- *N*-body gravitational field: $$\mathbf{E}(\mathbf{x}_j) = \sum_{i=1, i\neq j}^{N} m_i \frac{\mathbf{x}_j - \mathbf{x}_i}{r_{ij}^3}$$
- … and gravitational potential: $$\Phi(\mathbf{x}_j) = \sum_{i=1, i\neq j}^{N} \frac{m_i}{r_{ij}}$$

Direct evaluation of such a sum at *N* target points clearly results in O(N^2) operations.

SCIENTIFIC COMPUTING ADVANCED TRAINING

Fast Multipole Method - key idea

- Simple example: consider

$$s(x_i) = \sum_{j=1}^{N} \alpha_j (x_i - y_j)^2 \quad i = 1, \cdots, M$$

  - Direct summation will require *MN* operations
- Instead, can write the sum as:

$$s(x_i) = \left(\sum_{j=1}^{N} \alpha_j \right) x_i^2 + \left(\sum_{j=1}^{N} \alpha_j y_j^2 \right) - 2x_i \left(\sum_{j=1}^{N} \alpha_j y_j \right)$$

  - Can evaluate each bracketed sum over *j* then evaluate an expression of the type: $s(x_i) = \beta x_i^2 + \gamma - 2x_i \delta$
  - Requires O(*M* + *N*) operations
- Key idea – use analytical manipulation of series to achieve faster summation.

FMM "philosophy"

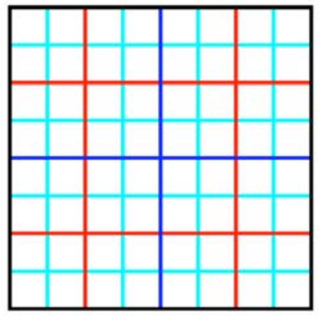
- In scientific computing we almost never seek exact answers
- At best, "exact" means to "machine precision"
- Instead of solving a problem, solve a "nearby" problem that gives "almost" the same answer.
- FMM:
  - Express functions in some appropriate functional space with a given basis
  - Manipulate series to achieve approximate evaluation
  - Use analytical expression to bound the error
- E.g. astrophysics
  - At some distance from the sources, the gravitational field is smooth and should be representable in some compressed form.

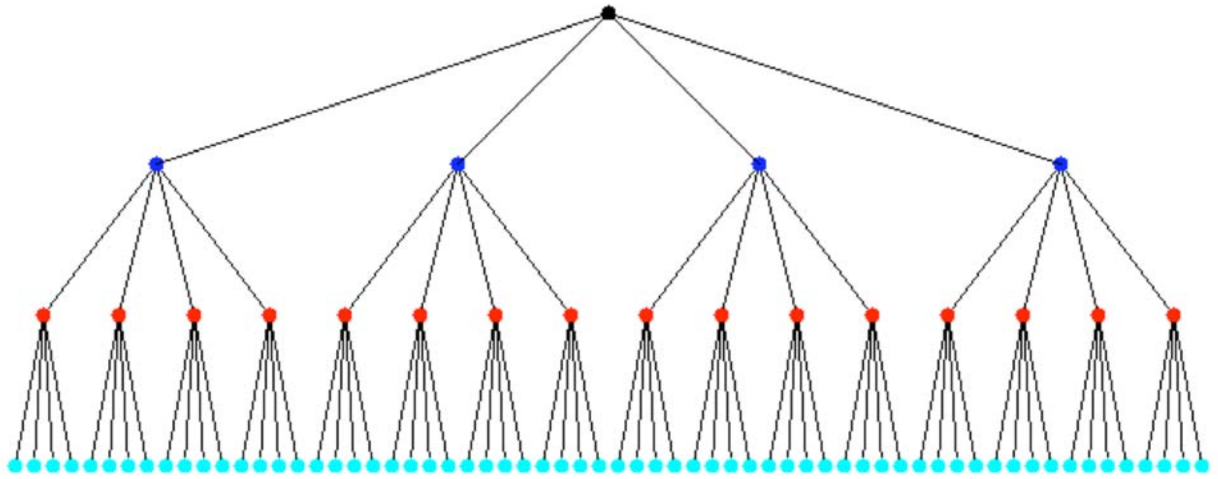
SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Hierarchical decomposition of space

- An essential part of the FMM is the data structure used to subdivide space:
  - Quadtree (2D)

A Complete Quadtree with 4 Levels

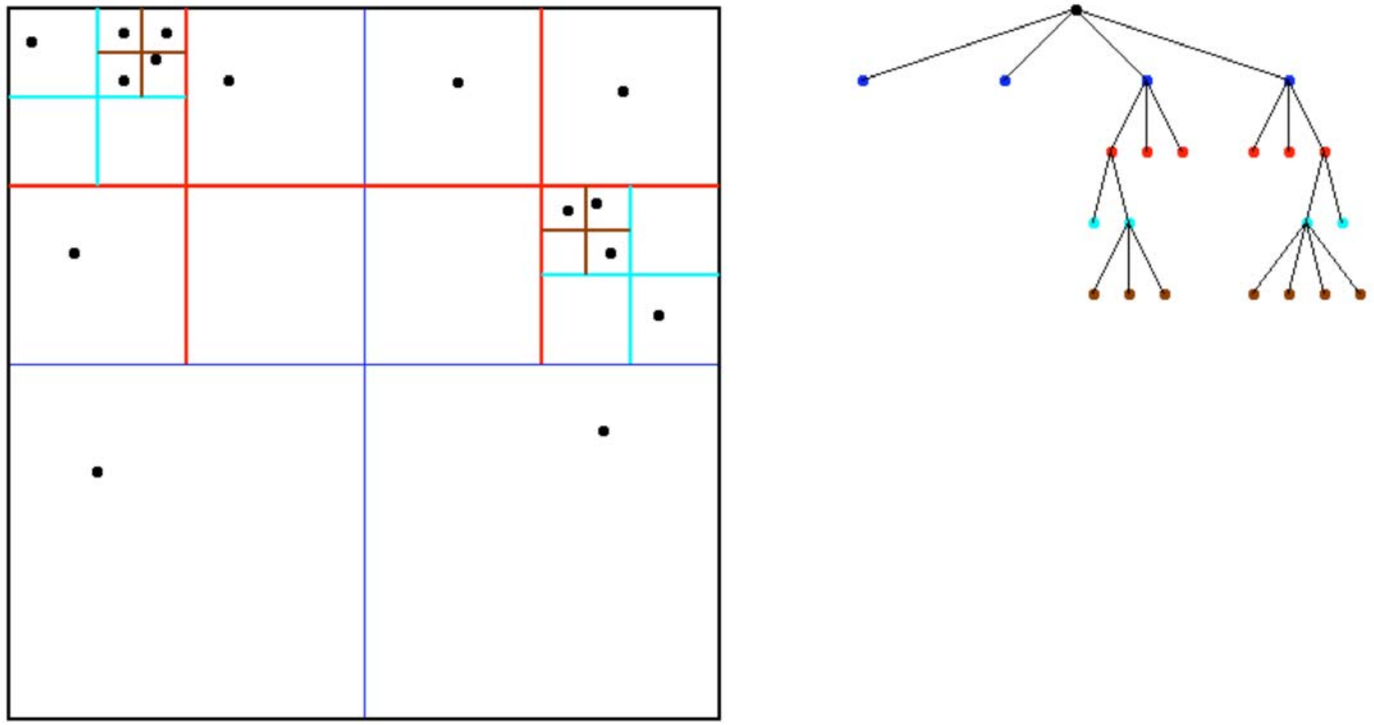




Hierarchical decomposition of space

- Begin by constructing a quadtree to store the particles

Adaptive quadtree where no square contains more than 1 particle



Idea of Multipole Expansions

- Recall: gravitational potential - satisfies the Poisson equation
  - In 2D $\phi(x,y) = \log(r)$
  - *N* points in the plane, with masses m_i

$$\phi(x,y) = \sum_{i=1}^{N} m_i \log(\sqrt{(x-x_i)^2 + (y-y_i)^2})$$

- Multipole expansion of the potential
  - A kind of Taylor expansion, but which is accurate when x^2+y^2 is large

$$\phi(z) \approx M \log(z) + \sum_{j=1}^{p} \frac{\alpha_j}{z^j}$$

… just a flavor of the FMM - would take a full lecture to present all of it

SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Parallel Computing

Two parallelization models:

MPI – distributed-memory machines

OpenMP – shared-memory machines

(MPI is most prevalent model today)

MPI -- Message Passing Interface

- Simply a collection of subroutines (in C or Fortran) which enable processors to exchange data.
  - Very portable
  - Rather steep learning curve
  - Each processor is running its own copy of the program
  - Different processors may take different paths through the code (because of conditional statements)
- In Fortran:
  - Include MPI header file

```
program myProgram
implicit none
include ‘mpif.h’
...
```

Basic MPI in Fortran

- Initialization/Finalization

```
integer error
call MPI_INIT(error)
...
call MPI_FINALIZE(error)
end program
```

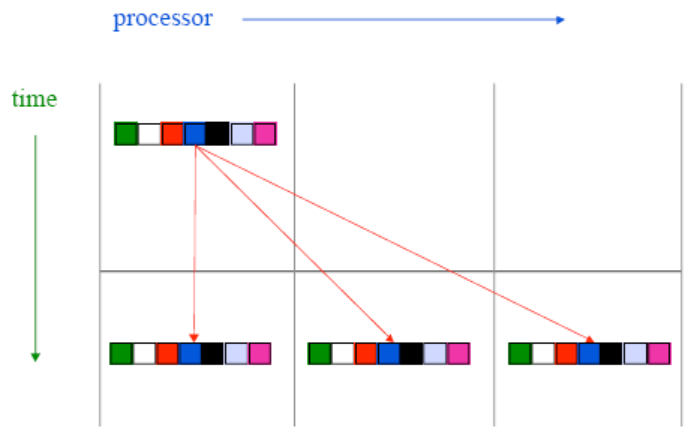
- Communicators
  - Like a network linking certain processors
  - Global communicator: `MPI_COMM_WORLD`
  - Determine the number of processors in a communicator

```
integer num_procs, error
call MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, error)
! num_procs will have been set to the number of processors
! in MPI_COMM_WORLD
```

Collective communications

- Broadcast
  - Send some data to all other processors in a communicator

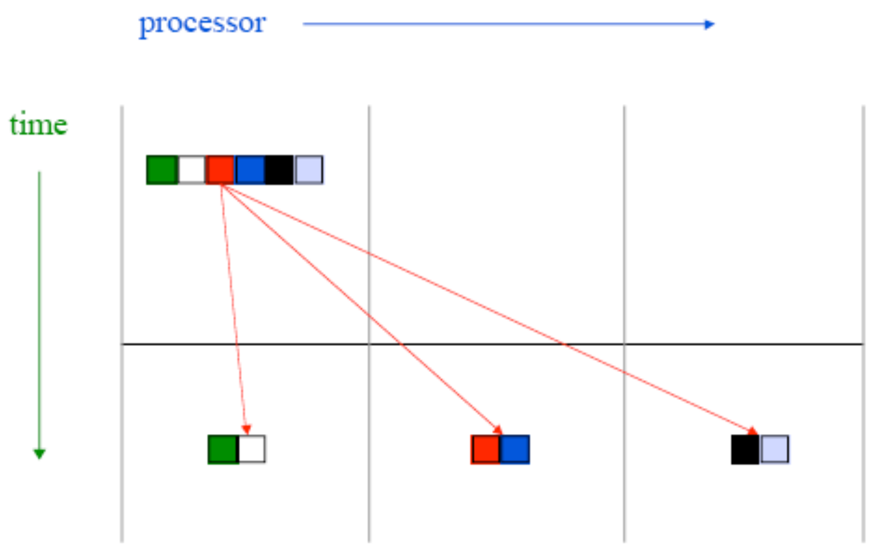
```
integer n = 10
integer error
integer bcastProc = 0 ! broadcasting processor
real*8 array(n)
call MPI_BCAST(array,size(array),MPI_REAL8,bcastProc, &
               MPI_COMM_WORLD,error)
```



SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Collective communications

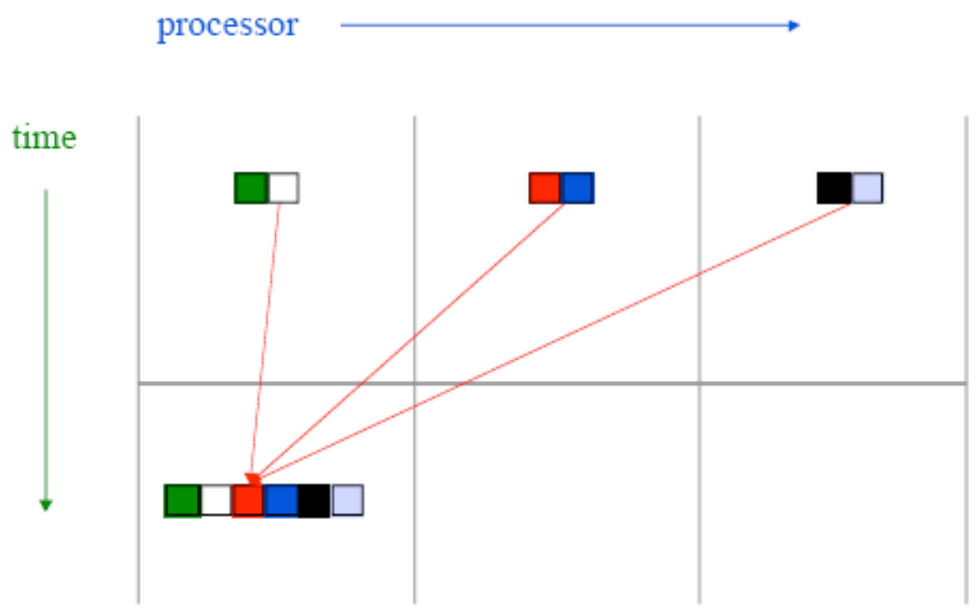
- Scatter
  - Sender divides some array of data up into as many portions as there are processors, and sends each processor one portion



SCIENTIFIC COMPUTING ADVANCED TRAINING

Collective communications

- Gather
  - The opposite of scattering: each processor has an array of data, and all of these are gathered and delivered to one processor



SCAT
SCIENTIFIC COMPUTING ADVANCED TRAINING

Parallel Scientific Libraries -- PETSc

- A powerful set of tools for the numerical solution of partial differential equations and related problems on high-performance computers.
- MPI *almost* invisible to the programmer
- PETSc objects:
  - Parallel vectors, matrices
  - Krylov subspace methods
  - Nonlinear solvers
  - Time steppers
- Based on BLAS, LAPACK, MPI
- Developed at Argonne National Laboratory; fully supported; free.

SCIENTIFIC COMPUTING ADVANCED TRAINING

PETSc examples

```
Vec x, b, u;
Mat A;
...
ierr = VecCreate(PETSC_COMM_WORLD,&x);CHKERRQ(ierr);
ierr = VecDuplicate(x,&b);CHKERRQ(ierr);
ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
```

- Etc.
  - Create matrices and vectors, let the library distribute among procs
  - Call solvers, use preconditioners… all in parallel

- Google: petsc