Guides, Unit tests, Object orientation and Parallel programming using MPI and OpenMP

Morten Hjorth-Jensen

Michigan State University, Michigan, U.S.A. and
University of Oslo, Oslo, Norway

Nuclear Talent course on DFT, July and August 2014, ECT*

Version control with Git, recommended

Git is an open source version control software, that makes it possible to have ”versions” of a project. That is, snapshots of the files in the project at certain points in time. By having different versions of a project, it is possible to see the changes that have been made to the code over time, and it is also possible to revert the project to another version. It should mentioned that when files remain unchanged from one version to another, Git simply links to the previous files, making everything fast and clean.

Qt creator for C++ programmers

Qt is a cross-platform ide and is part of the Qt Project. It consist of a number of features with the aim to increase the productivity of the developer and to help organizing large projects. Some of the features included in its editor are:

- rapid code navigation tools,
- syntax highlighting and code completion,
- static code checking and style hints as you type,
- context sensitive help,
- code folding.

Qt creator for C++ programmers

Qt includes a debugger plugin, providing a simplified representation of the raw information provided by the external native debuggers to debug the C++ language. Some of the possibilities in debugging mode are:

- interrupt program execution,
- step through the program line-by-line or instruction-by-instruction,
- set breakpoints,
- examine call stack contents, watchers, and local and global variables.

Qt also provides useful code analysis tools for detecting memory leaks and profiling function execution. For more details see the online resources on Qt.

Armadillo for C++ programmers

arma is an open source C++ linear algebra library, with the aim to provide an intuitive interface combined with efficient calculations. Its functionalities includes efficient classes for vectors, matrices and cubes, as well as many functions which operate on the classes. Some of the functionalities of armadillo are demonstrated in the example below:

```
vec x(10);                          // column vector of
    length 10
rowvec y = zeros<rowvec>(10);       // row vector of
    length 10

mat    A = randu<mat>(10,10);       // random matrix of
    dimension 10 X 10
rowvec z = A.row(5);                // extract a row
    vector

cube q(4,5,6);                      // cube of dimension
    4 X 5 X 6
mat B = q.slice(1);                 // extract a slice
    from the cube
                                    // (each slice is a
                                        matrix)
```

Armadillo

One very useful class in armadillo is **field**, where arbitrary objects in matrix-like or cube-like layouts can be stored. Each of these objects can have an arbitrary size. Here is an example of the usage of the **field** class:

```
field<vec> F(3,2);           // a field of dimension 3
    X 2 containing vectors

// each vector in the field can have an arbitrary
    size
F(0,0) = vec(5);
F(1,1) = randu<vec>(6);
F(2,0).set_size(7);

double x = F(2,0)(1);        // access element 1 of
    vector stored at 2,0
F.row(0) = F.row(2);         // copy a row of vectors
field<vec> G = F.row(1);     // extract a row of
    vectors from F
```

IPython Notebook

IPython Notebook is a web-based interactive computational environment for python where code execution, text, mathematics, plots and rich media can be combined into a single document. Some of the main features of ipynb are:

- In-browser editing for code, with automatic syntax highlighting, indentation, and tab completion/introspection.
- The ability to execute code from the browser, with the results of computations attached to the code which generated them.
- Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc.
- In-browser editing for rich text using the Markdown markup language, which can provide commentary for the code.
- The ability to easily include mathematical notation within markdown cells using LaTeX, and rendered natively by MathJax.

One very nice of feature of IPython Notebook documents is that they can be shared via the nbviewer, as long as they are publicly available. This service renders the notebook document, specified by an url, as a static web page. This makes it easy to share a document with other users that can read the document immediately without having to install anything.

SymPy

SymPy is a python library for doing symbolic math, including features such as basic symbolic arithmetic, simplification and other methods of rewriting, algebra, differentiation and integration, discrete mathematics and even quantum physics. SymPy is also able to format the result of the computations as LaTeX, ASCII, Fortran, C++ and python code. Some of the named features of SymPy are shown on the next slide.

SymPy

```
>>> from sympy import *
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> x+y+x-y
2*x
>>> simplify((x+x*y)/x)
1 + y
>>> series(cos(x), x)
1 - x**2/2 + x**4/24 + O(x**6)
>>> diff(sin(x), x)
cos(x)
>>> integrate(log(x), x)
-x + x*log(x)
>>> solve([x + 5*y - 2, -3*x + 6*y - 15], [x, y])
{y: 1, x: -3}
```

Hierarchical Data Format 5 (hdf5)

hdf5 is a library and binary file format for storing and organizing large amounts of numerical data, and is supported by many software platforms including Fortran, C++ and python. The core concepts in hdf5 are datasets, groups and attributes. Datasets are array-like collections of data which can be of any size and dimension, groups are folder-like collections consisting of datasets and other groups, and attributes are metadata associated with a group or dataset, stored right next to the data it describes. This limited primary structure makes the file design simple, but provides at the same time a very structured way to store data. Here is a short list of advantages of the hdf5 format:

- open-source software,
- different data types (images, tables, arrays, etc.) can be combined in one single file,
- support for user-defined data types,
- data can be accessed independently of the platform that generated the data,
- possible to read only part of the data, not the whole file,
- source code examples for reading and writing in this format is widely available.

Unit Testing

Unit Testing is the practice of testing the smallest testable parts, called units, of an application individually and independently to determine if they behave exactly as expected. Unit tests (short code fragments) are usually written such that they can be preformed at any time during the development to continually verify the behavior of the code. In this way, possible bugs will be identified early in the development cycle, making the debugging at later stage much easier. There are many benefits associated with Unit Testing, such as

- It increases confidence in changing and maintaining code. Big changes can be made to the code quickly, since the tests will ensure that everything still is working properly.
- Since the code needs to be modular to make Unit Testing possible, the code will be easier to reuse. This improves the code design.
- Debugging is easier, since when a test fails, only the latest changes need to be debugged.
- Different parts of a project can be tested without the need to wait for the other parts to be available.
- A unit test can serve as a documentation on the functionality of a unit of the code.

Object orientation, Fortran and C++

Why object orientation?

- Three main topics: objects, class hierarchies and polymorphism
- The aim here is to be to be able to write a more general code which can easily be tailored to new situations.
- **Polymorphism** is a term used in software development to describe a variety of techniques employed by programmers to create flexible and reusable software components. The term is Greek and it loosely translates to ”many forms”.

Strategy: try to single out the variables needed to describe a given system and those needed to describe a given solver.

Object orientation, Fortran and C++

In programming languages, a polymorphic object is an entity, such as a variable or a procedure, that can hold or operate on values of differing types during the program's execution. Because a polymorphic object can operate on a variety of values and types, it can also be used in a variety of programs, sometimes with little or no change by the programmer. The idea of write once, run many, also known as code reusability, is an important characteristic to the programming paradigm known as Object-Oriented Programming (OOP).
OOP describes an approach to programming where a program is viewed as a collection of interacting, but mostly independent software components. These software components are known as objects in OOP and they are typically implemented in a programming language as an entity that encapsulates both data and procedures.

Object orientation, Fortran and C++

A Fortran 90/95 module can be viewed as an object because it can encapsulate both data and procedures. Fortran 2003 (F2003 and now F2008) added the ability for a derived type to encapsulate procedures in addition to data. By definition, a derived type can now be viewed as an object as well in F2008.

F2008 also introduced type extension to its derived types. This feature allows F2008 programmers to take advantage of one of the more powerful OOP features known as inheritance. Inheritance allows code reusability through an implied inheritance link in which leaf objects, known as children, reuse components from their parent and ancestor objects.

Object orientation in C++

A class is a collection of variables and functions. By defining a class one determines what type of data and which kind of operations that can be preformed on these data. The variables and functions in a class are called class members. As an example, we consider the definition of a class for gaussian type orbitals:

```
class PrimitiveGTO
{
public:
    PrimitiveGTO();
    ~PrimitiveGTO();
    const double &exponent() const;
    void setExponent(const double &exponent);

    const double &weight() const;
    void setWeight(const double &weight);
    ...

private:
    double m_exponent;
    double m_weight;
    ...
};
```

Object orientation in C++

A class definition starts with the keyword **class** followed by the name of the class. The class body contains member variables and functions, in this example **m_exponent**, **m_weight**. The keywords **public** and **private** are access modifiers and set the accessibility of member variables and member functions. A **public** member can be assessed anywhere outside the class, while a **private** member only can be accessed within the current class.

Object orientation in C++

An instance of a class is called object. That is, a self-contained component that consist of both data and methods to manipulate the data. A **PrimitiveGTO** object can be declared by

```
PrimitiveGTO pGTO(); //or as a pointer
PrimitiveGTO* pGTO = new PrimitiveGTO();
```

Declaration of an object calls the constructor function **PrimitiveGTO()**) in a class, which initialize the new object. The constructor can have input parameters, used to assign values to member variables. To delete an object the destructor function (˜**PrimitiveGTO()**) is called.

Object orientation in C++

In object-oriented programming, objects can inherit properties and methods from existing classes. Inheritance provides the opportunity to reuse existing code. A class that is defined in terms of another class, is called a subclass or derived class, while the class used as the basis for inheritance is called a superclass or base class. The terms child class and parent class are also common to use for the subclass and superclass, respectively. An example of inheritance is shown below, where the class **RHF** is derived from the base class **HFsolver**:

Object orientation in C++

```
class HFsolver
{
public:
    HFsolver(ElectronicSystem *system);

    virtual void solveSingle() = 0;
    virtual void calculateEnergy() = 0;
    ...

protected:
    int m_nElectrons;
    ...
};
```

Object orientation in C++

```
class RHF : public HFsolver
{
public:
    RHF(ElectronicSystem *system);

    void solveSingle();
    void calculateEnergy();
    ...
};
```

When an object of class **RHF** is declared, it inherits all the members of **HFsolver** beside the private members of **HFsolver**. Note the special declaration of the functions in the **HFsolver** class. These functions are virtual functions whose behavior can be overridden in a derived class, allowing efficient implementation of new solvers.

Object orientation, Fortran

Example

```
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
end type shape
type, EXTENDS ( shape ) :: rectangle
        integer :: length
        integer :: width
end type rectangle
type, EXTENDS ( rectangle ) :: square
end type square
```

Object orientation, Fortran

We have a square type that inherits components from rectangle which inherits components from shape. The programmer indicates the inheritance relationship with the EXTENDS keyword followed by the name of the parent type in parentheses. A type that EXTENDS another type is known as a type extension (e.g., rectangle is a type extension of shape, square is a type extension of rectangle and shape). A type without any EXTENDS keyword is known as a base type (e.g., shape is a base type).

Object orientation, Fortran

A type extension inherits all of the components of its parent (and ancestor) types. A type extension can also define additional components as well. For example, rectangle has a length and width component in addition to the color, filled, x, and y components that were inherited from shape. The square type, on the other hand, inherits all of the components from rectangle and shape, but does not define any components specific to square objects. Below is an example on how we may access the color component of square:

```
type(square) :: sq                    ! declare sq as a
    square object
sq%color                              ! access color
    component for sq
sq%rectangle%color                    ! access color
    component for sq
sq%reactangle%shape%color             ! access color
    component for sq
```

All these declarations are equivalent. A type extension includes an implicit component with the same name and type as its parent type. This can come in handy when the programmer wants to operate on components specific to a parent type. It also helps illustrate an important relationship between the child and parent types.

Object orientation, Polymorphism in Fortran

The CLASS keyword allows F2008 programmers to create polymorphic variables. A polymorphic variable is a variable whose data type is dynamic at runtime. It must be a pointer variable, allocatable variable, or a dummy argument. Below is an example:

```
class(shape), pointer :: sh
```

In the example above, the sh object can be a pointer to a shape or any of its type extensions. So, it can be a pointer to a shape, a rectangle, a square, or any future type extension of shape. As long as the type of the pointer target "is a" shape, sh can point to it.

There are two basic types of polymorphism: procedure polymorphism and data polymorphism. Procedure polymorphism deals with procedures that can operate on a variety of data types and values. Data polymorphism deals with program variables that can store and operate on a variety of data types and values.

Object orientation, Polymorphism in Fortran

Procedure polymorphism occurs when a procedure, such as a function or a subroutine, can take a variety of data types as arguments. This is accomplished in F2008 when a procedure has one or more dummy arguments declared with the CLASS keyword. For example,

```
subroutine setColor(sh, color)
class(shape) :: sh
integer :: color
sh%color = color
end subroutine setColor
```

The setColor subroutine takes two arguments, sh and color. The sh dummy argument is polymorphic, based on the usage of class(shape). The subroutine can operate on objects that satisfy the ”is a” shape relationship. So, setColor can be called with a shape, rectangle, square, or any future type extension of shape

Object orientation, Polymorphism in Fortran

However, by default, only those components found in the declared type of an object are accessible. For example, shape is the declared type of sh. Therefore, you can only access the shape components, by default, for sh in setColor, that is

```
sh%color , sh%filled , sh%x , sh%y
```

If the programmer needs to access the components of the dynamic type of an object then they can use the F2008 SELECT TYPE construct.

Object orientation, Polymorphism in Fortran

The following example illustrates how a SELECT TYPE construct can access the components of a dynamic type of an object:

```
subroutine initialize(sh, color, filled, x, y,
    length, width)
! initialize shape objects
class(shape) :: sh
integer :: color
logical :: filled
integer :: x
integer :: y
integer, optional :: length
integer, optional :: width

sh%color = color
sh%filled = filled
sh%x = x
sh%y = y
```

Object orientation, Polymorphism in Fortran

```
select type (sh)
type is (shape)
      ! no further initialization required
class is (rectangle)
    ! rectangle or square specific initializations
    if (present(length)) then
       sh%length = length
    else
       sh%length = 0
    endif
    if (present(width)) then
        sh%width = width
    else
        sh%width = 0
    endif
class default
  ! give error for unexpected/unsupported type
     stop 'initialize: unexpected type for sh
         object!'
end select
```

Object orientation, Polymorphism in Fortran

The above example illustrates an initialization procedure for our shape example. It takes one shape argument, sh, and a set of initial values for the components of sh. Two optional arguments, length and width, are specified when we want to initialize a rectangle or a square object. The SELECT TYPE construct allows us to perform a type check on an object. There are two styles of type checks that we can perform. The first type check is called ”type is”. This type test is satisfied if the dynamic type of the object is the same as the type specified in parentheses following the ”type is” keyword. The second type check is called ”class is”. This type test is satisfied if the dynamic type of the object is the same or an extension of the specified type in parentheses following the ”class is” keyword.

Object orientation, Polymorphism in Fortran

Derived types in F2008 are considered objects because they now can encapsulate data as well as procedures. Procedures encapsulated in a derived type are called type-bound procedures. The example below illustrates how we may add a type-bound procedure to shape:

```
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
    contains
        procedure :: initialize
end type shape
```

Object orientation, Polymorphism in Fortran

Most OOP languages allow a child object to override a procedure inherited from its parent object. This is known as procedure overriding. In F2008, we can specify a type-bound procedure in a child type that has the same binding-name as a type-bound procedure in the parent type. When the child overrides a particular type-bound procedure, the version defined in its derived type will get invoked instead of the version defined in the parent. Below is an example where rectangle defines an initialize type-bound procedure that overrides shape's initialize type-bound procedure:

Object orientation, Polymorphism in Fortran

```
module shape_mod
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
    contains
        procedure :: initialize => initShape
end type shape
type, EXTENDS ( shape ) :: rectangle
        integer :: length
        integer :: width
    contains
        procedure :: initialize => initRectangle
end type rectangle
type, EXTENDS ( rectangle ) :: square
end type square
```

Object orientation, Polymorphism in Fortran

```
contains
subroutine initShape(this, color, filled, x, y,
    length, width)
! initialize shape objects
class(shape) :: this
integer :: color
logical :: filled
integer :: x
integer :: y
integer, optional :: length   ! ingnored for shape
integer, optional :: width    ! ignored for shape

this%color = color
this%filled = filled
this%x = x
this%y = y
end subroutine
```

Object orientation, Polymorphism in Fortran

```
subroutine initRectangle(this, color, filled, x, y,
    length, width)
! initialize rectangle objects
class(rectangle) :: this
integer :: color
logical :: filled
integer :: x
integer :: y
integer, optional :: length
integer, optional :: width

this%color = color
this%filled = filled
this%x = x
this%y = y
```

Object orientation, Polymorphism in Fortran

Continues

```
if (present(length)) then
    this%length = length
else
    this%length = 0
endif
if (present(width)) then
     this%width = width
else
      this%width = 0
endif
end subroutine
end module
```

In the sample code above, we defined a type-bound procedure called initialize for both shape and rectangle. The only difference is that shape's version of initialize will invoke a procedure called initShape and rectangle's version will invoke a procedure called initRectangle.

Object orientation, Polymorphism in Fortran

Note that the passed-object dummy in initShape is declared ”class(shape)” and the passed-object dummy in initRectangle is declared ”class(rectangle)”. A type-bound procedure’s passed-object dummy must match the type of the derived type that defined it. Other than differing passed-object dummy arguments, the interface for the child’s overriding type-bound procedure is identical with the interface for the parent’s type-bound procedure. That is because both type-bound procedures are invoked in the same manner:

```
type(shape) :: shp
                                        ! declare an
    instance of shape
type(rectangle) :: rect
                                    ! declare an
    instance of rectangle
type(square) :: sq
                                        ! declare an
    instance of square
call shp%initialize(1, .true., 10, 20)
                    ! calls initShape
call rect%initialize(2, .false., 100, 200, 11, 22)
     ! calls initRectangle
call sq%initialize(3, .false., 400, 500)
                   ! calls initRectangle
```

Object orientation, Polymorphism in Fortran

Note that **sq** is declared square but its initialize type-bound procedure invokes initRectangle because sq inherits the rectangle version of initialize.
Although a type may override a type-bound procedure, it is still possible to invoke the version defined by a parent type. Each type extension contains an implicit parent object of the same name and type as the parent. We can use this implicit parent object to access components specific to a parent, say, a parent's version of a type-bound procedure:

```
call rect%shape%initialize(2, .false., 100, 200)
                ! calls initShape
call sq%rectangle%shape%initialize(3, .false., 400,
    500) ! calls initShape
```

Object orientation, Polymorphism in Fortran

A quantum-mechanical example

```
MODULE singleparticledata
    USE constants
    USE inifile
    USE setupsystem
    IMPLICIT NONE
    PRIVATE

    TYPE, PUBLIC :: configuration_descriptor
      INTEGER :: numberconfs
      INTEGER, DIMENSION(:), POINTER :: config
    END TYPE configuration_descriptor
```

Object orientation, Polymorphism in Fortran

A quantum-mechanical example

```
! This is the basis type used, and contains all
    quantum numbers necessary
! for fermions in one dimension
TYPE, PUBLIC :: SpQuantumNumbers
    ! n is the principal quantum number taken as
        number of nodes-1
    ! s is the spin and ms is the spin
        projection, and parity is obvious
    INTEGER :: ndata
    INTEGER, DIMENSION(:), POINTER :: n, s, ms,
        parity => null()
    CHARACTER(LEN=100), DIMENSION(:), POINTER ::
        orbit_status, model_space => null()
    REAL(DP), DIMENSION(:), POINTER :: masses,
        energy => null()
    CONTAINS
        PROCEDURE :: initialize => init1dim
        PROCEDURE :: output => output1dim
        PROCEDURE :: countconfigs =>
            countconfigs1dim
```

Object orientation, Polymorphism in Fortran

```
! We add then quantum numbers appropriate for
    two-dimensional systems,
! suitable  for electrons in quantum dots for
    example
! Use as TYPE(TwoDim) :: qdelectrons
! n => qdelectrons%n
TYPE, EXTENDS(SpQuantumNumbers), PUBLIC   ::
    TwoDim
    INTEGER, DIMENSION(:), POINTER :: ml => null
        ()
    CONTAINS
        PROCEDURE :: initialize => init2dim
        PROCEDURE :: output => output2dim
        PROCEDURE :: countconfigs =>
            countconfigs2dim
        PROCEDURE :: setupconfigs =>
            setupconfigs2dim
END TYPE TwoDim
```

Object orientation, Polymorphism in Fortran

```
! Then we extend to three dimensions, suitable
    for atoms and electrons in
! 3d traps
! Use as TYPE(ThreeDim) :: electrons
! n => electrons%n
TYPE, EXTENDS(TwoDim), PUBLIC :: ThreeDim
    INTEGER, DIMENSION(:), POINTER :: l, j, mj =>
        null()
    CONTAINS
        PROCEDURE :: initialize => init3dim
        PROCEDURE :: output => output3dim
        PROCEDURE :: countconfigs =>
            countconfigs3dim
        PROCEDURE :: setupconfigs =>
            setupconfigs3dim
END TYPE ThreeDim
```

Object orientation, Polymorphism in Fortran

```
! Then we extends to nucleons (protons and
    neutrons), note that the masses are in
! SpQuantumNumbers. We add isospin and its
    projections
! Use as TYPE(nucleons) :: protons
! n => protons%n
TYPE, EXTENDS(ThreeDim), PUBLIC :: nucleons
    INTEGER, DIMENSION(:), POINTER :: t, tz =>
        null()
    CONTAINS
        PROCEDURE :: initialize => initnucleons
        PROCEDURE :: output => outputnucleons
        PROCEDURE :: countconfigs =>
            countconfigsnucleons
        PROCEDURE :: setupconfigs =>
            setupconfigsnucleons

END TYPE nucleons
```

Object orientation, Polymorphism in Fortran

```
! Finally we allow for studies of hypernuclei,
    adding strangeness
! Use as TYPE(hyperons) :: sigma
! n => sigma%n; s => sigma%strange
TYPE, EXTENDS(nucleons), PUBLIC :: hyperons
    INTEGER, DIMENSION(:), POINTER :: strange  =>
        null()
    CONTAINS
        PROCEDURE :: initialize => inithyperons
        PROCEDURE :: output => outputhyperons
        PROCEDURE :: countconfigs =>
            countconfigshyperons
        PROCEDURE :: setupconfigs =>
            setupconfigshyperons
    END TYPE hyperons
```

Object orientation, Polymorphism in Fortran

Initializing data

```
CONTAINS
    SUBROUTINE init1dim(this)
        CLASS(SpQuantumNumbers) :: this
        INTEGER :: i
        ALLOCATE(this%n(this%ndata), this%s(this%
            ndata))
        ALLOCATE(this%ms(this%ndata),this%parity(
            this%ndata))
        ALLOCATE(this%orbit_status(this%ndata), this
            %model_space(this%ndata))
        ALLOCATE(this%energy(this%ndata), this%
            masses(this%ndata))
```

Object orientation, Polymorphism in Fortran

Initializing data, continues

```
    DO i= 1, this%ndata
        this%model_space(i)= ' '; this%orbit_status(i
            )= ' '
        this%energy(i)=0.0_dp; this%masses(i)=0.0_dp
        this%n(i)=0; this%ms(i)=0; this%s(i)=0
        this%parity(i)=0
    ENDDO
END SUBROUTINE init1dim
```

Object orientation, Polymorphism in Fortran

An example of an output file

```
SUBROUTINE outputnucleons(this, out_unit)
    CLASS(nucleons) :: this
    INTEGER :: i, out_unit
    DO i= 1, this%ndata
        WRITE(out_unit,'(6I12,2X,2E16.8,2X,2A12)')
            this%n(i), this%mj(i), this%l(i), this%j(
            i), this%t(i), &
        this%tz(i), this%energy(i), this%masses(i),
            this%model_space(i), &
        this%orbit_status(i)
    ENDDO
END SUBROUTINE outputnucleons
```

Object orientation, Polymorphism in Fortran

Simple usage

```
PROGRAM obd_main
    USE constants
    USE inifile
    USE singleparticledata

    CLASS (nucleons), POINTER :: neutrons => NULL()
    CALL neutrons%initialize()
    CALL neutrons%output(6)

END PROGRAM obd_main
```

Target group and miscellania

- You have some experience in programming but have never tried to parallelize your codes
- Here I will base my examples on C/C++ and Fortran using Message Passing Interface (MPI) and OpenMP.
- Good text: Karniadakis and Kirby, Parallel Scientific Computing in C++ and MPI, Cambridge.

Strategies

- Develop codes locally, run with some few processes and test your codes. Do benchmarking, timing and so forth on local nodes, for example your laptop or PC. You can install MPICH2 on your laptop/PC.
- Test by typing *which mpd*
- When you are convinced that your codes run correctly, you start your production runs on available supercomputers, in our case titan.uio.no.

How do I run MPI on a PC/Laptop? (Ubuntu/linux setup here)

- Compile with mpicxx or mpic++ or mpif90
- Set up collaboration between processes and run

```
mpd --ncpus=4 &
#  run code with
mpiexec -n 4 ./nameofprog
```

  Here we declare that we will use 4 processes via the $-ncpus$ option and via $-n4$ when running.

- End with

```
mpdallexit
```

Can I do it on my own PC/laptop?

Of course:

- go to `http://www.mcs.anl.gov/research/projects/mpich2/`
- follow the instructions and install it on your own PC/laptop
- Versions for Ubuntu/Linux, windows and mac
- For windows, you may think of installing WUBI
- And for mac, parallels is a good software, vmware as well.

What is Message Passing Interface (MPI)?

MPI is a library, not a language. It specifies the names, calling sequences and results of functions or subroutines to be called from C/C++ or Fortran programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran, C or C++ are compiled with ordinary compilers and linked with the MPI library.
MPI programs should be able to run on all possible machines and run all MPI implementetations without change.
An MPI computation is a collection of processes communicating with messages.

Going Parallel with MPI

Task parallelism: the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulations or numerical integration are examples of this.
MPI is a message-passing library where all the routines have corresponding C/C++-binding

```
MPI_Command_name
```

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

```
MPI_COMMAND_NAME
```

MPI

MPI is a library specification for the message passing interface, proposed as a standard.

- independent of hardware;
- not a language or compiler specification;
- not a specific implementation or product.

A message passing standard for portability and ease-of-use.
Designed for high performance.
Insert communication and synchronization functions where necessary.

The basic ideas of parallel computing

- Pursuit of shorter computation time and larger simulation size gives rise to parallel computing.
- Multiple processors are involved to solve a global problem.
- The essence is to divide the entire computation evenly among collaborative processors. Divide and conquer.

A rough classification of hardware models

- Conventional single-processor computers can be called SISD (single-instruction-single-data) machines.
- SIMD (single-instruction-multiple-data) machines incorporate the idea of parallel processing, which use a large number of process- ing units to execute the same instruction on different data.
- Modern parallel computers are so-called MIMD (multiple-instruction- multiple-data) machines and can execute different instruction streams in parallel on different data.

Shared memory and distributed memory

- One way of categorizing modern parallel computers is to look at the memory configuration.
- In shared memory systems the CPUs share the same address space. Any CPU can access any data in the global memory.
- In distributed memory systems each CPU has its own memory. The CPUs are connected by some network and may exchange messages.

Different parallel programming paradigms

- **Task parallelism** the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulation is one example. Integration is another. However this paradigm is of limited use.
- **Data parallelism** use of multiple threads (e.g. one thread per processor) to dissect loops over arrays etc. This paradigm requires a single memory address space. Communication and synchronization between processors are often hidden, thus easy to program. However, the user surrenders much control to a specialized compiler. Examples of data parallelism are compiler-based parallelization and OpenMP directives.

Different parallel programming paradigms

- **Message-passing** all involved processors have an independent memory address space. The user is responsible for partitioning the data/work of a global problem and distributing the subproblems to the processors. Collaboration between processors is achieved by explicit message passing, which is used for data transfer plus synchronization.
- This paradigm is the most general one where the user has full control. Better parallel efficiency is usually achieved by explicit message passing. However, message-passing programming is more difficult.

SPMD

Although message-passing programming supports MIMD, it suffices with an SPMD (single-program-multiple-data) model, which is flexible enough for practical cases:

- Same executable for all the processors.
- Each processor works primarily with its assigned local data.
- Progression of code is allowed to differ between synchronization points.
- Possible to have a master/slave model. The standard option in Monte Carlo calculations and numerical integration.

Today's situation of parallel computing

- Distributed memory is the dominant hardware configuration. There is a large diversity in these machines, from MPP (massively parallel processing) systems to clusters of off-the-shelf PCs, which are very cost-effective.
- Message-passing is a mature programming paradigm and widely accepted. It often provides an efficient match to the hardware. It is primarily used for the distributed memory systems, but can also be used on shared memory systems.

In these lectures we consider only message-passing for writing parallel programs.

Overhead present in parallel computing

- **Uneven load balance**: not all the processors can perform useful work at all time.
- **Overhead of synchronization.**
- **Overhead of communication**.
- Extra computation due to parallelization.

Due to the above overhead and that certain part of a sequential algorithm cannot be parallelized we may not achieve an optimal parallelization.

Parallelizing a sequential algorithm

- Identify the part(s) of a sequential algorithm that can be executed in parallel. This is the difficult part,
- Distribute the global work and data among P processors.

Bindings to MPI routines

MPI is a message-passing library where all the routines have corresponding C/C++-binding

MPI_Command_name

and Fortran-binding (routine names are in uppercase, but can also be in lower case)

MPI_COMMAND_NAME

The discussion in these slides focuses on the C++ binding.

Communicator

- A group of MPI processes with a name (context).
- Any process is identified by its rank. The rank is only meaningful within a particular communicator.
- By default communicator MPI_COMM_WORLD contains all the MPI processes.
- Mechanism to identify subset of processes.
- Promotes modular design of parallel libraries.

Some of the most important MPI functions

- MPI_ Init - initiate an MPI computation
- MPI_Finalize - terminate the MPI computation and clean up
- MPI_Comm_size - how many processes participate in a given MPI communicator?
- MPI_Comm_rank - which one am I? (A number between 0 and size-1.)
- MPI_Send - send a message to a particular process within an MPI communicator
- MPI_Recv - receive a message from a particular process within an MPI communicator
- MPI_reduce or MPI_Allreduce, send and receive messages

The first MPI C/C++ program

Let every process write ”Hello world” (oh not this program again!!) on the standard output.

```
using namespace std;
#include <mpi.h>
#include <iostream>
int main (int nargs, char* args[])
{
int numprocs, my_rank;
//   MPI initializations
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
cout << "Hello world, I have  rank " << my_rank <<
    " out of "
     << numprocs << endl;
//  End MPI
MPI_Finalize ();
```

The Fortran program

```
PROGRAM hello
INCLUDE "mpif.h"
INTEGER:: size, my_rank, ierr

CALL  MPI_INIT(ierr)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
WRITE(*,*)"Hello world, I've rank ",my_rank," out
    of ",size
CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

Note 1

The output to screen is not ordered since all processes are trying to write to screen simultaneously. It is then the operating system which opts for an ordering. If we wish to have an organized output, starting from the first process, we may rewrite our program as in the next example.

Ordered output with MPI_Barrier

```
int main (int nargs, char* args[])
{
 int numprocs, my_rank, i;
 MPI_Init (&nargs, &args);
 MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
 MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
 for (i = 0; i < numprocs; i++) {}
 MPI_Barrier (MPI_COMM_WORLD);
 if (i == my_rank) {
 cout << "Hello world, I have  rank " << my_rank <<
         " out of " << numprocs << endl;}
      MPI_Finalize ();
```

Note 2

Here we have used the *MPI_Barrier* function to ensure that that every process has completed its set of instructions in a particular order. A barrier is a special collective operation that does not allow the processes to continue until all processes in the communicator (here *MPI_COMM_WORLD*) have called *MPI_Barrier*. The barriers make sure that all processes have reached the same point in the code. Many of the collective operations like *MPI_ALLREDUCE* to be discussed later, have the same property; viz. no process can exit the operation until all processes have started. However, this is slightly more time-consuming since the processes synchronize between themselves as many times as there are processes. In the next Hello world example we use the send and receive functions in order to a have a synchronized action.

Ordered output with MPI_Recv and MPI_Send

```
.....
int numprocs, my_rank, flag;
MPI_Status status;
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
if (my_rank > 0)
MPI_Recv (&flag, 1, MPI_INT, my_rank-1, 100,
            MPI_COMM_WORLD, &status);
cout << "Hello world, I have  rank " << my_rank <<
    " out of "
<< numprocs << endl;
if (my_rank < numprocs-1)
MPI_Send (&my_rank, 1, MPI_INT, my_rank+1,
           100, MPI_COMM_WORLD);
MPI_Finalize ();
```

Note 3

The basic sending of messages is given by the function *MPI_SEND*, which in C/C++ is defined as

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)}
```

This single command allows the passing of any kind of variable, even a large array, to any group of tasks. The variable **buf** is the variable we wish to send while **count** is the number of variables we are passing. If we are passing only a single value, this should be 1. If we transfer an array, it is the overall size of the array. For example, if we want to send a 10 by 10 array, count would be $10 \times 10 = 100$ since we are actually passing 100 values.

Note 4

Once you have sent a message, you must receive it on another task. The function **MPI_RECV** is similar to the send call.

```
int MPI_Recv( void *buf, int count, MPI_Datatype
    datatype,
              int source,
              int tag, MPI_Comm comm, MPI_Status *
                  status )
```

The arguments that are different from those in *MPI_SEND* are **buf** which is the name of the variable where you will be storing the received data, **source** which replaces the destination in the send command. This is the return ID of the sender.
Finally, we have used **MPI_Status status;** where one can check if the receive was completed.
The output of this code is the same as the previous example, but now process 0 sends a message to process 1, which forwards it further to process 2, and so forth.

Integrating π

- The code example computes π using the trapezoidal rules.
- The trapezoidal rule

$$I = \int_a^b f(x)dx \approx$$

-

$$h\left(f(a)/2 + f(a+h) + f(a+2h) + \cdots + f(b-h) + f_b/2\right).$$

Dissection of trapezoidal rule with MPI_reduce

```
//     Trapezoidal rule and numerical integration
    usign MPI, example program6.cpp
using namespace std;
#include <mpi.h>
#include <iostream>

//     Here we define various functions called by
    the main program

double int_function(double );
double trapezoidal_rule(double , double , int ,
    double (*)(double));

//   Main function begins here
int main (int nargs, char* args[])
{
  int n, local_n, numprocs, my_rank;
  double a, b, h, local_a, local_b, total_sum,
      local_sum;
  double  time_start, time_end, total_time;
```

Dissection of trapezoidal rule with MPI_reduce

```
// MPI initializations
MPI_Init (&nargs, &args);
MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
time_start = MPI_Wtime();
// Fixed values for a, b and n
a = 0.0 ; b = 1.0;  n = 1000;
h = (b-a)/n;     // h is the same for all
    processes
local_n = n/numprocs;
// make sure n > numprocs, else integer division
    gives zero
// Length of each process' interval of
// integration = local_n*h.
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
```

Dissection of trapezoidal rule with MPI_reduce

```
  total_sum = 0.0;
  local_sum = trapezoidal_rule(local_a, local_b,
      local_n,
                                  &int_function);
  MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE,
                MPI_SUM, 0, MPI_COMM_WORLD);
  time_end = MPI_Wtime();
  total_time = time_end-time_start;
  if ( my_rank == 0) {
    cout << "Trapezoidal rule = " <<  total_sum <<
        endl;
    cout << "Time = " <<  total_time
         << " on number of processors: "  <<
             numprocs  << endl;
  }
  // End MPI
  MPI_Finalize ();
  return 0;
}  // end of main program
```

MPI_reduce

Here we have used

```
MPI_reduce( void *senddata, void* resultdata, int
    count,
      MPI_Datatype datatype, MPI_Op, int root,
          MPI_Comm comm)
```

The two variables *senddata* and *resultdata* are obvious, besides the fact that one sends the address of the variable or the first element of an array. If they are arrays they need to have the same size. The variable *count* represents the total dimensionality, 1 in case of just one variable, while MPI_Datatype defines the type of variable which is sent and received.
The new feature is MPI_Op. It defines the type of operation we want to do. In our case, since we are summing the rectangle contributions from every process we define MPI_Op = MPI_SUM. If we have an array or matrix we can search for the largest og smallest element by sending either MPI_MAX or MPI_MIN. If we want the location as well (which array element) we simply transfer MPI_MAXLOC or MPI_MINOC. If we want the product we write MPI_PROD.
MPI_Allreduce is defined as

```
MPI_Allreduce( void *senddata, void* resultdata,
    int count,
          MPI_Datatype datatype, MPI_Op, MPI_Comm
              comm)
```

Dissection of trapezoidal rule with MPI_reduce

We use MPI_reduce to collect data from each process. Note also the use of the function MPI_Wtime. The final functions are

```
//  this function defines the function to integrate
double int_function(double x)
{
  double value = 4./(1.+x*x);
  return value;
} // end of function to evaluate
```

Dissection of trapezoidal rule with MPI_reduce

```
//  this function defines the trapezoidal rule
double trapezoidal_rule(double a, double b, int n,
                        double (*func)(double))
{
  double trapez_sum;
  double fa, fb, x, step;
  int    j;
  step=(b-a)/((double) n);
  fa=(*func)(a)/2. ;
  fb=(*func)(b)/2. ;
  trapez_sum=0.;
  for (j=1; j <= n-1; j++){
    x=j*step+a;
    trapez_sum+=(*func)(x);
  }
  trapez_sum=(trapez_sum+fb+fa)*step;
  return trapez_sum;
}  // end trapezoidal_rule
```

Optimization and profiling

Till now we have not paid much attention to speed and possible optimization possibilities inherent in the various compilers. We have compiled and linked as

```
mpic++  -c  mycode.cpp
mpic++  -o  mycode.exe  mycode.o
```

For Fortran replace with mpif90. This is what we call a flat compiler option and should be used when we develop the code. It produces normally a very large and slow code when translated to machine instructions. We use this option for debugging and for establishing the correct program output because every operation is done precisely as the user specified it.
It is instructive to look up the compiler manual for further instructions

```
man mpic++  >  out_to_file
```

Optimization and profiling

We have additional compiler options for optimization. These may include procedure inlining where performance may be improved, moving constants inside loops outside the loop, identify potential parallelism, include automatic vectorization or replace a division with a reciprocal and a multiplication if this speeds up the code.

```
mpic++  -O3 -c  mycode.cpp
mpic++  -O3 -o  mycode.exe  mycode.o
```

This is the recommended option. **But you must check that you get the same results as previously**.

Optimization and profiling

It is also useful to profile your program under the development stage. You would then compile with

```
mpic++  -pg -O3 -c  mycode.cpp
mpic++  -pg -O3 -o  mycode.exe  mycode.o
```

After you have run the code you can obtain the profiling information via

```
gprof mycode.exe >  out_to_profile
```

When you have profiled properly your code, you must take out this option as it increases your CPU expenditure. For memory tests use **valgrind**, see `valgrind.org`. An excellent GUI is also Qt, with debugging facilities.

Optimization and profiling

Other hints

- avoid if tests or call to functions inside loops, if possible.
- avoid multiplication with constants inside loops if possible

Bad code

```
for i = 1:n
    a(i) = b(i) +c*d
    e = g(k)
end
```

Better code

```
temp = c*d
for i = 1:n
    a(i) = b(i) + temp
end
e = g(k)
```

Monte Carlo integration: Acceptance-Rejection Method

This is a rather simple and appealing method after von Neumann. Assume that we are looking at an interval $x \in [a, b]$, this being the domain of the Probability distribution function (PDF) $p(x)$. Suppose also that the largest value our distribution function takes in this interval is M, that is

$$p(x) \leq M \qquad x \in [a, b].$$

Then we generate a random number x from the uniform distribution for $x \in [a, b]$ and a corresponding number s for the uniform distribution between $[0, M]$. If

$$p(x) \geq s,$$

we accept the new value of x, else we generate again two new random numbers x and s and perform the test in the latter equation again.

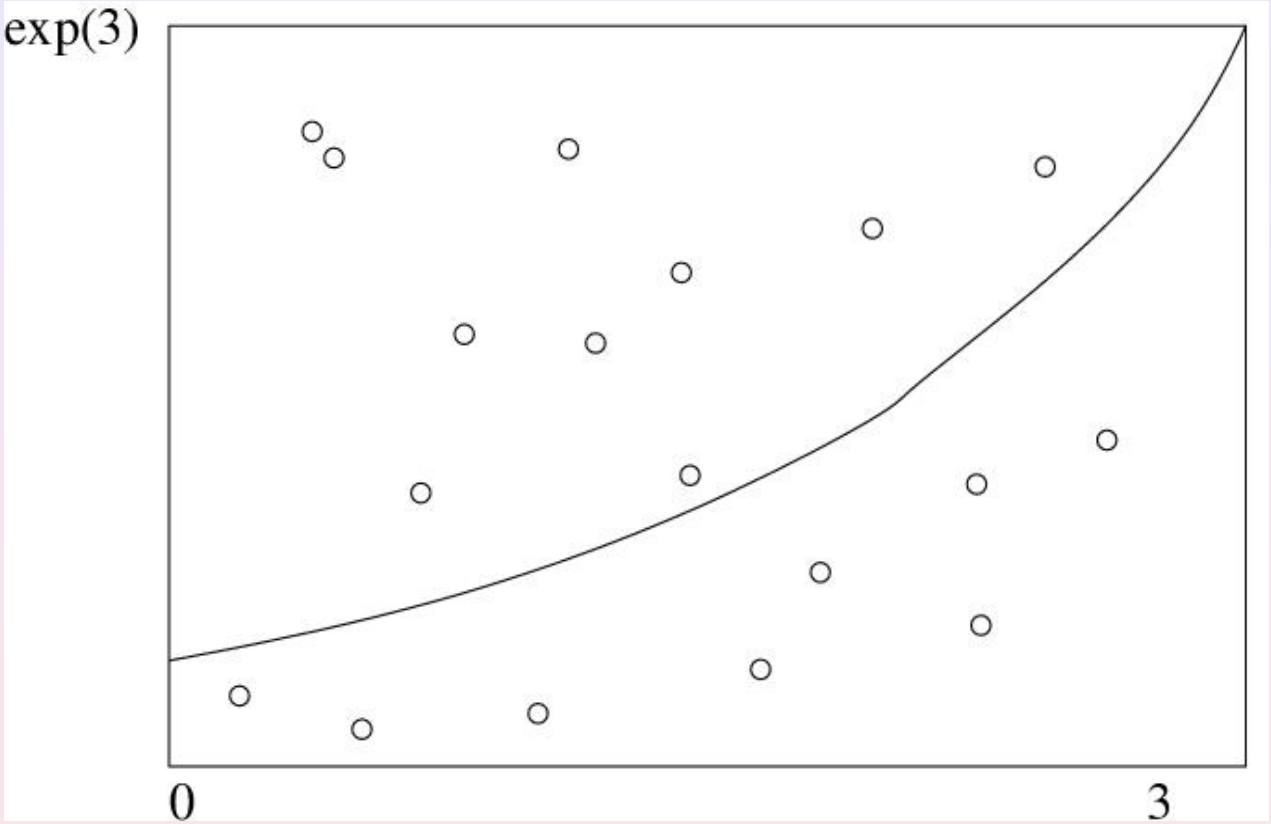
Acceptance-Rejection Method

As an example, consider the evaluation of the integral

$$I = \int_0^3 \exp(x) dx.$$

Obviously to derive it analytically is much easier, however the integrand could pose some more difficult challenges. The aim here is simply to show how to implent the acceptance-rejection algorithm using MPI. The integral is the area below the curve $f(x) = \exp(x)$. If we uniformly fill the rectangle spanned by $x \in [0, 3]$ and $y \in [0, \exp(3)]$, the fraction below the curve obatained from a uniform distribution, and multiplied by the area of the rectangle, should approximate the chosen integral. It is rather easy to implement this numerically, as shown in the following code.

Simple Plot of the Accept-Reject Method



algo: Acceptance-Rejection Method

```
//    Loop over Monte Carlo trials n
      integral =0.;
      for ( int i = 1;  i <= n; i++){
//    Finds a random value for x in the interval
    [0,3]
            x = 3*ran0(&idum);
//    Finds y-value between [0,exp(3)]
            y = exp(3.0)*ran0(&idum);
//    if the value of y at exp(x) is below the curve
    , we accept
            if ( y  < exp(x)) s = s+ 1.0;
//    The integral is area enclosed below the line f
    (x)=exp(x)
     }
//   Then we multiply with the area of the rectangle
     and
//   divide by the number of cycles
     Integral = 3.*exp(3.)*s/n
```

Acceptance-Rejection Method

Here it can be useful to split the program into subtasks

- A specific function which performs the Monte Carlo sampling
- A function which collects all data and performs statistical analysis and perhaps writes in parallel to file.

algo: Acceptance-Rejection Method

```
int main(int argc, char *argv[])
{
  // declarations ....
  // MPI initializations
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
  double time_start = MPI_Wtime();

  if (my_rank == 0 && argc <= 1) {
    cout << "Bad Usage: " << argv[0] <<
      " read also output file on same line" << endl
        ;
  }
  if (my_rank == 0 && argc > 1) {
    outfilename=argv[1];
    ofile.open(outfilename);
  }
```

algo: Acceptance-Rejection Method

```
    // Perform the integration
    integrate(MC_samples, integral);
    double time_end = MPI_Wtime();
    double total_time = time_end-time_start;
    if ( my_rank == 0) {
      cout << "Time = " <<  total_time  << " on
          number of processors: "  << numprocs  <<
          endl;
      ofile << setiosflags(ios::showpoint | ios::
          uppercase);
      ofile << setw(15) << setprecision(8) <<
          integral << endl;
      ofile.close();  // close output file
     }
    // End MPI
    MPI_Finalize ();
    return 0;
}  //  end of main function
```

algo: Acceptance-Rejection Method

```
void integrate(int number_cycles, double &Integral)
{
  double total_number_cycles;
  double variance, energy, error;
  double total_cumulative, total_cumulative_2,
      cumulative, cumulative_2;
  total_number_cycles = number_cycles*numprocs;
  //  Do the mc sampling
  cumulative = cumulative_2 = 0.0;
  total_cumulative = total_cumulative_2 = 0.0;
```

algo: Acceptance-Rejection Method

```
    mc_sampling(number_cycles, cumulative,
        cumulative_2);
    //  Collect data in total averages using MPI
        reduce
    MPI_Allreduce(&cumulative, &total_cumulative, 1,
        MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&cumulative_2, &total_cumulative_2,
         1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    Integral = total_cumulative/numprocs;
    variance = total_cumulative_2/numprocs-Integral*
        Integral;
    error=sqrt(variance/(total_number_cycles-1.0));
}  //  end of function integrate
```

What is OpenMP

- OpenMP provides high-level thread programming
- Multiple cooperating threads are allowed to run simultaneously
- Threads are created and destroyed dynamically in a fork-join pattern
  - An OpenMP program consists of a number of parallel regions
  - Between two parallel regions there is only one master thread
  - In the beginning of a parallel region, a team of new threads is spawned
  - The newly spawned threads work simultaneously with the master
  - thread
  - At the end of a parallel region, the new threads are destroyed

Getting started, things to remember

- Remember the header file **#include** $<$ *omp.h* $>$
- Insert compiler directives (**#pragma omp...** in C/C++ syntax), possibly also some OpenMP library routines
- Compile
  - For example, **c++ -fopenmp code.cpp**
- Execute
  - Remember to assign the environment variable **OMP NUM THREADS**
  - It specifies the total number of threads inside a parallel region, if not otherwise overwritten

General code structure

```
#include <omp.h>
main ()
{
int var1, var2, var3;
/* serial code */
/* ... */
/* start of a parallel region */
#pragma omp parallel private(var1, var2) shared(var3)
{
/* ... */
}
/* more serial code */
/* ... */
/* another parallel region */
#pragma omp parallel
{
/* ... */
}
}
```

Parallel region

- A parallel region is a block of code that is executed by a team of threads
- The following compiler directive creates a parallel region #pragma omp parallel ...
- Clauses can be added at the end of the directive
- Most often used clauses:
  - **default(shared)** or **default(none)**
  - **public(list of variables)**
  - **private(list of variables)**

Hello world

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
int th_id, nthreads;
#pragma omp parallel private(th_id) shared(nthreads)
{
th_id = omp_get_thread_num();
printf("Hello World from thread %d\n", th_id);
#pragma omp barrier
if ( th_id == 0 ) {
nthreads = omp_get_num_threads();
printf("There are %d threads\n",nthreads);
}
}
return 0;
}
```

Important OpenMP library routines

- **int omp get num threads ()**, returns the number of threads inside a parallel region
- **int omp get thread num ()**, returns the a thread for each thread inside a parallel region
- **void omp set num threads (int)**, sets the number of threads to be used
- **void omp set nested (int)**, turns nested parallelism on/off

Parallel for loop

- Inside a parallel region, the following compiler directive can be used to parallelize a for-loop: **#pragma omp for**
- Clauses can be added, such as
  - **schedule(static, chunk size)**
  - **schedule(dynamic, chunk size) (non-determinis**
  - **schedule(guided, chunk size)** (non-deterministic allocation)
  - **schedule(runtime)**
  - **private(list of variables)**
  - **reduction(operator:variable)**
  - **nowait**

```
#include <omp.h>
#define CHUNKSIZE 100
#define N
1000
main ()
{
int i, chunk;
float a[N], b[N], c[N];
for (i=0; i < N; i++)
a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
{
#pragma omp for schedule(dynamic,chunk)
for (i=0; i < N; i++)
c[i] = a[i] + b[i];
} /* end of parallel region */
}
```

More on Parallel for loop

- The number of loop iterations can not be non-deterministic; break, return, exit, goto not allowed inside the for-loop
- The loop index is private to each thread
- A reduction variable is special
  - During the for-loop there is a local private copy in each thread
  - At the end of the for-loop, all the local copies are combined together by the reduction operation
- Unless the nowait clause is used, an implicit barrier synchronization will be added at the end by the compiler
- **#pragma omp parallel and #pragma omp for** can be combined into **#pragma omp parallel for**

Inner product

$$\sum_{i=0}^{n-1} a_i b_i$$

```
int i;
double sum = 0.;
/* allocating and initializing arrays */
/* ... */
#pragma omp parallel for default(shared) private(i)
reduction(+:sum)
for (i=0; i<N; i++)
sum += a[i]*b[i];
}
```

Different threads do different tasks independently, each section is executed by one thread.

```
#pragma omp parallel
{
#pragma omp sections
{
#pragma omp section
funcA ();
#pragma omp section
funcB ();
#pragma omp section
funcC ();
}
}
```

Single execution

- **#pragma omp single ...**
  - code executed by one thread only, no guarantee which thread
  - an implicit barrier at the end
- **#pragma omp master ...**
  - code executed by the master thread, guaranteed
  - no implicit barrier at the end

Coordination and synchronization

- **#pragma omp barrier**, synchronization, must be encountered by all threads in a team (or none)
- **#pragma omp ordered a block of codes** , another form of synchronization (in sequential order)
- **#pragma omp critical a block of codes**
- **#pragma omp atomic single assignment statement** more efficient than **#pragma omp critical**

Data scope

- OpenMP data scope attribute clauses:
  - **shared**
  - **private**
  - **firstprivate**
  - **lastprivate**
  - **reduction**
- Purposes:
  - define how and which variables are transferred to a parallel region (and back)
  - define which variables are visible to all threads in a parallel region, and which variables are privately allocated to each thread

Some remarks

- When entering a parallel region, the **private** clause ensures each thread having its own new variable instances. The new variables are assumed to be uninitialized.
- A shared variable exists in only one memory location and all threads can read and write to that address. It is the programmer's responsibility to ensure that multiple threads properly access a shared variable.
- The **firstprivate** clause combines the behavior of the private clause with automatic initialization.
- The **lastprivate** clause combines the behavior of the private clause with a copy back (from the last loop iteration or section) to the original variable outside the parallel region.

Parallelizing nested for-loops

- Serial code

```
for (i=0; i<100; i++)
for (j=0; j<100; j++)
a[i][j] = b[i][j] + c[i][j]
```

- Parallelization

```
#pragma omp parallel for private(j)
for (i=0; i<100; i++)
for (j=0; j<100; j++)
a[i][j] = b[i][j] + c[i][j]
```

- Why not parallelize the inner loop? to save overhead of repeated thread forks-joins
- Why must **j** be private? To avoid race condition among the threads

Nested parallelism

When a thread in a parallel region encounters another parallel construct, it may create a new team of threads and become the master of the new team.

```
#pragma omp parallel num_threads(4)
{
/* .... */
#pragma omp parallel num_threads(2)
{
//
}
}
```

Parallel tasks

```
#pragma omp task
#pragma omp parallel shared(p_vec) private(i)
{
#pragma omp single
{
for (i=0; i<N; i++) {
double r = random_number();
if (p_vec[i] > r) {
#pragma omp task
do_work (p_vec[i]);
}
}
}
}
```

Common mistakes

Race condition

```
int nthreads;
#pragma omp parallel shared(nthreads)
{
nthreads = omp_get_num_threads();
}
```

Deadlock

```
#pragma omp parallel
{
...
#pragma omp critical
{
...
#pragma omp barrier
}
}
```

Matrix-matrix multiplication

```
# include <cstdlib>
# include <iostream>
# include <cmath>
# include <ctime>
# include <omp.h>

using namespace std;

// Main function
int main ( )
{
// brute force coding of arrays
  double a[500][500];
  double angle;
  double b[500][500];
  double c[500][500];
  int i;
  int j;
  int k;
```

Matrix-matrix multiplication

```
  int n = 500;
  double pi = acos(-1.0);
  double s;
  int thread_num;
  double wtime;

  cout << "\n";
  cout << "  C++/OpenMP version\n";
  cout << "  Compute matrix product C = A * B.\n";

  thread_num = omp_get_max_threads ( );

//
//  Loop 1: Evaluate A.
//
  s = 1.0 / sqrt ( ( double ) ( n ) );

  wtime = omp_get_wtime ( );
```

Matrix-matrix multiplication

```
# pragma omp parallel shared ( a, b, c, n, pi, s )
private ( angle, i, j, k )
{
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
    for ( j = 0; j < n; j++ )
    {
      angle = 2.0 * pi * i * j / ( double ) n;
      a[i][j] = s * ( sin ( angle ) + cos ( angle ) );
    }
  }
//
//  Loop 2: Copy A into B.
//
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
    for ( j = 0; j < n; j++ )
    {
      b[i][j] = a[i][j];
    }
```

Matrix-matrix multiplication

```
//  Loop 3: Compute C = A * B.
//
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
    for ( j = 0; j < n; j++ )
    {
      c[i][j] = 0.0;
      for ( k = 0; k < n; k++ )
      {
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
    }
  }
}
  wtime = omp_get_wtime ( ) - wtime;
  cout << "  Elapsed seconds = " << wtime << "\n";
  cout << "  C(100,100)  = " << c[99][99] << "\n";
//
//  Terminate.
//
  cout << "\n";
```

Matrix handling, Jacobi's method

- Parallel Jacobi Algorithm
- Different data distribution schemes
- Row-wise distribution
- Column-wise distribution
- Other alternatives not discussed here: Cyclic shifting

Matrix handling, Jacobi's method

- Direct solvers such as Gauss elimination and LU decomposition
- Iterative solvers such Basic iterative solvers, Jacobi, Gauss-Seidel, Successive over-relaxation
- Other iterative methods such as Krylov subspace methods with Generalized minimum residual (GMRES) and Conjugate gradient etc

Matrix handling, Jacobi's method

It is a simple method for solving

$$\hat{A}\mathbf{x} = \mathbf{b},$$

where $\hat{A}$ is a matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors. The vector $\mathbf{x}$ is the unknown.
It is an iterative scheme where after $k+1$ iterations we have

$$\mathbf{x}^{(k+1)} = \hat{D}^{-1}(\mathbf{b} - (\hat{L} + \hat{U})\mathbf{x}^{(k)}),$$

with $\hat{A} = \hat{D} + \hat{U} + \hat{L}$ and $\hat{D}$ being a diagonal matrix, $\hat{U}$ an upper triangular matrix and $\hat{L}$ a lower triangular matrix.

Matrix handling, Jacobi's method

Shared memory or distributed memory:

- Shared-memory parallelization very straightforward
- Consider distributed memory machine using MPI

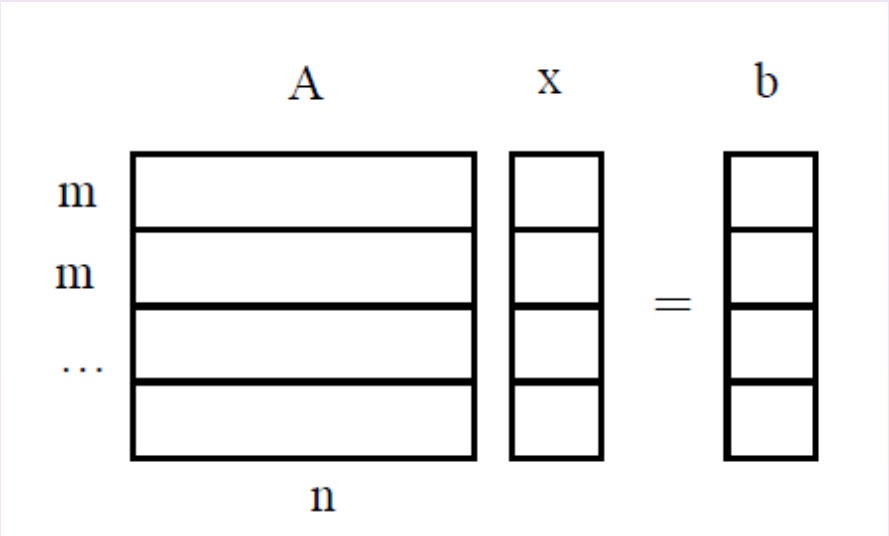
Questions to answer in parallelization:

- Data distribution (data locality)
- How to distribute coefficient matrix among CPUs?
- How to distribute vector of unknowns?
- How to distribute RHS?
- Communication: What data needs to be communicated?

Want to:

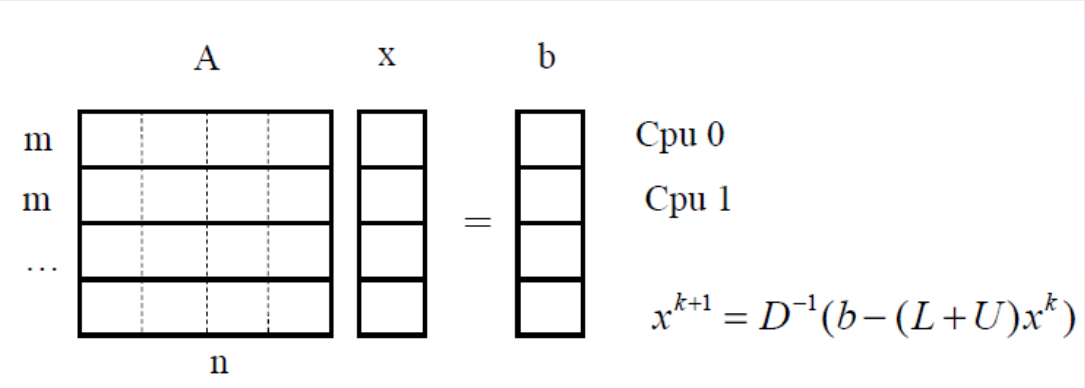
- Achieve data locality
- Minimize the number of communications
- Overlap communications with computations
- Load balance

Row-wise distribution



- Assume dimension of matrix $n \times n$ can be divided by number of CPUs P, $m = n/P$
- Blocks of m rows of coefficient matrix distributed to different CPUs;
- Vector of unknowns and RHS distributed similarly

Data to be communicated



- Already have all columns of matrix $\hat{A}$ on each CPU;
- Only part of vector **x** is available on a CPU; Cannot carry out matrix vector multiplication directly;
- Need to communicate the vector **x** in the computations.

How to Communicate Vector **x**?

- Gather partial vector **x** on each CPU to form the whole vector; Then matrix-vector multiplication on different CPUs proceed independently.
- Need MPI_Allgather() function call All *localdata* are collected in *olddata*.
- Simple to implement, but
- A lot of communications
- Does not scale well for a large number of processors.

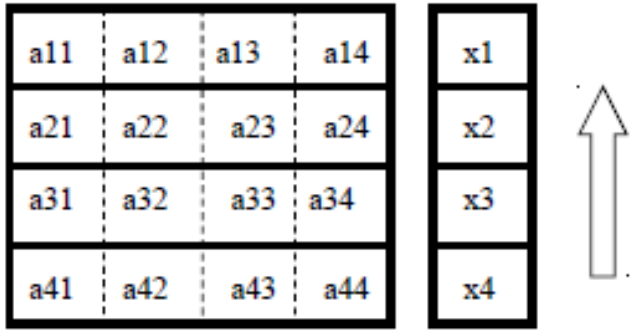
```
 MPI_Allgather( void *localdata,
int dim, void *olddata, int dim, MPI_Datatype datat
```

How to Communicate Vector **x**?

- Another method: Cyclic shift
- Shift partial vector **x** upward at each step;
- Do partial matrix-vector multiplication on each CPU at each step;
- After P steps (P is the number of CPUs), the overall matrix-vector multiplication is complete.
- Each CPU needs only to communicate with neighboring CPUs
- Provides opportunities to overlap communication with computations

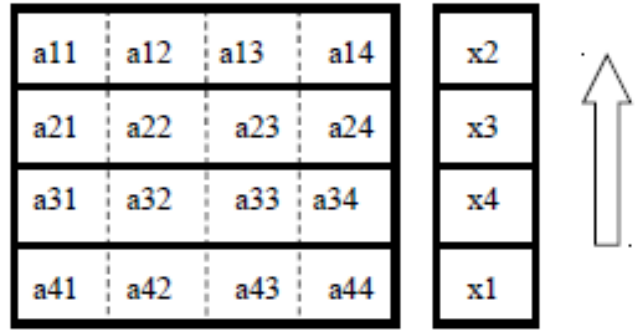
Row-wise algo

(1)



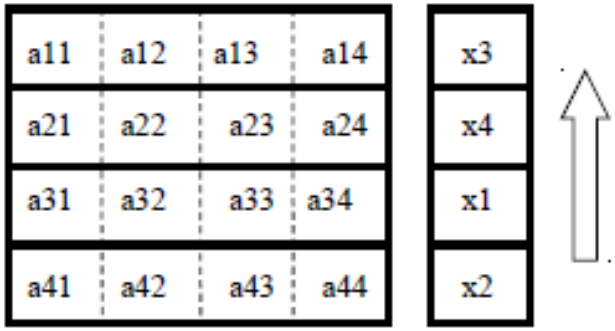
a11*x1 + a12*x2 + a13*x3 + a14*x4 Cpu 0
a21*x1 + a22*x2 + a23*x3 + a24*x4 Cpu 1
a31*x1 + a32*x2 + a33*x3 + a34*x4 Cpu 2
a41*x1 + a42*x2 + a43*x3 + a44*x4 Cpu 3

(2)



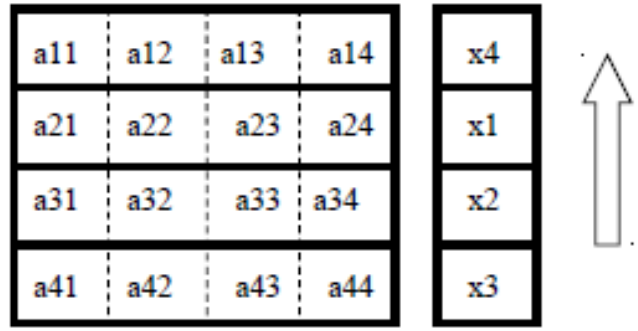
a11*x1 + a12*x2 + a13*x3 + a14*x4
a21*x1 + a22*x2 + a23*x3 + a24*x4
a31*x1 + a32*x2 + a33*x3 + a34*x4
a41*x1 + a42*x2 + a43*x3 + a44*x4

(3)



a11*x1 + a12*x2 + a13*x3 + a14*x4
a21*x1 + a22*x2 + a23*x3 + a24*x4
a31*x1 + a32*x2 + a33*x3 + a34*x4
a41*x1 + a42*x2 + a43*x3 + a44*x4

(4)



a11*x1 + a12*x2 + a13*x3 + a14*x4
a21*x1 + a22*x2 + a23*x3 + a24*x4
a31*x1 + a32*x2 + a33*x3 + a34*x4
a41*x1 + a42*x2 + a43*x3 + a44*x4

Overlap Communications with Computations

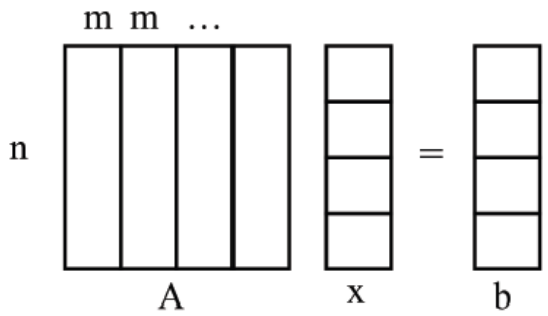
Communications

- Each CPU needs to send its own partial vector **x** to upper neighboring CPU;
- Each CPU needs to receive data from lower neighboring CPU

Overlap communications with computations: Each CPU does the following:

- Post non-blocking requests to send data to upper neighbor to to receive data from lower neighbor; This returns immediately
- Do partial computation with data currently available;
- Check non-blocking communication status; wait if necessary;
- Repeat above steps

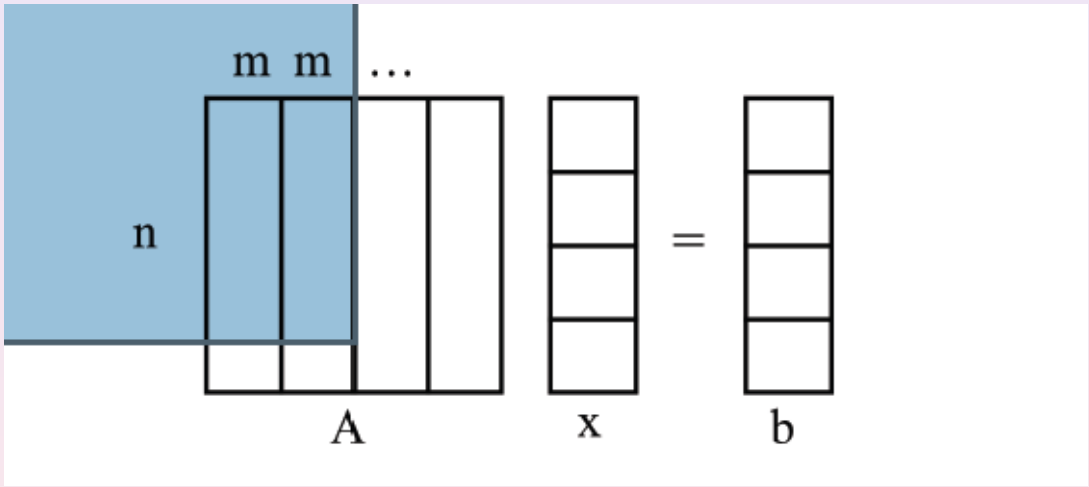
Column-wise distribution



$$x^{k+1} = D^{-1}(b-(L+U)x^k)$$

- Blocks of m columns of matrix $\hat{A}$ are distributed among the different P CPUs
- Blocks of m rows of vectors **x** and **b**are distributed to different CPUs

Data to be communicated



- Have already coefficient matrix data of m columns and a block of m rows of vector $\mathbf{x}$.
- A partial $\hat{A}\mathbf{x}$ can be computed on each CU independently.
- Need communication to get the whole $\hat{A}\mathbf{x}$ using MPI_Allreduce.

Libraries

If your needs (common in most problems) include handling of large arrays and linear algebra problem, we do not recommend to write your own vector-matrix or more general array handling class. It is easy to make errors. Use libraries like Armadillo (recommended). Use also well-tested libraries like Lapack and Blas.

- For C++ programmers (recommended) you can use armadillo, a great C++ library for handling arrays and doing linear algebra.
- Armadillo provides a user friendly interface to lapack and blas functions. Below you will find an example of using the Blas function **DGEMM** for matrix-matrix multiplication.
- After having installed armadillo, compile with **c++ -O3 -o test.x test.cpp -lblas**.

Matrix-matrix multiplication

```
#include <cstdlib>
#include <ios>
#include <iostream>
#include <armadillo>
using namespace std;
using namespace arma;

/*Because fortran files don't have any header files
    ,
 *we need to declare the functions ourself.*/
extern "C"
{
    void dgemm_(char*, char*, int*, int*, int*,
        double*,
            double*, int*, double*, int*, double*,
                double*, int*);
}
```

Matrix-matrix multiplication

```
int main(int argc, char** argv)
{
    //Dimensions
    int n = atoi(argv[1]);
    int m = n;
    int p = m;

    /*Create random matrices
     * (note that older versions of armadillo uses
        "rand" instead of "randu") */
    srand(time(NULL));
    mat A(n, p);
    A.randu();
```

Matrix-matrix multiplication

```
// Pretty print, and pretty save, are as easy
    as the two following lines.
//      cout << A << endl;
//      A.save("A.mat", raw_ascii);
mat A_trans = trans(A);
mat B(p, m);
B.randu();
mat C(n, m);
//      cout << B << endl;
//      B.save("B.mat", raw_ascii);
```

Matrix-matrix multiplication

```
//    ARMADILLO  TEST
cout << "Starting armadillo multiplication\n";
//Simple wall_clock timer is a part of
    armadillo.
wall_clock timer;
timer.tic();
C = A*B;
double num_sec = timer.toc();
cout << "-- Finished in " << num_sec << "
    seconds.\n\n";
```

Matrix-matrix multiplication

```
C = zeros<mat> (n, m);
cout << "Starting blas multiplication.\n";
{
    char trans = 'N';
    double alpha = 1.0;
    double beta = 0.0;
    int _numRowA = A.n_rows;
    int _numColA = A.n_cols;
    int _numRowB = B.n_rows;
    int _numColB = B.n_cols;
    int _numRowC = C.n_rows;
    int _numColC = C.n_cols;
    int lda = (A.n_rows >= A.n_cols) ? A.n_rows
        : A.n_cols;
    int ldb = (B.n_rows >= B.n_cols) ? B.n_rows
        : B.n_cols;
    int ldc = (C.n_rows >= C.n_cols) ? C.n_rows
        : C.n_cols;
```

Matrix-matrix multiplication, calling DGEMM

```
        dgemm_(&trans, &trans, &_numRowA, &_numColB
            , &_numColA, &alpha,
                A.memptr(), &lda, B.memptr(), &ldb,
                    &beta, C.memptr(), &ldc);
}
```