Programming Techniques (Master in Astrophysics - ULL)

Ángel de Vicente

September 2019

These are the notes to be followed in the course “Programming Techniques” (“Técnicas de Programación”) of the Master in Astrophysics (ULL).

The objective of the course is to learn some programming techniques necessary in many scientific codes. In particular, we will study about dynamic data structures and MPI parallel programming with Fortran.

All the material taught in the course will be motivated by a sample N-body problem. We will start by developing a naïve and serial code; then we will study about the much more efficient Barnes-Hut algorithm and in order to implement this algorithm we will have to learn about dynamic data structures, recursion, trees, lists, etc. Once a Barnes-Hut serial implementation is finished, we will focus on learning the basics of parallel programming, in this case using the MPI library, and on parallelizing the Barnes-Hut version of our code.

We will also touch briefly on code debugging and profiling (both in serial and in parallel codes) and on other parallel programming models (OpenMP, CUDA, OpenACC) and how to use them together with MPI.

Contents

PART I

NAÏVE N-BODY SOLUTION WITH FORTRAN

Chapter 1

N-body problem formulation

1.1 Introduction

A nice introduction to the N-Body problem can be found in the documentation for the XStar[1] code: "The "N-Body problem" is the problem of trying to find how n objects will move under one of the physical forces, such as gravity." [1]. The XStar code (written in the C language) solves what we will cover in this course and more, except for the parallelization part. Figure 1.1 shows an example screenshot of a simulation run with XStar.

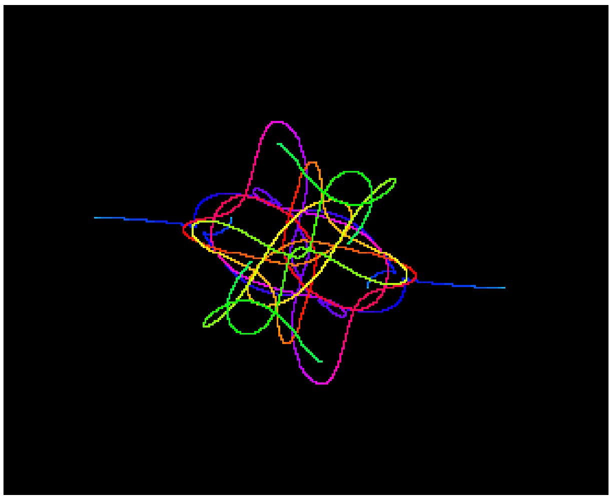
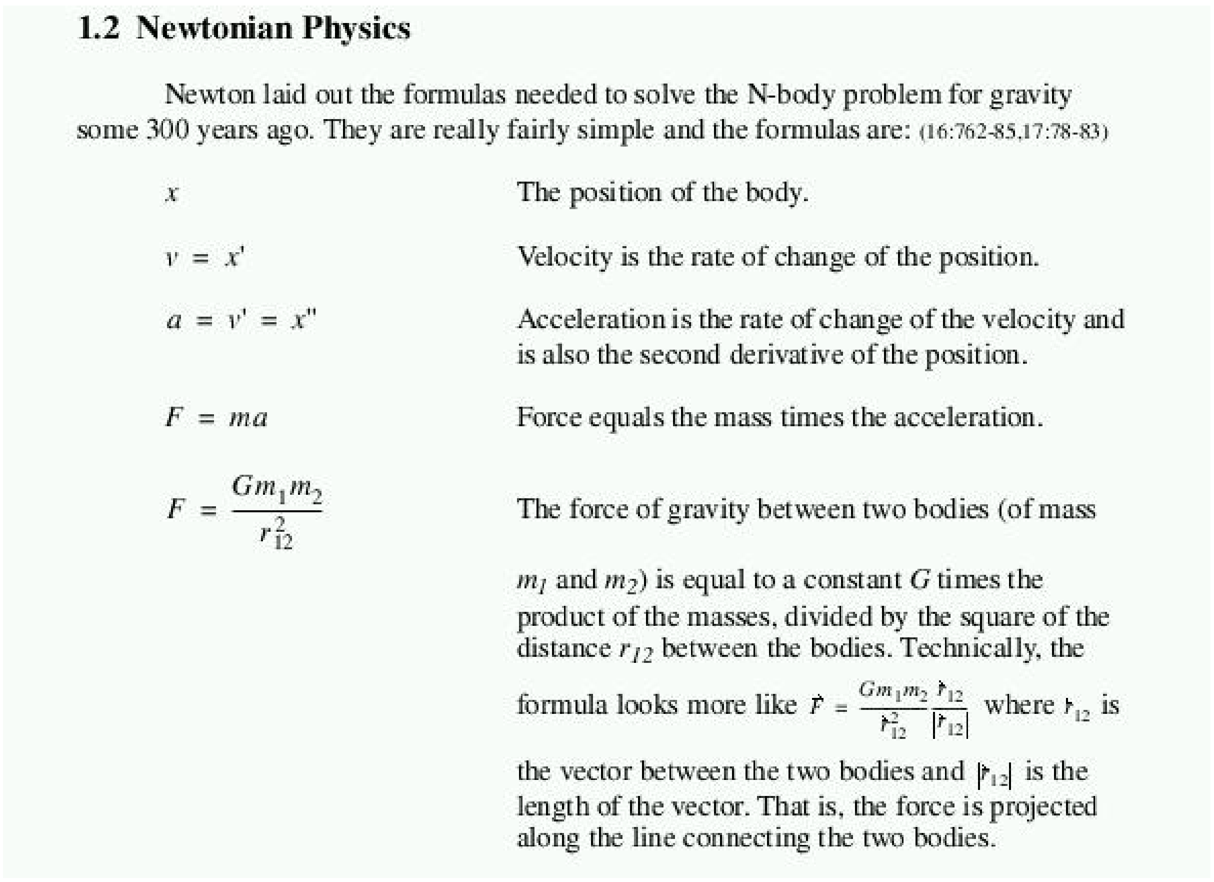


Figure 1.1: Sample XStar configuration simulation (from http://www.schlitt.net/xstar/screen_shots.html)

1.2 N-body equations

The equations to solve the N-body problem are very simple. See, for example, sections 1.2 and 1.3 of the Xstar guide (http://www.schlitt.net/xstar/n-body/nb-8.html), and page 65 of "Moving Stars Around".

1. http://www.schlitt.net/xstar/

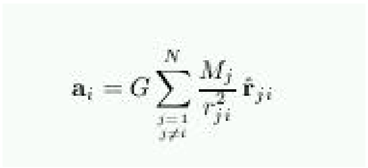


1.2 Newtonian Physics

Newton laid out the formulas needed to solve the N-body problem for gravity some 300 years ago. They are really fairly simple and the formulas are: (16:762-85,17:78-83)

x	The position of the body.
$v = x'$	Velocity is the rate of change of the position.
$a = v' = x''$	Acceleration is the rate of change of the velocity and is also the second derivative of the position.
$F = ma$	Force equals the mass times the acceleration.
$F = \frac{Gm_1m_2}{r_{12}^2}$	The force of gravity between two bodies (of mass m_1 and m_2) is equal to a constant G times the product of the masses, divided by the square of the distance r_{12} between the bodies. Technically, the formula looks more like $\vec{F} = \frac{Gm_1m_2}{\vec{r}_{12}^2}\frac{\vec{r}_{12}}{\lvert\vec{r}_{12}\rvert}$ where $\vec{r}_{12}$ is the vector between the two bodies and $\lvert\vec{r}_{12}\rvert$ is the length of the vector. That is, the force is projected along the line connecting the two bodies.

Figure 1.2: p.8 Guía Xstar

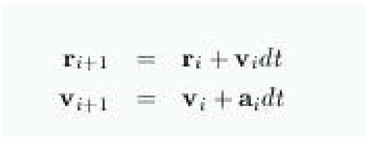


$$\mathbf{a}_i = G\sum_{\substack{j=1\\ j\neq i}}^{N} \frac{M_j}{r_{ji}^2}\hat{\mathbf{r}}_{ji}$$

Figure 1.3: p. 65 “Moving Stars Around”

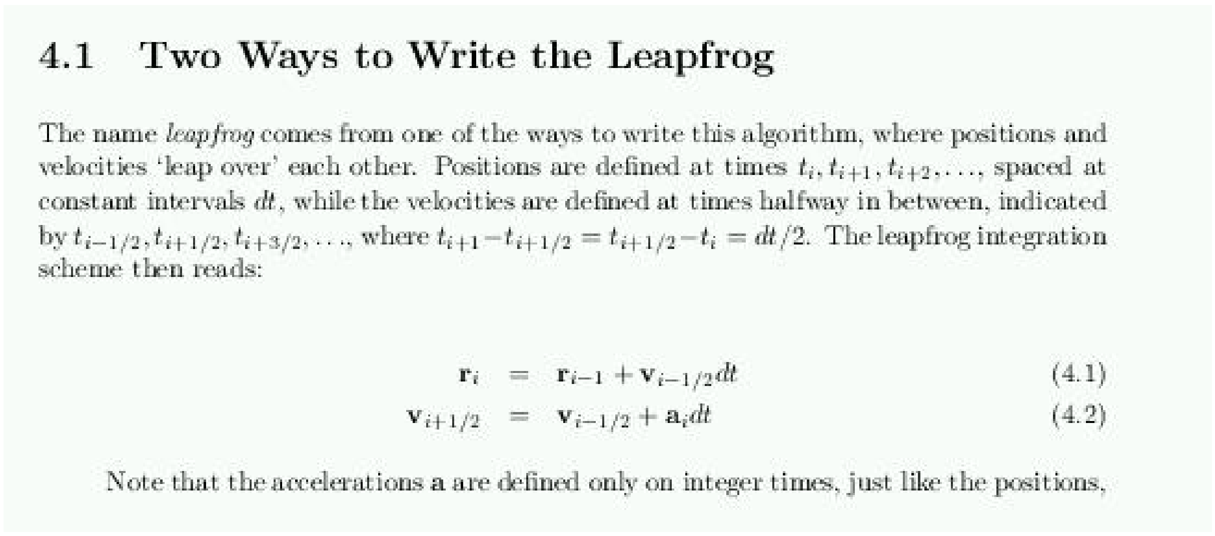
1.3 Time integration

To integrate in time, one can use several methods. The simplest one, of order 1 is Forward-Euler, as can be seen in page 24 of "Moving Stars Around". The one we are going to use here is of order 2: the leapfrom algorithm (see pages 55-56 of "Moving Stars Around").



$$\begin{aligned} \mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt \\ \mathbf{v}_{i+1} &= \mathbf{v}_i + \mathbf{a}_i dt \end{aligned}$$

Figure 1.4: p.24 "Moving Stars Around"



4.1 Two Ways to Write the Leapfrog

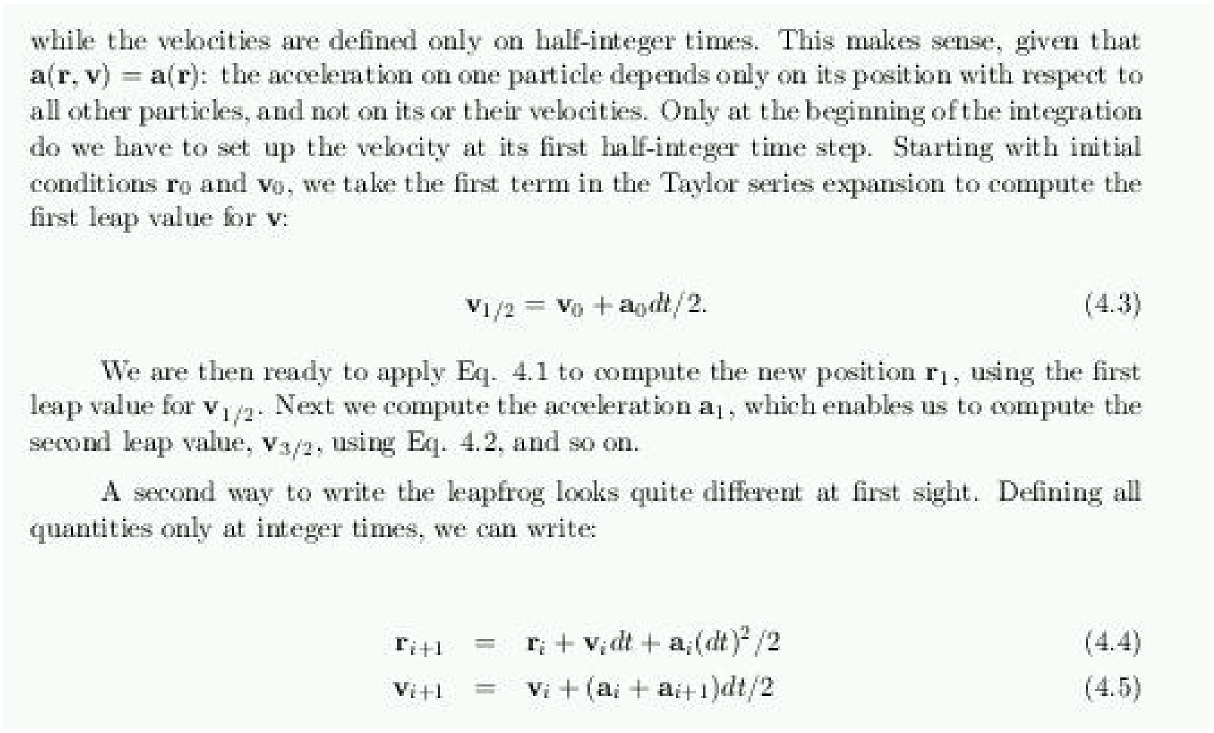
The name *leapfrog* comes from one of the ways to write this algorithm, where positions and velocities 'leap over' each other. Positions are defined at times $t_i, t_{i+1}, t_{i+2}, \ldots$, spaced at constant intervals dt, while the velocities are defined at times halfway in between, indicated by $t_{i-1/2}, t_{i+1/2}, t_{i+3/2}, \ldots$, where $t_{i+1} - t_{i+1/2} = t_{i+1/2} - t_i = dt/2$. The leapfrog integration scheme then reads:

$$\mathbf{r}_i = \mathbf{r}_{i-1} + \mathbf{v}_{i-1/2} dt \quad (4.1)$$

$$\mathbf{v}_{i+1/2} = \mathbf{v}_{i-1/2} + \mathbf{a}_i dt \quad (4.2)$$

Note that the accelerations $\mathbf{a}$ are defined only on integer times, just like the positions,

Figure 1.5: p.55 "Moving Stars Around"



while the velocities are defined only on half-integer times. This makes sense, given that $\mathbf{a}(\mathbf{r},\mathbf{v}) = \mathbf{a}(\mathbf{r})$: the acceleration on one particle depends only on its position with respect to all other particles, and not on its or their velocities. Only at the beginning of the integration do we have to set up the velocity at its first half-integer time step. Starting with initial conditions $\mathbf{r}_0$ and $\mathbf{v}_0$, we take the first term in the Taylor series expansion to compute the first leap value for $\mathbf{v}$:

$$\mathbf{v}_{1/2} = \mathbf{v}_0 + \mathbf{a}_0 dt/2. \tag{4.3}$$

We are then ready to apply Eq. 4.1 to compute the new position $\mathbf{r}_1$, using the first leap value for $\mathbf{v}_{1/2}$. Next we compute the acceleration $\mathbf{a}_1$, which enables us to compute the second leap value, $\mathbf{v}_{3/2}$, using Eq. 4.2, and so on.

A second way to write the leapfrog looks quite different at first sight. Defining all quantities only at integer times, we can write:

$$\begin{aligned} \mathbf{r}_{i+1} &= \mathbf{r}_i + \mathbf{v}_i dt + \mathbf{a}_i (dt)^2/2 && (4.4)\\ \mathbf{v}_{i+1} &= \mathbf{v}_i + (\mathbf{a}_i + \mathbf{a}_{i+1}) dt/2 && (4.5) \end{aligned}$$

Figure 1.6: p.56 “Moving Stars Around”

Chapter 2

Basic Fortran for first N-body implementation

2.1 Basic Fortran

In this chapter we will just cover the basic concepts of Fortran. Fortran is a complex language, so in this chapter we will only cover the surface of the language, only the necessary minimum to be able to write a first implementation of the N-body problem.

The following slides are taken from a 5-day intentive course on Fortran, given by the University of Liverpool.

THE UNIVERSITY *of* LIVERPOOL

Fortran 90 Programming

(5 Day Course)

—

Dr. A C Marshall (funded by JISC/NTI)

with acknowledgements to Steve Morgan and Lawrie Schonfelder.

Fortran 90 New features

Fortran 90 supports,

1. free source form;
2. array syntax and many more (array) intrinsics;
3. dynamic storage and pointers;
4. portable data types (`KIND`s);
5. derived data types and operators;
6. recursion;
7. `MODULE`s
   - □ procedure interfaces;
   - □ enhanced control structures;
   - □ user defined generic procedures;
   - □ enhanced I/O.

Example

Example Fortran 90 program:

```
MODULE Triangle_Operations
 IMPLICIT NONE
CONTAINS
 FUNCTION Area(x,y,z)
  REAL :: Area       ! function type
  REAL, INTENT( IN ) :: x, y, z
  REAL :: theta, height
  theta=ACOS((x**2+y**2-z**2)/(2.0*x*y))
  height=x*SIN(theta); Area=0.5*y*height
 END FUNCTION Area
END MODULE Triangle_Operations

PROGRAM Triangle
 USE Triangle_Operations
 IMPLICIT NONE
  REAL :: a, b, c, Area
  PRINT*, 'Welcome, please enter the&
         &lengths of the 3 sides.'
  READ*, a, b, c
  PRINT*,'Triangle''s area: ',Area(a,b,c)
END PROGRAM Triangle
```

Coding Style

It is recommended that the following coding convention is adopted:

- *always* use `IMPLICIT NONE`.
- Fortran 90 keywords, intrinsic functions and user defined entities should be in upper case,
- other user entities should be in lower case but may start with a capital letter.
- indentation should be 1 or 2 spaces and should be applied to the bodies of program units, control blocks, `INTERFACE` blocks, etc.
- the names of program units are always included in their `END` statements,
- argument keywords are always used for optional arguments,

Please note: In order that a program fits onto a slide these rules are sometimes relaxed here.

Source Form

Free source form:

- ☐ 132 characters per line;
- ☐ '!' comment initiator;
- ☐ '&' line continuation character;
- ☐ ';' statement separator;
- ☐ significant blanks.

Example,

```
PRINT*, "This line is continued &
        &On the next line"; END ! of program
```

Intrinsic Types

Fortran 90 has three broad classes of object type,

- character;
- boolean;
- numeric.

these give rise to six simple intrinsic types, known as default types,

```
CHARACTER          :: sex  ! letter
CHARACTER(LEN=12)  :: name ! string
LOGICAL            :: wed  ! married?
REAL               :: height
DOUBLE PRECISION   :: pi   ! 3.14...
INTEGER            :: age  ! whole No.
COMPLEX            :: val  ! x + iy
```

Literal Constants

A literal constant is an entity with a fixed value:

```
12345       ! INTEGER
1.0         ! REAL
-6.6E-06    ! REAL: -6.6*10**(-6)
.FALSE.     ! LOGICAL
.TRUE.      ! LOGICAL
"Mau'dib"   ! CHARACTER
'Mau''dib' ! CHARACTER
```

Note,

- ☐ there are only two `LOGICAL` values;
- ☐ `REAL`s contain a decimal point, `INTEGER`s do not,
- ☐ `REAL`s have an exponential form
- ☐ character literals delimited by `"` and `'`;
- ☐ two occurrences of the delimiter inside a string produce one occurrence on output;
- ☐ there is only a finite range of values that numeric literals can take.

Implicit Typing

Undeclared variables have an implicit type,

- □ if first letter is `I, J, K, L, M` or `N` then type is `INTEGER`;
- □ any other letter then type is `REAL`s.

Implicit typing is potentially very dangerous and should **always** be turned off by adding:

```
    IMPLICIT NONE
```

as the first line after any `USE` statements.

Consider,

```
    DO 30 I = 1.1000
      ...
 30 CONTINUE
```

in fixed format with implicit typing this declares a `REAL` variable `DO30I` and sets it to `1.1000` instead of performing a loop 1000 times!

Numeric and Logical Declarations

With `IMPLICIT NONE` variables must be declared. A simplified syntax follows,

$<$ *type* $>$ [,$<$ *attribute*-list $>$] ::$<$ *variable*-list $>$&
[=$<$ *value* $>$]

The following are all valid declarations,

```
REAL                       :: x
INTEGER                    :: i, j
LOGICAL, POINTER           :: ptr
REAL, DIMENSION(10,10)     :: y, z
INTEGER                    :: k = 4
```

The `DIMENSION` attribute declares an array (10×10).

Constants (Parameters)

Symbolic constants, oddly known as *parameters* in Fortran, can easily be set up either in an attributed declaration or parameter statement,

```
REAL, PARAMETER :: pi = 3.14159
CHARACTER(LEN=*), PARAMETER :: &
                  son = 'bart', dad = "Homer"
```

`CHARACTER` constants can assume their length from the associated literal (`LEN=*`).

Parameters should be used:

- ☐ if it is known that a variable will only take one value;
- ☐ for legibility where a 'magic value' occurs in a program such as π;
- ☐ for maintainability when a 'constant' value could feasibly be changed in the future.

Initialisation

Variables can be given initial values:

- □ can use *initialisation expressions*,
- □ may only contain PARAMETERs or literals.

```
REAL              :: x, y =1.0D5
INTEGER           :: i = 5, j = 100
CHARACTER(LEN=5) :: light = 'Amber'
CHARACTER(LEN=9) :: gumboot = 'Wellie'
LOGICAL  :: on = .TRUE., off = .FALSE.
REAL, PARAMETER :: pi = 3.141592
REAL, PARAMETER :: radius = 3.5
REAL :: circum = 2 * pi * radius
```

gumboot will be padded, to the right, with blanks.

In general, intrinsic functions *cannot* be used in initialisation expressions, the following can be: REPEAT, RESHAPE, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER, TRIM, LBOUND, UBOUND, SHAPE, SIZE, KIND, LEN, BIT_SIZE and numeric inquiry intrinsics, for, example, HUGE, TINY, EPSILON.

Expressions

Each of the three broad type classes has its own set of intrinsic (in-built) operators, for example, +, // and `.AND.`,

The following are valid expressions,

- ☐ `NumBabiesBorn+1` — numeric valued
- ☐ `"Ward "//Ward` — character valued
- ☐ `TimeSinceLastBirth .GT. MaxTimeTwixtBirths` — logical valued

Expressions can be used in many contexts and can be of any intrinsic type.

Assignment

Assignment is defined between all expressions of the same type:

Examples,

```
    a = b
    c = SIN(.7)*12.7  ! SIN in radians
    name = initials//surname
    bool = (a.EQ.b.OR.c.NE.d)
```

The LHS is an object and the RHS is an expression.

Intrinsic Numeric Operations

The following operators are valid for numeric expressions,

- `**` exponentiation, dyadic operator, for example, `10**2`, (evaluated right to left);
- `*` and `/` multiply and divide, dyadic operators, for example, `10*7/4`;
- `+` and `-` plus and minus or add and subtract, monadic and dyadic operators, for example, `10+7-4` and `-3`;

Can be applied to literals, constants, scalar and array objects. The only restriction is that the RHS of `**` must be scalar.

Example,

```
a = b - c
f = -3*6/5
```

Relational Operators

The following *relational operators* deliver a `LOGICAL` result when combined with numeric operands,

`.GT.`	`>`	greater than
`.GE.`	`>=`	greater than or equal to
`.LE.`	`<=`	less than or equal to
`.LT.`	`<`	less than
`.NE.`	`/=`	not equal to
`.EQ.`	`==`	equal to

For example,

```
bool = i .GT. j
boule = i > j
IF (i .EQ. j) c = D
IF (i == j)   c = D
```

When using real-valued expressions (which are approximate) `.EQ.` and `.NE.` have no real meaning.

```
REAL :: Tol = 0.0001
IF (ABS(a-b) .LT. Tol) same = .TRUE.
```

Intrinsic Logical Operations

A `LOGICAL` or boolean expression returns a `.TRUE.` / `.FALSE.` result. The following are valid with `LOGICAL` operands,

- ☐ `.NOT.` — `.TRUE.` if operand is `.FALSE.`.
- ☐ `.AND.` — `.TRUE.` if both operands are `.TRUE.`;
- ☐ `.OR.` — `.TRUE.` if at least one operand is `.TRUE.`;
- ☐ `.EQV.` — `.TRUE.` if both operands are the same;
- ☐ `.NEQV.` — `.TRUE.` if both operands are different.

For example, if `T` is `.TRUE.` and `F` is `.FALSE.`

- ☐ `.NOT. T` is `.FALSE.`, `.NOT. F` is `.TRUE.`.
- ☐ `T .AND. F` is `.FALSE.`, `T .AND. T` is `.TRUE.`.
- ☐ `T .OR. F` is `.TRUE.`, `F .OR. F` is `.FALSE.`.
- ☐ `T .EQV. F` is `.FALSE.`, `F .EQV. F` is `.TRUE.`.
- ☐ `T .NEQV. F` is `.TRUE.`, `F .NEQV. F` is `.FALSE.`.

Control Flow

Control constructs allow the normal sequential order of execution to be changed.

Fortran 90 supports:

- ☐ conditional execution statements and constructs, (`IF` ... and `IF` ... `THEN` ... `ELSE` ... `END IF`);
- ☐ loops, (`DO` ... `END DO`);
- ☐ multi-way choice construct, (`SELECT CASE`);

`IF` Statement

Example,

```
    IF (bool_val) A = 3
```

The basic syntax is,

`IF(`< *logical-expression* >`)`< *exec-stmt* >

If < *logical-expression* > evaluates to `.TRUE.` then execute < *exec-stmt* > otherwise do not.

For example,

```
    IF (x .GT. y) Maxi = x
```

means 'if `x` is greater than `y` then set `Maxi` to be equal to the value of `x`'.

More examples,

```
    IF (a*b+c <= 47) Boolie = .TRUE.
    IF (i .NE. 0 .AND. j .NE. 0) k = 1/(i*j)
    IF (i /= 0 .AND. j /= 0) k = 1/(i*j) ! same
```

`IF ... THEN ... ELSE` Construct

The block-`IF` is a more flexible version of the single line `IF`. A simple example,

```
IF (i .EQ. 0) THEN
 PRINT*, "I is Zero"
ELSE
 PRINT*, "I is NOT Zero"
ENDIF
```

note the how indentation helps.

Can also have one or more `ELSEIF` branches:

```
IF (i .EQ. 0) THEN
 PRINT*, "I is Zero"
ELSE IF (i .GT. 0) THEN
 PRINT*, "I is greater than Zero"
ELSE
 PRINT*, "I must be less than Zero"
ENDIF
```

Both `ELSE` and `ELSEIF` are optional.

Conditional Exit Loops

Can set up a DO loop which is terminated by simply jumping out of it. Consider,

```
i = 0
DO
  i = i + 1
  IF (i .GT. 100) EXIT
  PRINT*, "I is", i
END DO
! if i>100 control jumps here
PRINT*, "Loop finished. I now equals", i
```

this will generate

```
I is   1
I is   2
I is   3
  ....
I is   100
Loop finished. I now equals   101
```

The EXIT statement tells control to jump out of the current DO loop.

Conditional Cycle Loops

Can set up a `DO` loop which, on some iterations, only executes a subset of its statements. Consider,

```
    i = 0
    DO
      i = i + 1
      IF (i >= 50 .AND. i <= 59) CYCLE
      IF (i > 100) EXIT
      PRINT*, "I is", i
    END DO
    PRINT*, "Loop finished. I now equals", i
```

this will generate

```
  I is   1
  I is   2
    ....
  I is   49
  I is   60
    ....
  I is   100
  Loop finished. I now equals   101
```

`CYCLE` forces control to the **innermost** active `DO` statement and the loop begins a new iteration.

Named and Nested Loops

Loops can be given names and an `EXIT` or `CYCLE` statement can be made to refer to a particular loop.

```
0|      outa: DO
1|       inna: DO
2|        ...
3|        IF (a.GT.b) EXIT outa  ! jump to line 9
4|        IF (a.EQ.b) CYCLE outa ! jump to line 0
5|        IF (c.GT.d) EXIT inna  ! jump to line 8
6|        IF (c.EQ.a) CYCLE      ! jump to line 1
7|       END DO inna
8|      END DO outa
9|      ...
```

The (optional) name following the `EXIT` or `CYCLE` highlights which loop the statement refers to.

Loop names can only be used once per program unit.

DO ... WHILE Loops

If a condition is to be tested at the top of a loop a DO ... WHILE loop could be used,

```
DO WHILE (a .EQ. b)
 ...
END DO
```

The loop only executes if the logical expression evaluates to .TRUE.. Clearly, here, the values of a or b must be modified within the loop otherwise it will never terminate.

The above loop is functionally equivalent to,

```
DO; IF (a .NE. b) EXIT
 ...
END DO
```

Indexed `DO` Loops

Loops can be written which cycle a fixed number of times. For example,

```
    DO i1 = 1, 100, 1
      ... ! i is 1,2,3,...,100
      ... ! 100 iterations
    END DO
```

The formal syntax is as follows,

`DO` < `DO`-*var* >=< *expr1* >,< *expr2* > [,< *expr3* >]
      < *exec-stmts* >
`END DO`

The number of iterations, which is evaluated **before** execution of the loop begins, is calculated as

`MAX(INT((`< *expr2* >-< *expr1* >+< *expr3* >)/< *expr3* >),0)

If this is zero or negative then the loop is not executed.

If < *expr3* > is absent it is assumed to be equal to `1`.

Examples of Loop Counts

A few examples of different loops,

1. upper bound not exact,

```
loopy: DO i = 1, 30, 2
 ... ! i is 1,3,5,7,...,29
 ... ! 15 iterations
END DO loopy
```

2. negative stride,

```
DO j = 30, 1, -2
 ... ! j is 30,28,26,...,2
 ... ! 15 iterations
END DO
```

3. a zero-trip loop,

```
DO k = 30, 1, 2
 ... ! 0 iterations
 ... ! loop skipped
END DO
```

4. missing stride — assume it is `1`,

```
DO l = 1,30
 ... ! i = 1,2,3,...,30
 ... ! 30 iterations
END DO
```

SELECT CASE Construct I

Simple example

```
    SELECT CASE (i)
      CASE (3,5,7)
        PRINT*,"i is prime"
      CASE (10:)
        PRINT*,"i is > 10"
      CASE DEFAULT
        PRINT*, "i is not prime and is < 10"
    END SELECT
```

An IF .. ENDIF construct could have been used but a SELECT CASE is neater and more efficient. Another example,

```
    SELECT CASE (num)
      CASE (6,9,99,66)
!     IF(num==6.OR. .. .OR.num==66) THEN
        PRINT*, "Woof woof"
      CASE (10:65,67:98)
!     ELSEIF((num >= 10 .AND. num <= 65) .OR. ...
        PRINT*, "Bow wow"
      CASE DEFAULT
!     ELSE
        PRINT*, "Meeeoow"
     END SELECT
!    ENDIF
```

Intrinsic Procedures

Fortran 90 has 113 in-built or *intrinsic* procedures to perform common tasks efficiently, they belong to a number of classes:

- elemental such as:
  - mathematical, for example, `SIN` or `LOG`.
  - numeric, for example, `SUM` or `CEILING`;
  - character, for example, `INDEX` and `TRIM`;
  - bit, for example, `IAND` and `IOR`;
- inquiry, for example, `ALLOCATED` and `SIZE`;
- transformational, for example, `REAL` and `TRANSPOSE`;
- miscellaneous (non-elemental `SUBROUTINE`s), for example, `SYSTEM_CLOCK` and `DATE_AND_TIME`.

Note all intrinsics which take `REAL` valued arguments also accept `DOUBLE PRECISION` arguments.

Type Conversion Functions

It is easy to transform the type of an entity,

- ☐ `REAL(i)` converts `i` to a real approximation,
- ☐ `INT(x)` truncates `x` to the integer equivalent,
- ☐ `DBLE(a)` converts `a` to `DOUBLE PRECISION`,
- ☐ `IACHAR(c)` returns the position of `CHARACTER c` in the ASCII collating sequence,
- ☐ `ACHAR(i)` returns the i^{th} character in the ASCII collating sequence.

All above are intrinsic functions. For example,

```
PRINT*, REAL(1), INT(1.7), INT(-0.9999)
PRINT*, IACHAR('C'), ACHAR(67)
```

are equal to

```
1.000000 1 0
67 C
```

Mathematical Intrinsic Functions

Summary,

`ACOS(x)`	arccosine
`ASIN(x)`	arcsine
`ATAN(x)`	arctangent
`ATAN2(y,x)`	arctangent of complex number (x, y)
`COS(x)`	cosine where x is in radians
`COSH(x)`	hyperbolic cosine where x is in radians
`EXP(x)`	e raised to the power x
`LOG(x)`	natural logarithm of x
`LOG10(x)`	logarithm base 10 of x
`SIN(x)`	sine where x is in radians
`SINH(x)`	hyperbolic sine where x is in radians
`SQRT(x)`	the square root of x
`TAN(x)`	tangent where x is in radians
`TANH(x)`	tangent where x is in radians

Numeric Intrinsic Functions

Summary,

`ABS(a)`	absolute value
`AINT(a)`	truncates `a` to whole `REAL` number
`ANINT(a)`	nearest whole `REAL` number
`CEILING(a)`	smallest `INTEGER` greater than or equal to `REAL` number
`CMPLX(x,y)`	convert to `COMPLEX`
`DBLE(x)`	convert to `DOUBLE PRECISION`
`DIM(x,y)`	positive difference
`FLOOR(a)`	biggest `INTEGER` less than or equal to real number
`INT(a)`	truncates a into an `INTEGER`
`MAX(a1,a2,a3,...)`	the maximum value of the arguments
`MIN(a1,a2,a3,...)`	the minimum value of the arguments
`MOD(a,p)`	remainder function
`MODULO(a,p)`	modulo function
`NINT(x)`	nearest `INTEGER` to a `REAL` number
`REAL(a)`	converts to the equivalent `REAL` value
`SIGN(a,b)`	transfer of sign — `ABS(a)*(b/ABS(b))`

PRINT Statement

This is the simplest form of directing unformatted data to the standard output channel, for example,

```
PROGRAM Owt
 IMPLICIT NONE
  CHARACTER(LEN=*), PARAMETER :: &
     long_name = "Llanfair...gogogoch"
  REAL :: x, y, z
  LOGICAL :: lacigol
   x = 1; y = 2; z = 3
   lacigol = (y .eq. x)
   PRINT*, long_name
   PRINT*, "Spock says ""illogical&
            &Captain"" "
   PRINT*, "X = ",x," Y = ",y," Z = ",z
   PRINT*, "Logical val: ",lacigol
END PROGRAM Owt
```

produces on the screen,

```
Llanfair...gogogoch
Spock says "illogical Captain"
X =  1.000  Y =  2.000  Z =  3.000
Logical val:  F
```

READ Statement

READ accepts unformatted data from the standard input channel, for example, if the type declarations are the same as for the PRINT example,

```
    READ*, long_name
    READ*, x, y, z
    READ*, lacigol
```

accepts

```
    Llanphairphwyll...gogogoch
    0.4 5. 1.0e12
    T
```

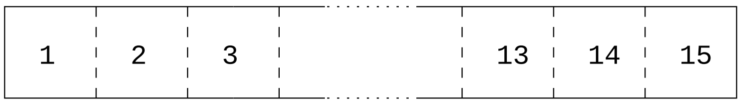
Note,

- ☐ each READ statement reads from a newline;
- ☐ the READ statement can transfer any object of intrinsic type from the standard input;

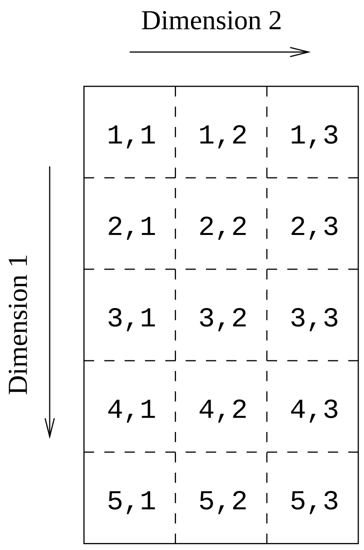
Arrays

Arrays (or matrices) hold a collection of different values at the same time. Individual elements are accessed by **subscripting** the array.

A 15 element array can be visualised as:



And a 5 × 3 array as:



Every array has a type and each element holds a value of that type.

Array Terminology

Examples of declarations:

```
REAL, DIMENSION(15)      :: X
REAL, DIMENSION(1:5,1:3) :: Y, Z
```

The above are *explicit-shape* arrays.

Terminology:

- **rank** — number of dimensions.

  Rank of `X` is 1; rank of `Y` and `Z` is 2.

- **bounds** — upper and lower limits of indices.

  Bounds of `X` are 1 and 15; Bound of `Y` and `Z` are 1 and 5 and 1 and 3.

- **extent** — number of elements in dimension;

  Extent of `X` is 15; extents of `Y` and `Z` are 5 and 3.

- **size** — total number of elements.

  Size of `X`, `Y` and `Z` is 15.

- **shape** — rank and extents;

  Shape of `X` is 15; shape of `Y` and `Z` is 5,3.

- **conformable** — same shape.

  `Y` and `Z` are conformable.

Declarations

Literals and constants can be used in array declarations,

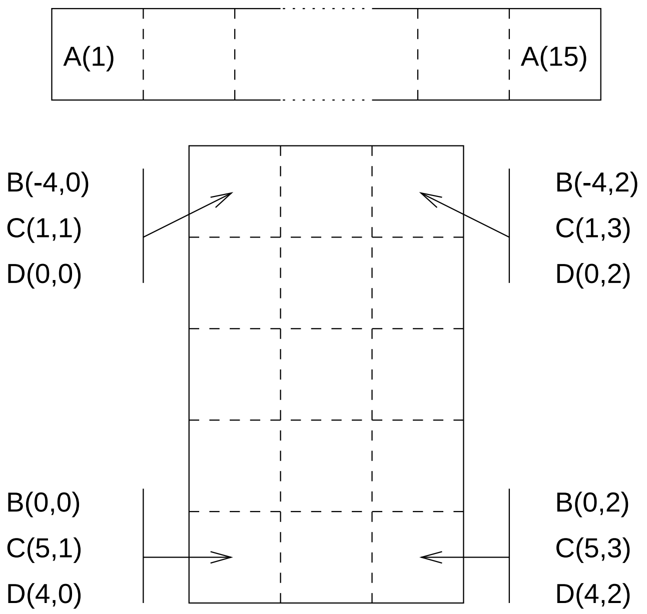
```
REAL, DIMENSION(100)       :: R
REAL, DIMENSION(1:10,1:10) :: S
REAL                       :: T(10,10)
REAL, DIMENSION(-10:-1)    :: X
INTEGER, PARAMETER           :: lda = 5
REAL, DIMENSION(0:lda-1)     :: Y
REAL, DIMENSION(1+lda*lda,10) :: Z
```

- default lower bound is 1,
- bounds can begin and end anywhere,
- arrays can be zero-sized (if `lda = 0`),

Visualisation of Arrays

```
REAL, DIMENSION(15)        :: A
REAL, DIMENSION(-4:0,0:2) :: B
REAL, DIMENSION(5,3)       :: C
REAL, DIMENSION(0:4,0:2)  :: D
```

Individual array elements are denoted by *subscripting* the array name by an `INTEGER`, for example, `A(7)` 7th element of `A`, or `C(3,2)`, 3 elements down, 2 across.



Array Conformance

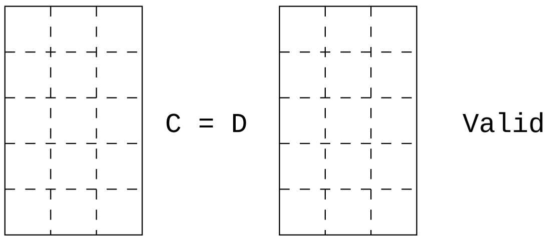
Arrays or sub-arrays must conform with all other objects in an expression:

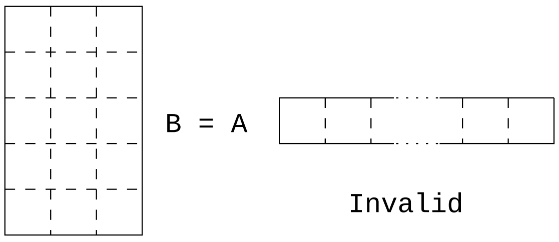
- ☐ a scalar conforms to an array of any shape with the same value for every element:

```
C = 1.0   ! is valid
```

- ☐ two array references must conform in their shape.

  Using the declarations from before:





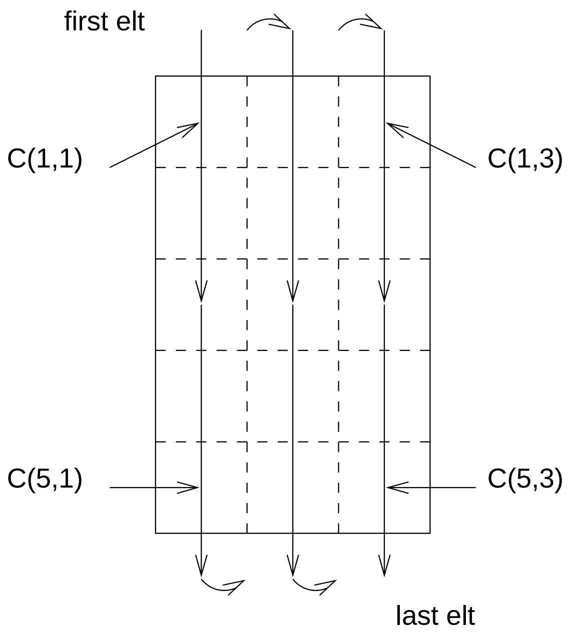
A and B have the same size but have different shapes so cannot be directly equated.

Array Element Ordering

Organisation in memory:

- ☐ Fortran 90 does not specify anything about how arrays should be located in memory. **It has no storage association.**
- ☐ Fortran 90 does define an array element ordering for certain situations which is of column major form,

The array is conceptually ordered as:



```
C(1,1),C(2,1),..,C(5,1),C(1,2),C(2,2),..,C(5,3)
```

Array Syntax

Can reference:

- whole arrays
  - `A = 0.0`
    sets whole array `A` to zero.
  - `B = C + D`
    adds `C` and `D` then assigns result to `B`.
- elements
  - `A(1) = 0.0`
    sets one element to zero,
  - `B(0,0) = A(3) + C(5,1)`
    sets an element of `B` to the sum of two other elements.
- array sections
  - `A(2:4) = 0.0`
    sets `A(2)`, `A(3)` and `A(4)` to zero,
  - `B(-1:0,1:2) = C(1:2,2:3) + 1.0`
    adds one to the subsection of `C` and assigns to the subsection of `B`.

Whole Array Expressions

Arrays can be treated like a single variable in that:

- can use intrinsic operators between conformable arrays (or sections),

  ```
  B = C * D - B**2
  ```

  this is equivalent to concurrent execution of:

  ```
  B(-4,0) = C(1,1)*D(0,0)-B(-4,0)**2 ! in ||
  B(-3,0) = C(2,1)*D(1,0)-B(-3,0)**2 ! in ||
   ...
  B(-4,1) = C(1,2)*D(0,1)-B(-4,1)**2 ! in ||
   ...
  B(0,2)  = C(5,3)*D(4,2)-B(0,2)**2  ! in ||
  ```

- elemental intrinsic functions can be used,

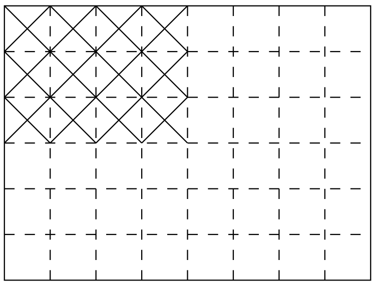
  ```
  B = SIN(C)+COS(D)
  ```

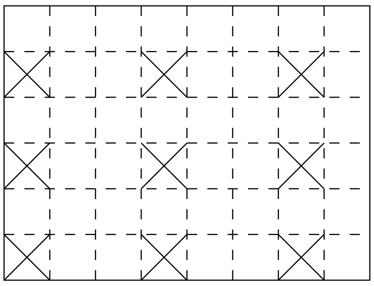
  the function is applied element by element.

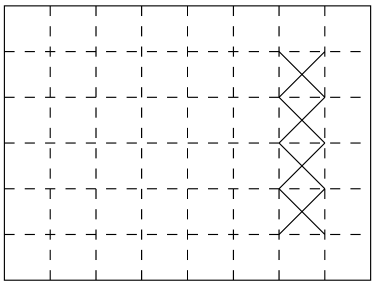
Array Sections — Visualisation

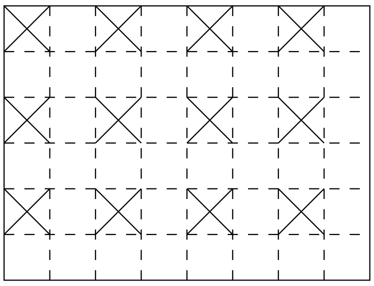
Given,

```
REAL, DIMENSION(1:6,1:8) :: P
```


P(1:3,1:4)


P(2:6:2,1:7:3)


P(2:5,7) P(2:5,7:7)


P(1:6:2,1:8:2)

Consider the following assignments,

- ☐ `P(1:3,1:4) = P(1:6:2,1:8:2)` and `P(1:3,1:4) = 1.0` are valid.
- ☐ `P(2:8:2,1:7:3) = P(1:3,1:4)` and `P(2:6:2,1:7:3) = P(2:5,7)` are not.
- ☐ `P(2:5,7)` is a 1D section (scalar in dimension 2) whereas `P(2:5,7:7)` is a 2D section.

Array Sections

subscript-triplets specify sub-arrays. The general form is:

[< *bound1* >]:[< *bound2* >][:< *stride* >]

The section starts at < *bound1* > and ends at or before < *bound2* >. < *stride* > is the increment by which the locations are selected.

< *bound1* >, < *bound2* > and < *stride* > must all be scalar integer expressions. Thus

```
A(:)          ! the whole array
A(3:9)        ! A(m) to A(n) in steps of 1
A(3:9:1)      ! as above
A(m:n)        ! A(m) to A(n)
A(m:n:k)      ! A(m) to A(n) in steps of k
A(8:3:-1)     ! A(8) to A(3) in steps of -1
A(8:3)        ! A(8) to A(3) step 1 => Zero size
A(m:)         ! from A(m) to default UPB
A(:n)         ! from default LWB to A(n)
A(::2)        ! from default LWB to UPB step 2
A(m:m)        ! 1 element section
A(m)          ! scalar element - not a section
```

are all valid sections.

Array I/O

The conceptual ordering of array elements is useful for defining the order in which array elements are output. If `A` is a 2D array then:

```
    PRINT*, A
```

would produce output in the order:

```
  A(1,1),A(2,1),A(3,1),..,A(1,2),A(2,2),..
```

```
    READ*, A
```

would assign to the elements in the above order.

This order could be changed by using intrinsic functions such as `RESHAPE`, `TRANSPOSE` or `CSHIFT`.

Array I/O Example

Consider the matrix `A`:

$$\begin{array}{|c|c|c|}\hline 1 & 4 & 7 \\ \hline 2 & 5 & 8 \\ \hline 3 & 6 & 9 \\ \hline\end{array}$$

The following `PRINT` statements

```
    ...
     PRINT*, 'Array element   =',a(3,2)
     PRINT*, 'Array section   =',a(:,1)
     PRINT*, 'Sub-array       =',a(:2,:2)
     PRINT*, 'Whole Array     =',a
     PRINT*, 'Array Transp''d =',TRANSPOSE(a)
   END PROGRAM Owt
```

produce on the screen,

```
    Array element    = 6
    Array section    = 1 2 3
    Sub-array        = 1 2 4 5
    Whole Array      = 1 2 3 4 5 6 7 8 9
    Array Transposed = 1 4 7 2 5 8 3 6 9
```

Allocatable Arrays

Fortran 90 allows arrays to be created on-the-fly; these are known as *deferred-shape* arrays:

- ☐ Declaration:

  ```
  INTEGER, DIMENSION(:), ALLOCATABLE :: ages    ! 1D
  REAL, DIMENSION(:,:), ALLOCATABLE :: speed    ! 2D
  ```

  Note `ALLOCATABLE` attribute and fixed rank.

- ☐ Allocation:

  ```
  READ*, isize
  ALLOCATE(ages(isize), STAT=ierr)
  IF (ierr /= 0) PRINT*, "ages : Allocation failed"

  ALLOCATE(speed(0:isize-1,10),STAT=ierr)
  IF (ierr /= 0) PRINT*, "speed : Allocation failed"
  ```

- ☐ the optional `STAT=` field reports on the success of the storage request. If the `INTEGER` variable `ierr` is zero the request was successful otherwise it failed.

Deallocating Arrays

Heap storage can be reclaimed using the `DEALLOCATE` statement:

```
IF (ALLOCATED(ages)) DEALLOCATE(ages,STAT=ierr)
```

- ☐ it is an error to deallocate an array without the `ALLOCATE` attribute or one that has not been previously allocated space,
- ☐ there is an intrinsic function, `ALLOCATED`, which returns a scalar `LOGICAL` values reporting on the status of an array,
- ☐ the `STAT=` field is optional but its use is recommended,
- ☐ if a procedure containing an allocatable array which does not have the `SAVE` attribute is exited without the array being `DEALLOCATE`d then this storage becomes inaccessible.

2.2 Basic Fortran exercises

If you have never used Fortran before, the way to run Fortran code is by writing the "source" code in a text file (using any text editor), then compiling the code, and then executing it.

For example, you could write the following source code in any text editor and save it as "test.f90":

```
$ cat hello.f90
PROGRAM HELLO_WORLD
  IMPLICIT NONE

  INTEGER :: count

  PRINT*, "Hello World x 10"
  DO count=1,10
     PRINT*, "Hello World!", count
  END DO

END PROGRAM HELLO_WORLD
```

Then, in order to translate that source code into an executable file, you need to compile it (for example with the Fortran compiler in the GCC suite, using the command "gfortran"). Below, the option -o tells the compiler to generate an executable file with name "hello.x":

```
$ gfortran -o hello.x hello.f90
```

Then you would have to execute it by typing the number of the executable file produced above. The syntax "./" will tell your shell that the executable file is located in the directory where you are running this command.

```
$ ./hello
 Hello World x 10
 Hello World!           1
 Hello World!           2
 Hello World!           3
 Hello World!           4
 Hello World!           5
 Hello World!           6
 Hello World!           7
 Hello World!           8
 Hello World!           9
 Hello World!          10
```

Below there are some exercises to familiarize yourself with the features of Fortran that we have seen in section 2.1. If you need more basic exercises, you could try for example the first 10 problems in the Project Euler website[1].

You can find solutions to these exercises in section A.1, but you are *highly encouraged* to try to solve them first on your own.

2.2.1 Read numbers and print them in reverse order

Write a program that reads first an integer N, and then N integer numbers, which should then be printed in reverse order.

```
$ ./ex1
 Enter number of data to read:
5
```

1. https://projecteuler.net/

```
 Enter (in one line)           5 integers
34 56 78 98 45
 The data in reverse order are:
          45          98          78          56          34
```

2.2.2 Print all leap years between two given years

Write a program that given starting and ending years, prints all leap years within that period. (A year is a leap year if (divisible by 4 but not by 100) or (divisible by 400)). See: https://en.wikipedia.org/wiki/Leap_year

```
$ ./ex2
 Enter starting and end years
95 120
 Leap years between years          95 and         120 are:
          96
         104
         108
         112
         116
         120
```

2.2.3 Write a naïve program to perform matrix multiplication

This program will expect to first read (using the basic READ* Fortran command) three integers: m,n,p Then the first matrix (size m x n) will be read row by row. Then the second matrix (size n x p) will be read row by row.

In order to not type these every time you want to execute the code, we can use input redirection, which means that we will store in a file the values that we would normally type in the keyboard. For example, we can have the following file:

```
$ cat ex3.input
3 4 5     !! m,n,p
4 5 6 7   !! matriz A (m x n)
8 4 3 2
9 2 4 5
2 5 6 7 8 !! matriz B (n x p)
3 4 1 9 4
8 3 6 4 2
9 6 3 7 1
```

And then execute the code as follows:

```
$ ./ex3 < ex3.input
   134.000000       100.000000       86.0000000       146.000000       71.0000000
   70.0000000       77.0000000       76.0000000       118.000000       88.0000000
   101.000000       95.0000000       95.0000000       132.000000       93.0000000
```

2.2.4 Write a program that given an integer, checks whether it is palindromic (i.e. that can be read the same way backwards and forwards)

```
$ ./ex4
 Enter num
243
 NOT Palindromic
```

```
$ ./ex4
 Enter num
5678765
 Palindromic
```

2.2.5 Summation Example

Write a program that given an array of N floats and a W (width), find which W consecutive numbers have the greatest sum:

```
$ cat ex5.input
20   ! N - number of floats
5    ! W - window width
6.3  ! N floats to follow
7.6
9.2
3.4
5.6
7.23
9.76
6.83
5.45
4.56
4.86
5.8
6.4
7.43
7.87
8.6
9.25
8.9
8.4
7.23

$ ./ex5 < ex5.input
 Greatest sum is:   43.0200043     given by the following numbers:
   7.86999989
   8.60000038
   9.25000000
   8.89999962
   8.39999962
```

2.2.6 Salaries Example

Write a program to calculate the cost to a company of increasing the salary of its employees. The input will be given as per the following example:

```
$ cat ex6.input
9     ! n - number of employees
3     ! nc - number of categories
10500 ! salaries of each employee (n lines)
16140
22300
15960
14150
12180
13230
15760
31000
1     ! category of each employee (n lines)
2
```

```
3
2
1
1
1
2
3
5     ! increase (percentage) for each category
4
2

$ ./ex6 < ex6.input
 Current cost:   151220.000      New cost:   156703.406     Difference:   5483.40625
```

2.2.7 Travelling Salesman Problem

The travelling salesman problem is a very well known and computationally very hard problem, so this will only work for a very small number of towns, but the idea is that a salesman travels between a number of towns whose distances (integer numbers) are given, and you are required to find the shortest route which brings the salesman to all the towns. Since we haven't seen recursion yet, the number of towns will be fixed for the time being to 5. Later on we will improve this to work with any number of towns.

```
$ cat ex7.input
5                       ! n - number of towns
0   120 180 202 300     ! n lines, each line indicating the distances from town i [1:5]
120 0   175 340 404     !          to all towns j [1:5]
180 175 0   98  56
202 340 98  0   168
300 404 56  168 0

$ ./ex7 < ex7.input
 Shortest route travelled is :           1           2           3           5           4
 Distance travelled =          721
```

2.2.8 N-body problem

Given the description of the N-body problem in chapter 1, write a program to solve the N-body problem.

PART II

N-BODY SOLUTION WITH BARNES-HUT ALGORITHM

Chapter 3

Barnes-Hut algorithm

The material in this chapter is an abriged version of the `https://people.eecs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html` webpage

3.1 Introduction

Far field forces like gravity are very expensive to compute because the force on each particle depends on all the other particles, as we saw in the naïve solution to the n-body problem A.1.8.

```
a = 0.0
DO i = 1,n
   DO j = i+1,n
      rji = r(j,:) - r(i,:)
      r2 = SUM(rji**2)
      r3 = r2 * SQRT(r2)
      a(i,:) = a(i,:) + m(j) * rji / r3
      a(j,:) = a(j,:) - m(i) * rji / r3
   END DO
END DO
```

Thus, the calculation cost rises as O(n2), so even with parallelism, the far field forces will extremely expensive to compute. Fortunately, it turns out that there are clever divide-and-conquer algorithms which only take O(n log n) or even just O(n) time for this problem.

3.2 How to reduce the number of particles in the force sum

Suppose we wanted to compute the gravitational force on the earth from the known stars and planets. A glance skyward on a clear night reveals a dauntingly large number of stars that must be included in the calculation, each one contributing a term to the force sum.

One of those dots of light we might want to include in our sum is, however, not a single star (particle) at all, but rather the Andromeda galaxy, which itself consist of billions of stars. But these appear so close together at this distance that they show up as a single dot to the naked eye. It is tempting – and correct – to suspect that it is good enough to treat the Andromeda galaxy as a single point anyway, located at the center of mass of the Andromeda galaxy, and with a mass equal to the total mass of the Andromeda galaxy. This is indicated below, with a red x marking the center of mass. More mathematically, since the ratio

```
                size of box containing Andromeda
      D/r  = ------------------------------------
              distance of center of mass from Earth
```

is so small, we can safely and accurately replace the sum over all stars in Andromeda with one term at their center of mass (see figure 3.1).

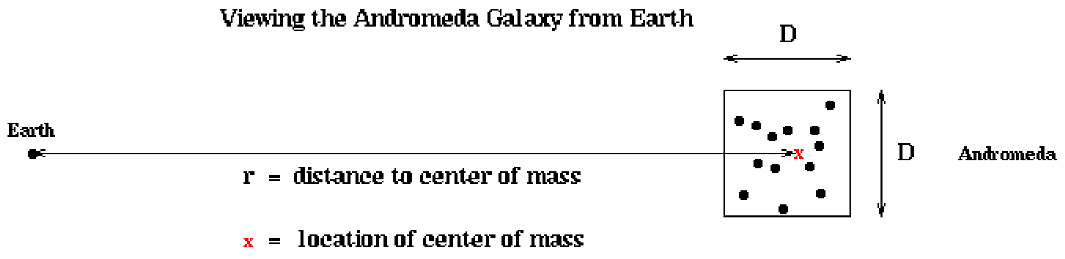

Viewing the Andromeda Galaxy from Earth
D
Earth
D
Andromeda
r = distance to center of mass
x = location of center of mass

Figure 3.1: Viewing the Andromeda Galaxy from Earth

This idea is hardly new, but what is new is applying this idea recursively. First, it is clear that from the point of view of an observer in the Andromeda galaxy, our own Milky Way galaxy can also be well approximated by a point mass at our center of mass. But more importantly, within the Andromeda (or Milky Way) galaxy itself, this geometric picture repeats itself as shown below: as long as the ratio D1/r1 is also small, the stars inside the smaller box can be replaced by their center of mass in order to compute the gravitational force on, say, the planet Vulcan. This nesting of boxes within boxes can be repeated recursively (see figure 3.2)

3.3 Quadtrees and Octtrees

What we need is a data structure to subdivide space that makes this recursion easy. The answer in 3D is the octtree, and in 2D the answer is the quadtree. We begin by describing the quadtree, because it is easier to draw in 2D; the octtree will be analogous. The quadtree begins with a square in the plane; this is the root of the quadtree. This large square can be broken into four smaller squares of half the perimeter and a quarter the area each; these are the four children of the root. Each child can in turn be broken into 4 subsquares to get its children, and so on. This is shown below. Each colored dot in the tree corresponds to a square in the picture on the left, with edges of that color (and of colors from higher tree levels). See figure 3.3.

An octtree is similar, but with 8 children per node, corresponding to the 8 subcubes of the larger cube, as illustrated in figure 3.4.

The algorithm we present below begins by constructing a quadtree (or octtree) to store the particles. Thus, the leaves of the tree will contain (or have pointers to) the positions and masses of the particles in the corresponding box.

The most interesting problems occur when the particles are not uniformly distributed in their bounding box, so that many of the leaves of a complete quadtree would be empty. In this case, it makes no sense to store these empty parts of the quadtree. Instead, we continue to subdivide squares only when they contain more than 1 particle (or some small number of particles). This leads to the adaptive quadtree

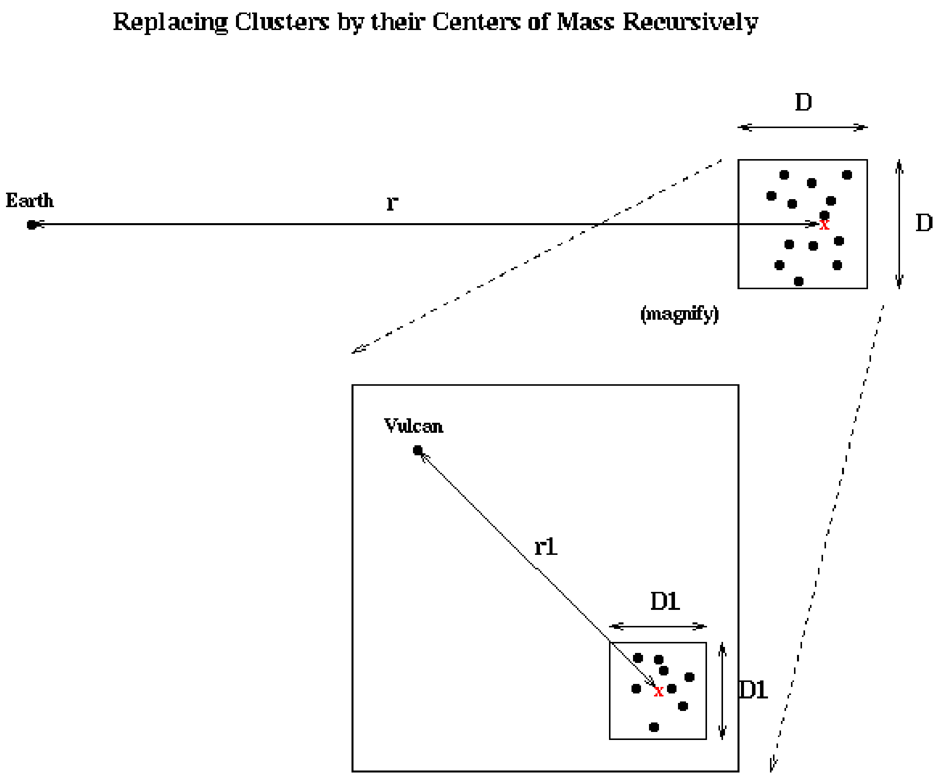


Figure 3.2: Replacing Clusters by their Centers of Mass Recursively

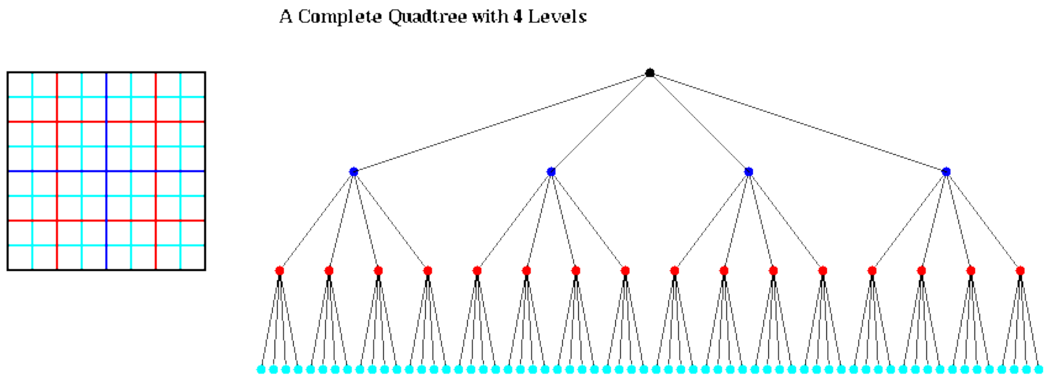


Figure 3.3: A complete quadtree with 4 levels

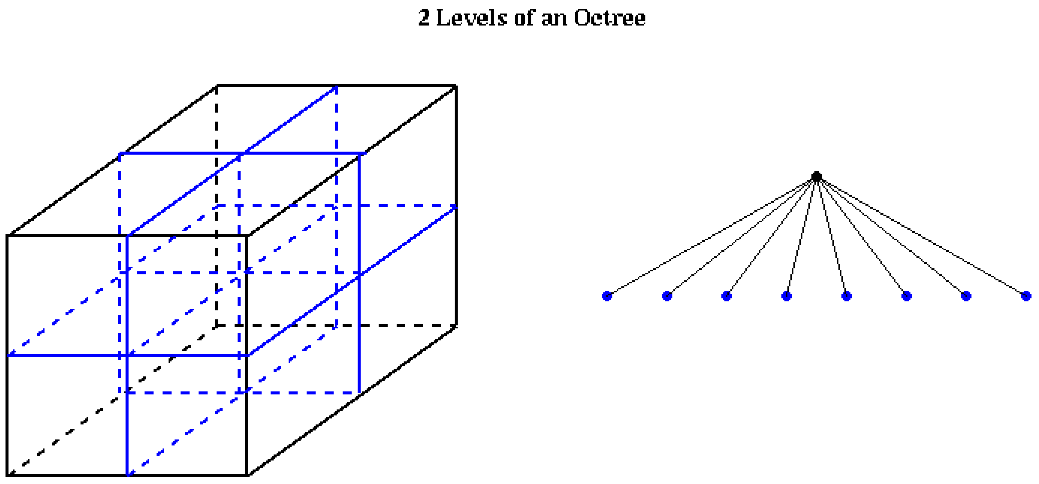


Figure 3.4: 2 Levels of an Octtree

shown in the figure below. Note that there are exactly as many leaves as particles. Children are ordered counterclockwise starting at the lower left, see figure 3.5.

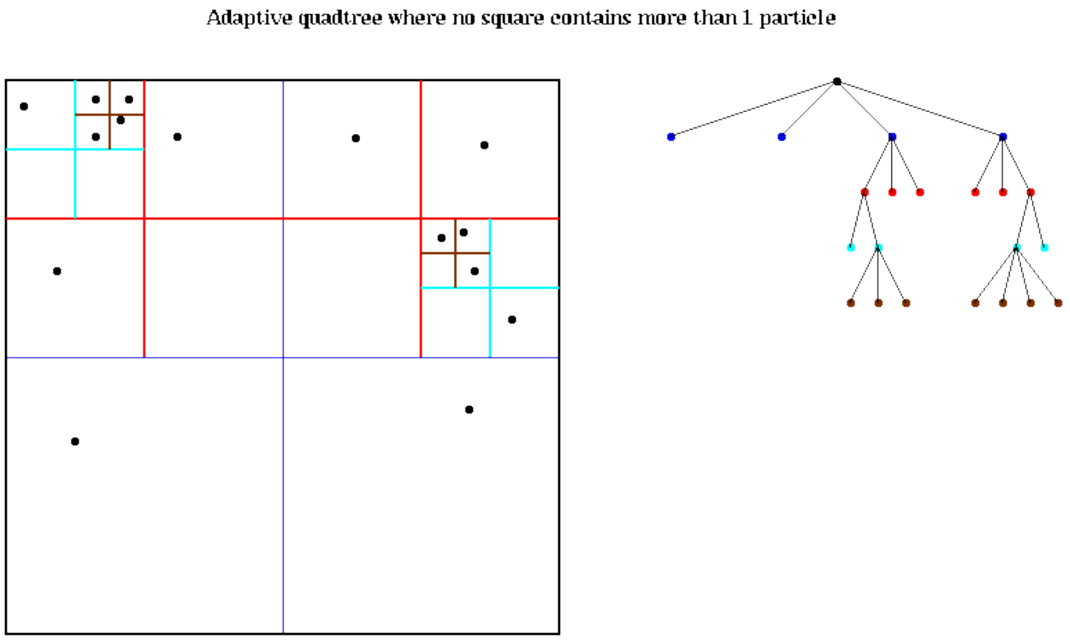


Figure 3.5: Adaptive quadtree where no square contains more than 1 particle

3.4 The Barnes-Hut Algorithm

At a high level, here is the Barnes-Hut algorithm:

1. Build the Quadtree
2. For each subsquare in the quadtree, compute the center of mass and total mass for all the particles it contains.
3. For each particle, traverse the tree to compute the force on it.

The core of the algorithm is computing the force on each particle. For that we use the idea in the first figure above, namely that if the ratio:

```
D                       size of box
-   =   ----------------------------------------------
r       distance from particle to center of mass of box
```

is small enough, then we can compute the force due to all the particles in the box just by using the mass and center of mass of the particles in the box. We will compare D/r to a user-supplied threshold theta (usually a little less than 1) to make this decision.

Thus, if D/r < theta, we compute the gravitational force on the particle as if all particles in the box were just one particle with mass being the total mass of the particles in the box and the position being the center of mass of all the particles in the box.

Chapter 4

Intermediate Fortran for Barnes-Hut N-body implementation

In order to implement the Barnes-Hut algorithm described in chapter 3 we will need to learn some intermediate programming concepts.

4.1 Procedures and recursion

Program Units

Fortran 90 has two main program units

- ☐ main PROGRAM,

  the place where execution begins and where control should eventually return before the program terminates. May contain procedures.

- ☐ MODULE.

  a program unit which can contain procedures and declarations. It is intended to be attached to any other program unit where the entities defined within it become accessible.

There are two types of procedures:

- ☐ SUBROUTINE,

  a parameterised named sequence of code which performs a specific task and can be invoked from within other program units.

- ☐ FUNCTION,

  as a SUBROUTINE but returns a result in the function name (of any specified type and kind).

Main Program Syntax

```
PROGRAM Main
 ! ...
CONTAINS ! Internal Procs
   SUBROUTINE Sub1(..)
    ! Executable stmts
   END SUBROUTINE Sub1
    ! etc.
   FUNCTION Funkyn(...)
    ! Executable stmts
   END FUNCTION Funkyn
 END PROGRAM Main
```

```
[ PROGRAM [ < main program name > ] ]
        < declaration of local objects >
                ...
        < executable stmts >
                ...
[ CONTAINS
        < internal procedure definitions > ]
END [ PROGRAM [ < main program name > ] ]
```

Program Example

```
PROGRAM Main
 IMPLICIT NONE
 REAL :: x
  READ*, x
  PRINT*, FLOOR(x) ! Intrinsic
  PRINT*, Negative(x)
CONTAINS
 REAL FUNCTION Negative(a)
  REAL, INTENT(IN) :: a
   Negative = -a
 END FUNCTION Negative
END PROGRAM Main
```

Subroutines

Consider the following example,

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
  CALL OutputFigures(NumberSet)
 .....
CONTAINS
 SUBROUTINE OutputFigures(Numbers)
  REAL, DIMENSION(:), INTENT(IN) :: Numbers
   PRINT*, "Here are the figures", Numbers
 END SUBROUTINE OutputFigures
END PROGRAM Thingy
```

Internal subroutines lie between `CONTAINS` and `END PROGRAM` statements and have the following syntax

```
SUBROUTINE < procname >[ (< dummy args >) ]
       < declaration of dummy args >
       < declaration of local objects >
              ...
       < executable stmts >
END [ SUBROUTINE [< procname > ] ]
```

Note that, in the example, the `IMPLICIT NONE` statement applies to the whole program including the `SUBROUTINE`.

Functions

Consider the following example,

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
  PRINT*, F(a,b)
 .....
CONTAINS
 REAL FUNCTION F(x,y)
  REAL, INTENT(IN) :: x,y
   F = SQRT(x*x + y*y)
 END FUNCTION F
END PROGRAM Thingy
```

Functions also lie between `CONTAINS` and `END PROGRAM` statements. They have the following syntax:

[< *prefix* >] `FUNCTION` < *procname* >([< *dummyargs* >])
    < *declaration of dummy args* >
    < *declaration of local objects* >
        ...
    < *executable stmts, assignment of result* >
`END` [`FUNCTION` [< *procname* >]]

It is also possible to declare the function type in the declarations area instead of in the header.

Argument Association

Recall, on the `SUBROUTINE` slide we had an invocation:

```
      CALL OutputFigures(NumberSet)
```

and a declaration,

```
      SUBROUTINE OutputFigures(Numbers)
```

`NumberSet` is an *actual argument* and is *argument associated* with the *dummy argument* `Numbers`.

For the above call, in `OutputFigures`, the name `Numbers` is an **alias** for `NumberSet`. Likewise, consider,

```
      PRINT*, F(a,b)
```

and

```
      REAL FUNCTION F(x,y)
```

The actual arguments `a` and `b` are associated with the dummy arguments `x` and `y`.

If the value of a dummy argument changes then so does the value of the actual argument.

Local Objects

In the following procedure

```
SUBROUTINE Madras(i,j)
 INTEGER, INTENT(IN) :: i, j
 REAL                :: a
 REAL, DIMENSION(i,j):: x
```

`a`, and `x` are know as *local objects*. They:

- □ are created each time a procedure is invoked,
- □ are destroyed when the procedure completes,
- □ *do not* retain their values between calls,
- □ do not exist in the programs memory between calls.

`x` will probably have a different size and shape on each call.

The space usually comes from the programs stack.

Argument Intent

Hints to the compiler can be given as to whether a dummy argument will:

- □ only be referenced — `INTENT(IN)`;
- □ be assigned to before use — `INTENT(OUT)`;
- □ be referenced and assigned to — `INTENT(INOUT)`;

```
SUBROUTINE example(arg1,arg2,arg3)
 REAL, INTENT(IN) :: arg1
 INTEGER, INTENT(OUT) :: arg2
 CHARACTER, INTENT(INOUT) :: arg3
 REAL :: r
  r = arg1*ICHAR(arg3)
  arg2 = ANINT(r)
  arg3 = CHAR(MOD(127,arg2))
END SUBROUTINE example
```

The use of `INTENT` attributes is recommended as it:

- □ allows good compilers to check for coding errors,
- □ facilitates efficient compilation and optimisation.

Note: if an actual argument is ever a literal, then the corresponding dummy must be `INTENT(IN)`.

Scoping Rules

Fortran 90 is *not* a traditional block-structured language:

- ☐ the *scope* of an entity is the range of program unit where it is visible and accessible;
- ☐ internal procedures can inherit entities by *host association*.
- ☐ objects declared in modules can be made visible by *use-association* (the `USE` statement) — useful for global data;

Host Association — Global Data

Consider,

```
PROGRAM CalculatePay
 IMPLICIT NONE
 REAL :: Pay, Tax, Delta
 INTEGER :: NumberCalcsDone = 0
 Pay = ...;  Tax = ... ; Delta = ...
 CALL PrintPay(Pay,Tax)
 Tax = NewTax(Tax,Delta)
  ....
CONTAINS
 SUBROUTINE PrintPay(Pay,Tax)
  REAL, INTENT(IN) :: Pay, Tax
  REAL :: TaxPaid
   TaxPaid = Pay * Tax
   PRINT*, TaxPaid
   NumberCalcsDone = NumberCalcsDone + 1
 END SUBROUTINE PrintPay
 REAL FUNCTION NewTax(Tax,Delta)
  REAL, INTENT(IN) :: Tax, Delta
   NewTax =  Tax + Delta*Tax
   NumberCalcsDone = NumberCalcsDone + 1
 END FUNCTION NewTax
END PROGRAM CalculatePay
```

Here, `NumberCalcsDone` is a *global* variable. It is available in all procedures by *host association*.

Scope of Names

Consider the following example,

```
PROGRAM Proggie
 IMPLICIT NONE
 REAL ::  A, B, C
 CALL sub(A)
CONTAINS
 SUBROUTINE Sub(D)
  REAL :: D      ! D is dummy (alias for A)
  REAL :: C      ! local C (diff from Proggie's C)
   C = A**3      ! A cannot be changed
   D = D**3 + C  ! D can be changed
   B = C         ! B from Proggie gets new value
 END SUBROUTINE Sub
END PROGRAM Proggie
```

In `Sub`, as `A` is argument associated it may not be have its value changed but may be referenced.

`C` in `Sub` is totally separate from `C` in `Proggie`, changing its value in `Sub` does **not** alter the value of `C` in `Proggie`.

Recursive Procedures

In Fortran 90 recursion is supported as a feature.

- ☐ recursive procedures call themselves (either directly or indirectly),
- ☐ recursion is a neat technique
- ☐ recursion may incur certain efficiency overheads,
- ☐ recursive procedures must be explicitly declared
- ☐ recursive functions declarations must contain a `RESULT` keyword, and one type declaration refers to both the function name and the result variable.

Recursive Function Example

The following example calculates the factorial of a number and uses $n! = n(n-1)!$

```
PROGRAM Mayne
 IMPLICIT NONE
  PRINT*, fact(12) ! etc
CONTAINS
 RECURSIVE FUNCTION fact(N) RESULT(N_Fact)
  INTEGER, INTENT(IN)  :: N
  INTEGER :: N_Fact ! also defines type of fact
   IF (N > 0) THEN
     N_Fact = N * fact(N-1)
   ELSE
     N_Fact = 1
   END IF
 END function FACT
END PROGRAM Mayne
```

To calculate 4!,

1. 4! is $4 \times 3!$, so calculate 3! then multiply by 4,
2. 3! is $3 \times 2!$, need to calculate 2!,
3. 2! is $2 \times 1!$, 1! is $1 \times 0!$ and $0! = 1$
4. can now work back up the calculation and fill in the missing values.

4.2 Recursion exercises

You can find solutions to these exercises in section A.2, but you are *highly encouraged* to try to solve them first on your own.

A useful blog about recursion: `https://blog.angularindepth.com/learn-recursion-in-10-minute`

4.2.1 Recursive Fibonacci function

Write a recursive function to calculate the Fibonacci numbers `https://en.wikipedia.org/wiki/Fibonacci_number`

4.2.2 Non-Recursive Fibonacci function

Write a non-recursive version of the function above and compare the execution times of both versions for larger 'n' values (for example from 30-45). In Linux you can check the execution time with the command 'time', for example:

```
time ./fibo_r 30
```

4.2.3 Memoized Recursive Fibonacci function

Most probably, your recursive implementation is much slower than the iterative version. Can you guess what is going on? [Modify the recursive routine to count the number of times that the fibonacci function is called. Then, look for information on "memoization" and try to modify the recursive routine so that you still use recursion, but in a way that it is as efficient as the iterative version.

4.2.4 Pseudo-code for printing a Binary Search Tree

Write pseudo-code, that is, without paying attention to the syntax of Fortran, a recursive function to print in ascending order a Binary Search Tree (`https://en.wikipedia.org/wiki/Binary_search_tree`). You can assume that your function will be given a tree estructure called "tree", and that these functions are provided to you: left_branch_exists(tree), which will return TRUE is "tree" has a left branch, right_branch_exists(tree), left_branch(tree), which will return the left branch of the tree, right_branch(tree), and node_value(tree), which will return the value stored in the root node of the tree "tree".

4.2.5 Tower of Hanoi game

Try to solve the Tower of Hanoi game `https://en.wikipedia.org/wiki/Tower_of_Hanoi`

This is more difficult than the previous exercises, but it shows very nicely how useful recursion is in some cases. You assume that you have a pile of pieces (the code should work for any number of pieces) in column 1 and you want to move them to column 3. As for the previous exercise, start with just pseudo-code, so you can concentrate on the problem while forgetting about the details of Fortran. But this one is perfectly doable with the little Fortran we have learnt so far, so you could try to go for a full implementation.

In the page `https://www.mathsisfun.com/games/towerofhanoi.html` you can see a demonstration of how to solve the puzzle and you can try to solve it by hand. A basic implementation to solve this problem is very easy, just needing a routine that prints the necessary movements to solve the puzzle (we do not need to store the state of the puzzle at all, only print the moves that would solve the game. When executing the code, we could just print the number of moves, where each move has the following format:

```
n (f -> t)
```

where n is the piece number to move (1 is the smalles piece), f is the column whence the piece is coming, and t is the column where the piece is going to. For example:

```
$ ./hanoi
 Enter number of pieces
3
           1 (           1  ->           3 )
           2 (           1  ->           2 )
           1 (           3  ->           2 )
           3 (           1  ->           3 )
           1 (           2  ->           1 )
           2 (           2  ->           3 )
           1 (           1  ->           3 )
```

4.2.6 Recursive travelling salesman problem

In section 2.2.7 we looked at the Travelling Salesman Problem, but fixing it at 5 cities. Think about (and if possible implement) a recursive version that would work for any number of cities. If you understand how to write this one, then you are doing fantastic progress with recursion, as it gets a bit tricky to keep track of visited cities. If you don't get it at all, don't panic, we will see a solution in class.

4.2.7 Contained digits

Write a recursive function that given two numbers: N1, N2, will say if all the digits in N1 are contained in number N2.

For example, given (101,231001), we should return TRUE, since all digits in 101 are contained in 231001.

4.2.8 Permutations of a number

Write a recursive subroutine that given a number N1, will print all possible permutations of its digits. Assume for simplicity that N1 has no repeating digits, and before you call the recursive subroutine place all the digits of N1 in an array. You can also use any other auxiliary arrays or scalars to keep track of the permutations you have covered so far.

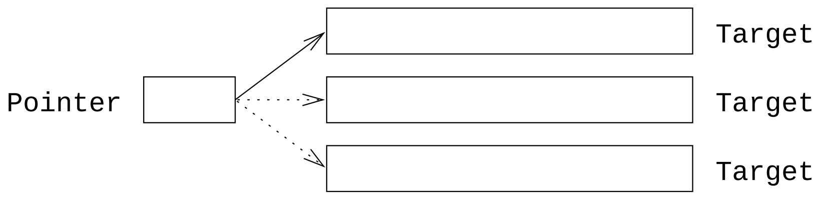
4.2.9 8 queens problem

Try to find a way to solve the 8-queens puzzle (see `https://en.wikipedia.org/wiki/Eight_queens_puzzle`). As always, start without worrying about the implementation and just think about how you could solve this problem (size = 8), assuming you could solve a smaller problem (e.g. size = 7).

4.3 Pointers and derived types

Pointers and Targets

It is often useful to have variables where the space referenced by the variable can be changed as well as the values stored in that space.



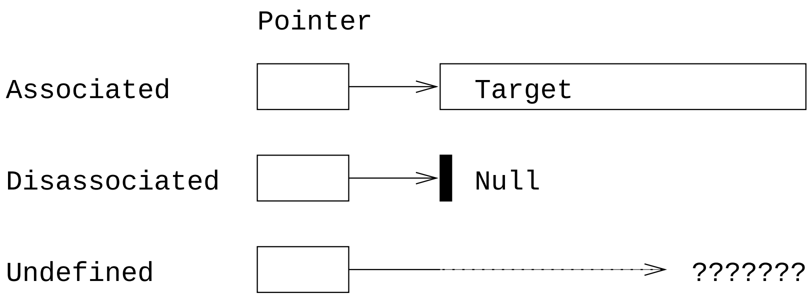
- ☐ The pointer often uses less space than the target.
- ☐ A reference to the pointer will in general be a reference to the target, (pointers are automatically dereferenced).

Terminology

A pointer has 3 possible states:

- ☐ if a pointer has a particular target then the pointer is said to be *associated* with that target,
- ☐ a pointer can be made to have no target — the pointer is *disassociated*,
- ☐ the initial status of a pointer is *undefined*,

Visualisation,



Use **ASSOCIATED** intrinsic to get the (association) status of a pointer.

Pointer Declaration

A `POINTER`:

- □ is a variable with the `POINTER` attribute;
- □ has static type, kind and rank determined by its declaration, for example,

```
REAL, POINTER :: Ptor
REAL, DIMENSION(:,:), POINTER :: Ptoa
```

  - ◇ `Ptor` is a pointer to a scalar real target,
  - ◇ `Ptoa` is a pointer to a 2-D array of reals.

So,

- □ the declaration fixes the type, kind and rank of the target;
- □ pointers to arrays are declared with deferred-shape array specifications;
- □ the rank of a target is fixed but the shape may vary.

Target Declaration

Targets of a pointer must have the `TARGET` attribute.

```
REAL, TARGET :: x, y
REAL, DIMENSION(5,3), TARGET :: a, b
REAL, DIMENSION(3,5), TARGET :: c
```

With these declarations (and those from the previous slide):

- ☐ `x` or `y` may become associated with `Ptor`;
- ☐ `a`, `b` or `c` may become associated with `Ptoa`.

Pointer Manipulation

The following operators manipulate pointers:

- =>, *pointer assignment* — alias a pointer with a given target;

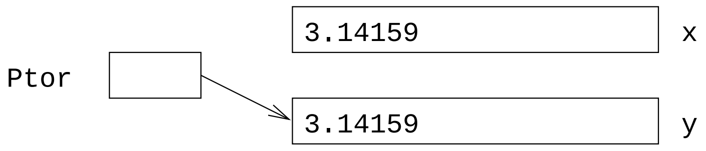
- =, *'normal' assignment* — assign a value to the space pointed at by the pointer.

Pointer assignment makes the pointer and the variable reference the same space while the normal assignment alters the value contained in that space.

Pointer Assignment

Consider,

```
x = 3.14159
Ptor => y
Ptor = x
```



- ☐ `x` and `Ptor` have the same value.
- ☐ `Ptor` is an alias for `y` so the last statement sets `y = 3.14159`.
- ☐ if the value of `x` is subsequently changed, the value of `Ptor` and `y` do not.

Coding,

```
Ptor = 5.0
```

sets `y` to `5.0`.

Dynamic Targets

Targets can also be created dynamically by allocation. ALLOCATE can make space become the target of a pointer.

```
ALLOCATE(Ptor,STAT=ierr)
ALLOCATE(Ptoa(n*n,2*k-1),STAT=ierr)
```

- ☐ the first statement allocates a single real as the target of Ptor.
- ☐ the second allocates a rank 2 real array as the target of Ptoa.

It is not an error to allocate an array pointer that is already associated.

Association Status

The status of a defined pointer may be tested by an intrinsic function:

```
    ASSOCIATED(Ptoa)
```

If `Ptoa` is defined and associated then this will return `.TRUE.`; if it is defined and disassociated it will return `.FALSE.`. If it is undefined the result is also undefined.

The target of a defined pointer may also be tested:

```
    ASSOCIATED(Ptoa, arr)
```

If `Ptoa` is defined and currently associated with the specific target, `arr`, then the function will return `.TRUE.`, otherwise if it will return `.FALSE.`.

Pointer Disassociation

Pointers can be disassociated with their targets by:

- nullification

  ```
  NULLIFY(Ptor)
  ```

  - breaks the connection of its pointer argument with its target (if any),
  - disassociates the pointer.

  Note: it is good practise to nullify *all* pointers before use.

- deallocation

  ```
  DEALLOCATE(Ptoa, STAT=ierr)
  ```

  - breaks the connection between the pointer and its target,
  - deallocates the target.

Practical Example

Pointers can be of great use in iterative problems. Iterative methods:

- ☐ make guess at required solution;
- ☐ use guess as input to an equation to produce better approximation;
- ☐ use new approximation to obtain better approximation;
- ☐ repeat until degree of accuracy is obtained;

```
REAL, DIMENSION(100,100), TARGET :: app1, app2
REAL, DIMENSION(:,:), POINTER :: prev_app, &
                                  next_app, swap
prev_app => app1
next_app => app2
prev_app = initial_app(.....)
DO
 next_app = iteration_function_of(prev_app)
 IF(ABS(MAXVAL(next_app-prev_app))<0.0001)EXIT
 swap => prev_app
 prev_app => next_app
 next_app => swap
END DO
```

Using pointers here avoids having to copy the large matrices.

Derived Types

It is often advantageous to express some objects in terms of aggregate structures, for example: coordinates, (x, y, z).

Fortran 90 allows compound entities or *derived types* to be defined:

```
TYPE COORDS_3D
 REAL :: x, y, z
END TYPE COORDS_3D
TYPE(COORDS_3D) :: pt1, pt2
```

Derived types definitions should be placed in a `MODULE`.

Supertypes

Previously defined types can be used as components of other derived types,

```
TYPE SPHERE
 TYPE (COORDS_3D) :: centre
 REAL             :: radius
END TYPE SPHERE
```

Objects of type SPHERE can be declared:

```
TYPE (SPHERE) :: bubble, ball
```

Derived Type Assignment

Values can be assigned to derived types in two ways:

- component by component;
- as an object.

An individual component may be selected, using the `%` operator:

```
pt1%x = 1.0
bubble%radius = 3.0
bubble%centre%x = 1.0
```

The whole object may be selected and assigned to using a constructor:

```
pt1 = COORDS_3D(1.,2.,3.)
bubble%centre = COORDS_3D(1.,2.,3.)
bubble = SPHERE(bubble%centre,10.)
bubble = SPHERE(COORDS_3D(1.,2.,3.),10.)
```

The derived type component of `SPHERE` must also be assigned to using a constructor.

Assignment between two objects of the same derived type is intrinsically defined,

```
ball = bubble
```

Derived Type I/O

Derived type objects which do not contain pointers (or private) components may be input or output using 'normal' methods:

```
    PRINT*, bubble
```

is exactly equivalent to

```
    PRINT*, bubble%centre%x, bubble%centre%y, &
            bubble%centre%z, bubble%radius
```

Derived types are handled on a component by component basis.

POINTER Components of Derived Types

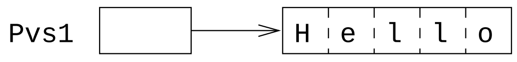
- ALLOCATABLE arrays cannot be used as components in a derived type,
- POINTERs can be,

Dynamically sized structures can be created and manipulated, for example,

```
TYPE VSTRING
 CHARACTER, DIMENSION(:), POINTER :: chars
END TYPE VSTRING
```

this has a component which is a pointer to a 1-D array of characters.

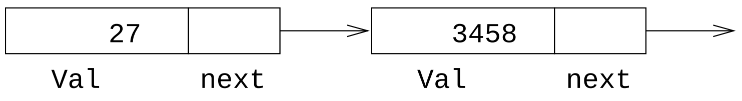
```
TYPE(VSTRING) :: Pvs1
 ...
ALLOCATE(Pvs1%chars(5))
Pvs1%chars = (/"H","e","l","l","o"/)
```



Pointers and Recursive Data Structures

- Derived types which include pointer components provide support for recursive data structures such as linked lists.

```
TYPE CELL
 INTEGER :: val
 TYPE (CELL), POINTER :: next
END TYPE CELL
```



- Assignment between structures containing pointer components is subtlely different from normal,

```
TYPE(CELL) :: A
TYPE(CELL), TARGET :: B
A = B
```

is equivalent to:

```
A%val = B%val
A%next => B%next
```

Practical Example of Linked Lists

The following fragment would create a linked list of cells starting at `head` and terminating with a cell whose `next` pointer is null (disassociated).

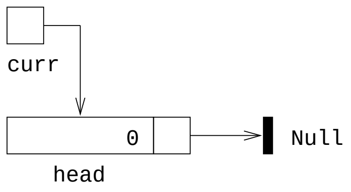
```
PROGRAM Thingy
 IMPLICIT NONE
 TYPE (CELL), TARGET  :: head
 TYPE (CELL), POINTER :: curr, temp
 INTEGER              :: k
 head%val = 0             ! listhead = default
 NULLIFY(head%next)       ! un-undefine
 curr => head             ! curr head of list
 DO
  READ*, k                ! get value of k
  ALLOCATE(temp)          ! create new cell
  temp%val = k            ! assign k to new cell
  NULLIFY(temp%next)      ! set disassociated
  curr%next => temp       ! attach new cell to
                          ! end of list
  curr => temp            ! curr points to new
                          ! end of list
 END DO
END PROGRAM Thingy
```

Example (Continued)

The statements,

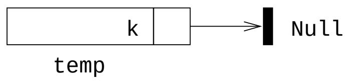
```
head%val = 0; NULLIFY(head%next); curr => head
```

give,



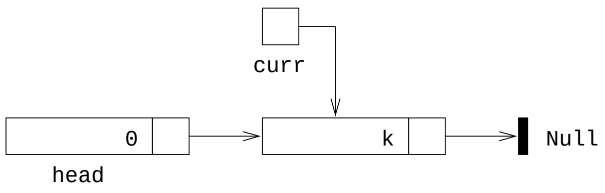
```
ALLOCATE(temp); temp%val = k; NULLIFY(temp%next)
```

give,

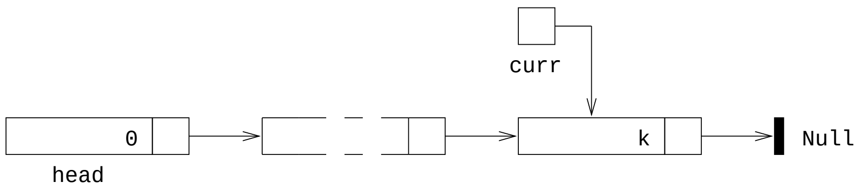


```
curr%next => temp; curr => temp
```

give,



The final list structure is,



Example (Continued)

A “walk-through’ the previous linked list could be written as follows:

```
curr => head
DO
 PRINT*, curr%val
 IF(.NOT.ASSOCIATED(curr%next)) EXIT
 curr => curr%next
ENDDO
```

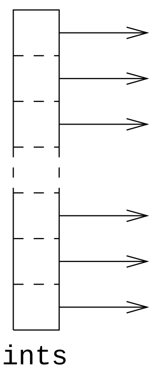
All sorts of multiply linked lists can be created and manipulated in analogous ways.

Arrays of Pointers

It is possible to create what are in effect arrays of pointers:

```
TYPE iPTR
 INTEGER, POINTER :: compon
END TYPE iPTR
TYPE(iPTR), DIMENSION(100) :: ints
```

Visualisation,



Cannot refer to a whole array of pointers,

```
ints(10)%compon ! valid
ints(:)%compon  ! not valid
```

If desired `ints` could have been `ALLOCATABLE`.

4.4 Pointers exercises

You can find solutions to these exercises in section A.2, but you are *highly encouraged* to try to solve them first on your own.

4.4.1 Pointer definitions

Specify two pointers, and let one of them point to a whole vector and the other one point to the seventh element of the same vector.

4.4.2 Swapping numbers

Write a subroutine that will use pointers as its arguments and will swap the values of its two arguments. Write a main program that uses this function to swap the values of two integer variables. Note: you will have to declare two integer variables, plus two pointers to point to these variables and then make the call to the subroutine which will be in charge of swapping the values of the variables.

4.4.3 Aliasing arrays

Declare a one-dimensional array and two pointers, which you can use to assign all even elements of a vector the value 13 and all odd elements of a vector the value 17.

4.5 Pointers and recursion exercises

You can find solutions to these exercises in section A.2, but you are *highly encouraged* to try to solve them first on your own.

4.5.1 Bi-directional list

Modify the example "Practical Example of Linked Lists" in the slides, so that the list will have links both to the previous and to the next element. Also create a recursive subroutine that will print the list both forward and backwards. You will have to take decisions, as to how many temporary pointers you need, etc. You are free to do it in any way, as far as the list is properly formed (i.e. it has links to the previous and next elements, and the ends of the list are signalled with a pointer to NULL), and the routines for printing are able to print the whole list forward and backwards.

4.5.2 Sorted bi-directional list

Modify exercise 4.5.1 so that the elements are inserted in the list sorted.

4.5.3 Sort numbers with a Binary Search Tree

In this exercise we are going to build a binary search tree and then print the tree (remember that we already solved with pseudo-code how to print a binary search tree so that its elements will be printed sorted). The node structure is basically the same as in exercise 4.5.2, but now, instead of "previous" and "next", think of "left" and "right". So first you will have to think of a procedure that will create the tree and then another one to print the tree. Think carefully what you need, and above all make sure that you never leave a node "unattended" (i.e. that you don't create memory leaks by creating nodes and then deleting all pointers to it).

To make it easier, use this main code as a template, and just fill in the routines "place_number", and "Print". We don't want to deal with repeated numbers, so if you give a repeated number, the procedure place_number should just ignore it and print a warning message.

```
PROGRAM listas
IMPLICIT NONE

TYPE CELL
INTEGER :: val
TYPE (CELL), POINTER :: left,right
END TYPE CELL

TYPE (CELL), POINTER :: head
INTEGER :: n,k,i

READ*, n

READ*, k
ALLOCATE(head)
head%val = k
NULLIFY(head%left)
NULLIFY(head%right)

DO i=2,n
READ*, k
CALL place_number(head,k)
END DO

CALL Print(head)

CONTAINS
RECURSIVE SUBROUTINE place_number(node,number)
???
END SUBROUTINE place_number

RECURSIVE SUBROUTINE Print (node)
???
END SUBROUTINE Print

END PROGRAM listas
```

And remember to use input redirection so that you don't have to type in the numbers everytime you run the code. For example, with the following "numeros.txt" file you could run the code as:

```
[angelv@nodo1]$ cat numbers.txt
10
6
3
15
2
7
2
```

```
5
4
21
31

[angelv@nodo1]$ ./tree < numbers.txt
INSERTING NUMBERS
-----------------
Repeated number 2, item ignored!

PRINTING BST
------------
2
3
4
5
6
7
15
21
31
[angelv@nodo1]$
```

4.5.4 Counting number of elements of a Binary Search Tree

Add a function to the exercise 4.5.3, that will return the number of elements in a BST.

4.5.5 Calculating the depth of a Binary Search Tree

Add a function to the exercise 4.5.3, that will return the depth of a BST. An individual node would be depth=1, if this node has at least one child node, then it would be depth=2, if it has “grandchildren”, then it would be depth=3, etc.

4.5.6 Deleting a node in a Binary Search Tree – DIFFICULT

Add a function to the exercise 4.5.3, that given a tree and a number, it will delete the node with value==number (if it exists in the tree).

Before you try to code anything, you will have to make sure that you understand the algorithm to delete the node, which you can check at `https://en.wikipedia.org/wiki/Binary_search_tree#Deletion`

Chapter 5

Barnes-Hut code

barnes-hut_serial.f90

```
PROGRAM tree
  IMPLICIT NONE

  INTEGER :: i,j,k,n
  REAL :: dt, t_end, t, dt_out, t_out, rs, r2, r3
  REAL, PARAMETER :: theta = 1
  REAL, DIMENSION(:), ALLOCATABLE :: m
  REAL, DIMENSION(:,:), ALLOCATABLE :: r,v,a
  REAL, DIMENSION(3) :: rji

  TYPE RANGE
     REAL, DIMENSION(3) :: min,max
  END TYPE RANGE

  TYPE CPtr
     TYPE(CELL), POINTER :: ptr
  END TYPE CPtr

  TYPE CELL
     TYPE (RANGE) :: range
     REAL, DIMENSION(3) :: part
     INTEGER :: pos
     INTEGER :: type !! 0 = no particle; 1 = particle; 2 = conglomerado
     REAL :: mass
     REAL, DIMENSION(3) :: c_o_m
     TYPE (CPtr), DIMENSION(2,2,2) :: subcell
  END TYPE CELL

  TYPE (CELL), POINTER :: head, temp_cell

!! Lectura de datos
!!!!!!!!!!!!!!!!!!!
  READ*, dt
  READ*, dt_out
  READ*, t_end
  READ*, n

  ALLOCATE(m(n))
  ALLOCATE(r(n,3))
  ALLOCATE(v(n,3))
  ALLOCATE(a(n,3))

  DO i = 1, n
     READ*, m(i), r(i,:),v(i,:)
  END DO
```

```
!! Inicialización head node
!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ALLOCATE(head)

  CALL Calculate_ranges(head)
  head%type = 0
  CALL Nullify_Pointers(head)


!! Creación del árbol inicial
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  DO i = 1,n
     CALL Find_Cell(head,temp_cell,r(i,:))
     CALL Place_Cell(temp_cell,r(i,:),i)
  END DO

  CALL Borrar_empty_leaves(head)
  CALL Calculate_masses(head)

!! Calcular aceleraciones iniciales
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  a = 0.0
  CALL Calculate_forces(head)

!! Bucle principal
!!!!!!!!!!!!!!!!!!
  t_out = 0.0
  DO t = 0.0, t_end, dt
     v = v + a * dt/2
     r = r + v * dt

     !! Las posiciones han cambiado, por lo que tenemos que borrar
     !! y reinicializar el árbol
     CALL Borrar_tree(head)

     CALL Calculate_ranges(head)
     head%type = 0
     CALL Nullify_Pointers(head)

     DO i = 1,n
        CALL Find_Cell(head,temp_cell,r(i,:))
        CALL Place_Cell(temp_cell,r(i,:),i)
     END DO

     CALL Borrar_empty_leaves(head)
     CALL Calculate_masses(head)

     a = 0.0
     CALL Calculate_forces(head)
     v = v + a * dt/2

     t_out = t_out + dt
     IF (t_out >= dt_out) THEN
        DO i = 1,10
           PRINT*, r(i,:)
        END DO
        PRINT*, "-----------------------------------"
        PRINT*, ""
        t_out = 0.0
     END IF
  END DO

CONTAINS
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calculate_Ranges                       !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Calcula los rangos de las partículas en la
!! matriz r en las 3 dimensiones y lo pone en la
!! variable apuntada por goal
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE Calculate_Ranges(goal)
    TYPE(CELL),POINTER :: goal

    REAL, DIMENSION(3) :: mins,maxs,medios
    REAL :: span

    mins = MINVAL(r,DIM=1)
    maxs = MAXVAL(r,DIM=1)
      ! Al calcular span le sumo un 10% para que las
      ! particulas no caigan justo en el borde
    span = MAXVAL(maxs - mins) * 1.1
    medios = (maxs + mins) / 2.0
    goal%range%min = medios - span/2.0
    goal%range%max = medios + span/2.0
  END SUBROUTINE Calculate_Ranges

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Find_Cell                              !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Encuentra la celda donde colocaremos la particula.
!! Si la celda que estamos considerando no tiene
!! particula o tiene una particula, es esta celda donde
!! colocaremos la particula.
!! Si la celda que estamos considerando es un "conglomerado",
!! buscamos con la función BELONGS a que subcelda de las 8
!! posibles pertenece y con esta subcelda llamamos de nuevo
!! a Find_Cell
!!
!! NOTA: Cuando se crea una celda "conglomerado" se crean las
!! 8 subceldas, por lo que podemos asumir que siempre existen
!! las 8. Las celdas vacías se borran al final del todo, cuando
!! todo el árbol ha sido ya creado.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  RECURSIVE SUBROUTINE Find_Cell(root,goal,part)
    REAL, DIMENSION(3) :: part
    TYPE(CELL),POINTER :: root,goal,temp
    INTEGER :: i,j,k

    SELECT CASE (root%type)
       CASE (2)
          out: DO i = 1,2
             DO j = 1,2
                DO k = 1,2
                   IF (Belongs(part,root%subcell(i,j,k)%ptr)) THEN
                      CALL Find_Cell(root%subcell(i,j,k)%ptr,temp,part)
                      goal => temp
                      EXIT out
                   END IF
                END DO
             END DO
          END DO out
       CASE DEFAULT
          goal => root
     END SELECT
  END SUBROUTINE Find_Cell
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Place_Cell                             !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Se ejecuta tras Find_Cell, en la celda que
!! esa función nos devuelve, por lo que siempre
!! es una celda de tipo 0 (sin particula) o de tipo 1
!! (con una particula). En el caso de que es una celda
!! de tipo 1 habra que subdividir la celda y poner en
!! su lugar las dos particulas (la que originalmente
!! estaba, y la nueva).
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  RECURSIVE SUBROUTINE Place_Cell(goal,part,n)
    TYPE(CELL),POINTER :: goal,temp
    REAL, DIMENSION(3) :: part
    INTEGER :: n

    SELECT CASE (goal%type)
       CASE (0)
          goal%type = 1
          goal%part = part
          goal%pos = n
       CASE (1)
          CALL Crear_Subcells(goal)
          CALL Find_Cell(goal,temp,part)
          CALL Place_Cell(temp,part,n)
       CASE DEFAULT
          print*,"SHOULD NOT BE HERE. ERROR!"
    END SELECT
  END SUBROUTINE Place_Cell

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Crear_Subcells                         !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Esta funcion se llama desde Place_Cell y
!! solo se llama cuando ya hay una particula
!! en la celda, con lo que la tenemos que
!! subdividir. Lo que hace es crear 8 subceldas
!! que "cuelgan" de goal y la particula que
!! estaba en goal la pone en la subcelda que
!! corresponda de la 8 nuevas creadas.
!!
!! Para crear las subceldas utilizar las funciones
!! CALCULAR_RANGE, BELONGS y NULLIFY_POINTERS
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE Crear_Subcells(goal)
    TYPE(CELL), POINTER :: goal
    REAL,DIMENSION(3) :: part
    INTEGER :: i,j,k,n
    INTEGER, DIMENSION(3) :: octant

    part = goal%part
    goal%type=2

    DO i = 1,2
       DO j = 1,2
          DO k = 1,2
             octant = (/i,j,k/)
             ALLOCATE(goal%subcell(i,j,k)%ptr)
             goal%subcell(i,j,k)%ptr%range%min = Calcular_Range (0,goal,octant)
             goal%subcell(i,j,k)%ptr%range%max = Calcular_Range (1,goal,octant)

             IF (Belongs(part,goal%subcell(i,j,k)%ptr)) THEN
```

```
                goal%subcell(i,j,k)%ptr%part = part
                goal%subcell(i,j,k)%ptr%type = 1
                goal%subcell(i,j,k)%ptr%pos = goal%pos
             ELSE
                goal%subcell(i,j,k)%ptr%type = 0
             END IF
             CALL Nullify_Pointers(goal%subcell(i,j,k)%ptr)
          END DO
       END DO
    END DO
  END SUBROUTINE Crear_Subcells


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Nullify_Pointers                      !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Simplemente me NULLIFYca los punteros de
!! las 8 subceldas de la celda "goal"
!!
!! Se utiliza en el bucle principal y por
!! CREAR_SUBCELLS
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE Nullify_Pointers(goal)
    TYPE(CELL), POINTER :: goal
    INTEGER :: i,j,k

    DO i = 1,2
       DO j = 1,2
          DO k = 1,2
             NULLIFY(goal%subcell(i,j,k)%ptr)
          END DO
       END DO
    END DO
  END SUBROUTINE Nullify_Pointers


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Belongs                               !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Devuelve TRUE si la particula "part" está
!! dentro del rango de la celda "goal"
!!
!! Utilizada por FIND_CELL
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  FUNCTION Belongs (part,goal)
    REAL, DIMENSION(3) :: part
    TYPE(CELL), POINTER :: goal
    LOGICAL :: Belongs

    IF (part(1) >= goal%range%min(1) .AND. &
         part(1) <= goal%range%max(1) .AND. &
         part(2) >= goal%range%min(2) .AND. &
         part(2) <= goal%range%max(2) .AND. &
         part(3) >= goal%range%min(3) .AND. &
         part(3) <= goal%range%max(3)) THEN
       Belongs = .TRUE.
    ELSE
       Belongs = .FALSE.
    END IF
  END FUNCTION Belongs

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calcular_Range                        !!
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Dado un octante "otctant" (1,1,1, 1,1,2 ... 2,2,2),
!! calcula sus rangos en base a los rangos de
!! "goal". Si "what" = 0 calcula los minimos. Si what=1
!! calcula los maximos.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  FUNCTION Calcular_Range (what,goal,octant)
    INTEGER :: what,n
    TYPE(CELL), POINTER :: goal
    INTEGER, DIMENSION(3) :: octant
    REAL, DIMENSION(3) :: Calcular_Range, valor_medio

    valor_medio = (goal%range%min + goal%range%max) / 2.0

    SELECT CASE (what)
    CASE (0)
       WHERE (octant == 1)
          Calcular_Range = goal%range%min
       ELSEWHERE
          Calcular_Range = valor_medio
       ENDWHERE
    CASE (1)
       WHERE (octant == 1)
          Calcular_Range = valor_medio
       ELSEWHERE
          Calcular_Range = goal%range%max
       ENDWHERE
    END SELECT
  END FUNCTION Calcular_Range

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Borrar_empty_leaves                    !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Se llama una vez completado el arbol para
!! borrar (DEALLOCATE) las celdas vacías (i.e.
!! sin partícula).
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  RECURSIVE SUBROUTINE Borrar_empty_leaves(goal)
    TYPE(CELL),POINTER :: goal
    INTEGER :: i,j,k

    IF (ASSOCIATED(goal%subcell(1,1,1)%ptr)) THEN
       DO i = 1,2
          DO j = 1,2
             DO k = 1,2
                CALL Borrar_empty_leaves(goal%subcell(i,j,k)%ptr)
                IF (goal%subcell(i,j,k)%ptr%type == 0) THEN
                   DEALLOCATE (goal%subcell(i,j,k)%ptr)
                END IF
             END DO
          END DO
       END DO
    END IF
  END SUBROUTINE Borrar_empty_leaves

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Borrar_tree                            !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Borra el arbol completo, excepto la "head".
!!
!! El arbol se ha de regenerar continuamente,
!! por lo que tenemos que borrar el antiguo
!! para evitar "memory leaks".
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  RECURSIVE SUBROUTINE Borrar_tree(goal)
    TYPE(CELL),POINTER :: goal
    INTEGER :: i,j,k

       DO i = 1,2
          DO j = 1,2
             DO k = 1,2
                IF (ASSOCIATED(goal%subcell(i,j,k)%ptr)) THEN
                   CALL Borrar_tree(goal%subcell(i,j,k)%ptr)
                   DEALLOCATE (goal%subcell(i,j,k)%ptr)
                END IF
             END DO
          END DO
       END DO
  END SUBROUTINE Borrar_tree

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calculate_masses                       !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Nos calcula para todas las celdas que cuelgan
!! de "goal" su masa y su center-of-mass.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  RECURSIVE SUBROUTINE Calculate_masses(goal)
    TYPE(CELL),POINTER :: goal
    INTEGER :: i,j,k
    REAL :: mass
    REAL, DIMENSION(3) :: c_o_m

    goal%mass = 0
    goal%c_o_m = 0

    SELECT CASE (goal%type)
       CASE (1)
          goal%mass = m(goal%pos)
          goal%c_o_m = r(goal%pos,:)
       CASE (2)
          DO i = 1,2
             DO j = 1,2
                DO k = 1,2
                   IF (ASSOCIATED(goal%subcell(i,j,k)%ptr)) THEN
                      CALL Calculate_masses(goal%subcell(i,j,k)%ptr)
                      mass = goal%mass
                      goal%mass = goal%mass + goal%subcell(i,j,k)%ptr%mass
                      goal%c_o_m = (mass * goal%c_o_m + &
                           goal%subcell(i,j,k)%ptr%mass * goal%subcell(i,j,k)%ptr%c_o_m) /  goal%mass
                   END IF
                END DO
             END DO
          END DO
    END SELECT
  END SUBROUTINE Calculate_masses


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calculate_forces                       !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Calcula las fuerzas de todas las particulas contra "head".
!! Se sirve de la funcion Calculate_forces_aux que es la
!! que en realidad hace los calculos para cada particula
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE Calculate_forces(head)
```

```
    TYPE(CELL),POINTER :: head
    INTEGER :: i,j,k,start,end

    DO i = 1,n
       CALL Calculate_forces_aux(i,head)
    END DO

  END SUBROUTINE Calculate_forces

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calculate_forces_aux                  !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!! Dada una particula "goal" calcula las fuerzas
!! sobre ella de la celda "tree". Si "tree" es una
!! celda que contiene una sola particula el caso
!! es sencillo, pues se tratan de dos particulas.
!!
!! Si "tree" es una celda conglomerado, hay que ver primero
!! si l/D < theta. Es decir si el lado de la celda (l)
!! dividido entre la distancia de la particula goal
!! al center_of_mass de la celda tree (D) es menor que theta.
!! En caso de que asi sea, tratamos a la celda como una
!! sola particula. En caso de que no se menor que theta,
!! entonces tenemos que considerar todas las subceldas
!! de tree y para cada una de ellas llamar recursivamente
!! a Calculate_forces_aux
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  RECURSIVE SUBROUTINE Calculate_forces_aux(goal,tree)
    TYPE(CELL),POINTER :: tree
    INTEGER :: i,j,k,goal
    REAL :: l,D

    SELECT CASE (tree%type)
       CASE (1)
          IF (goal .NE. tree%pos) THEN
             rji = tree%c_o_m - r(goal,:)
             r2 = SUM(rji**2)
             r3 = r2 * SQRT(r2)
             a(goal,:) = a(goal,:) + m(tree%pos) * rji / r3
          END IF
       CASE (2)
            !! El rango tiene el mismo span en las 3 dimensiones
            !! por lo que podemos considerar una dimension cualquiera
            !! para calcular el lado de la celda (en este caso la
            !! dimension 1)
          l = tree%range%max(1) - tree%range%min(1)
          rji = tree%c_o_m - r(goal,:)
          r2 = SUM(rji**2)
          D = SQRT(r2)
          IF (l/D < theta) THEN
              !! Si conglomerado, tenemos que ver si se cumple l/D < @
             r3 = r2 * D
             a(goal,:) = a(goal,:) + tree%mass * rji / r3
          ELSE
             DO i = 1,2
                DO j = 1,2
                   DO k = 1,2
                      IF (ASSOCIATED(tree%subcell(i,j,k)%ptr)) THEN
                         CALL Calculate_forces_aux(goal,tree%subcell(i,j,k)%ptr)
                      END IF
                   END DO
                END DO
             END DO
          END IF
```

```
        END SELECT

      END SUBROUTINE Calculate_forces_aux

    END PROGRAM tree
```

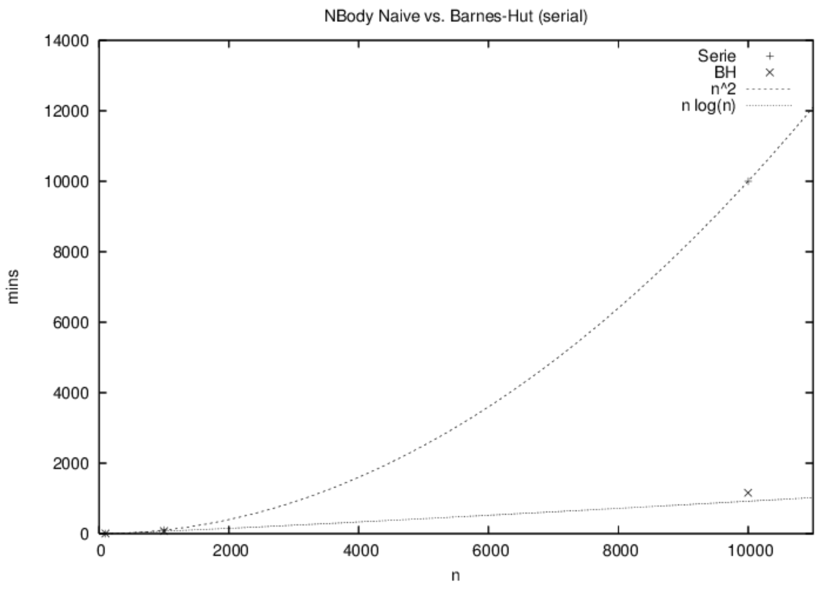


Figure 5.1: Performance Barnes Hut (1)

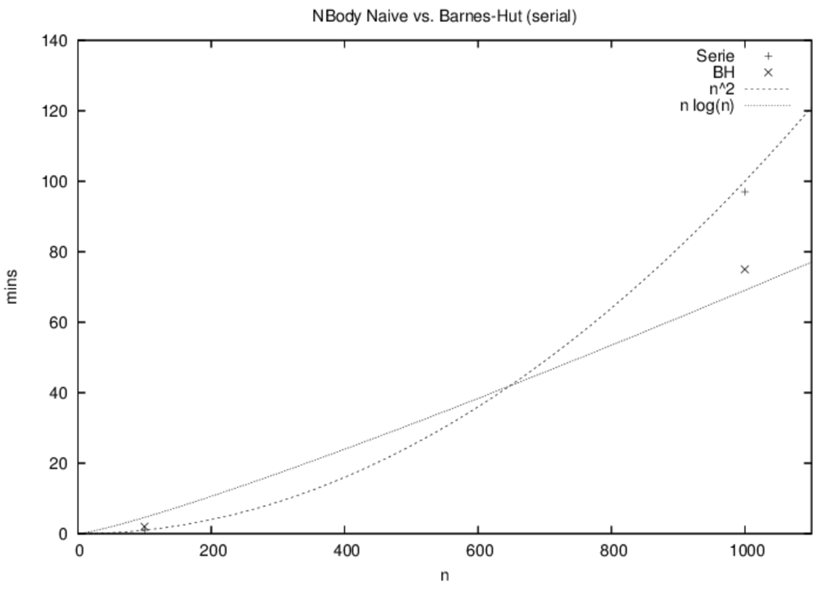


Figure 5.2: Performance Barnes Hut (2)

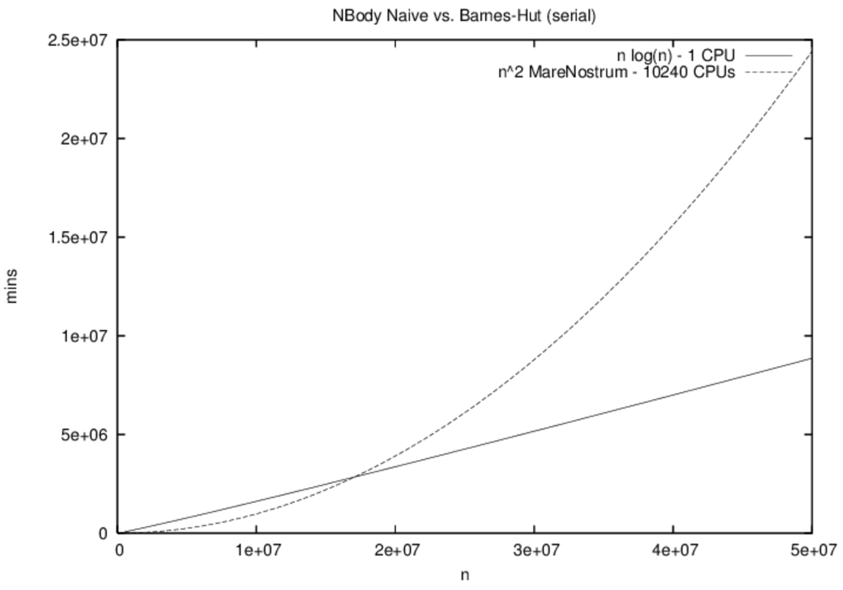


Figure 5.3: Performance Barnes Hut (3)

PART III

BARNES-HUT IN PARALLEL

Chapter 6

Getting started with MPI

In order to improve the performance of the Barnes-Hut algorithm learned in the previous chapter, we will now learn about parallel programming, so several computers can collaborate to complete the same problem.

6.1 Basic MPI

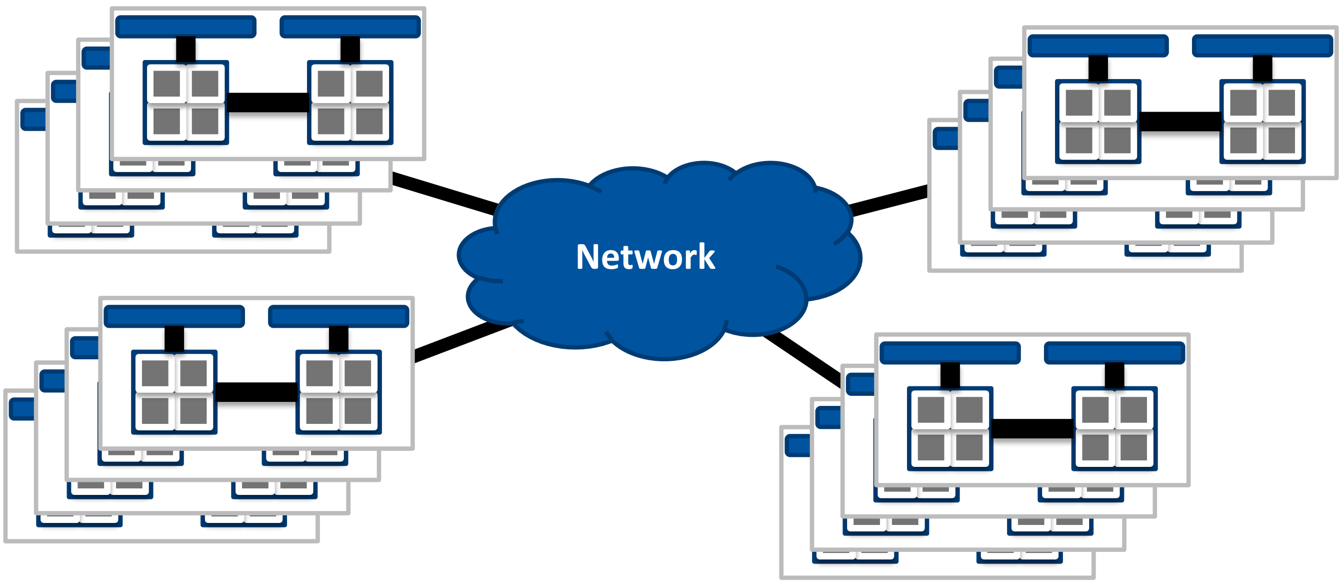
Message Passing with MPI

PPCES 2016

Hristo Iliev
IT Center / JARA-HPC

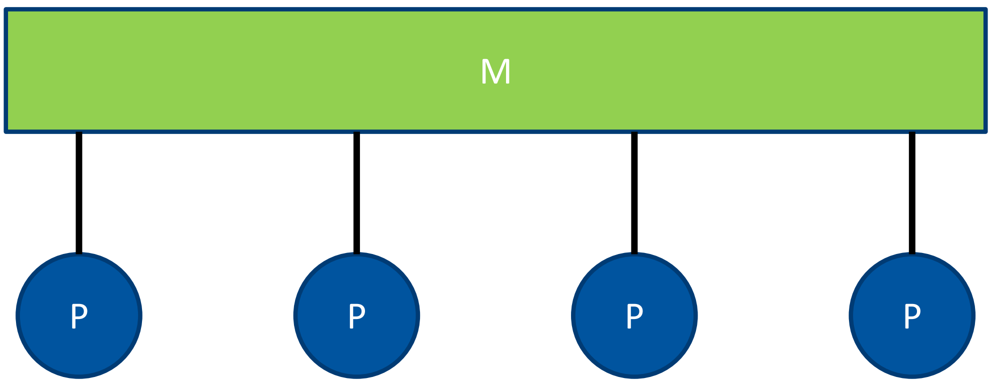
Parallel Architectures

- **Clusters**
  - → HPC market is at large dominated by distributed memory *multicomputers*: *clusters* and specialised *supercomputers*
  - → Nodes have no direct access to other nodes' memory and run a separate copy of the (possibly stripped down) OS



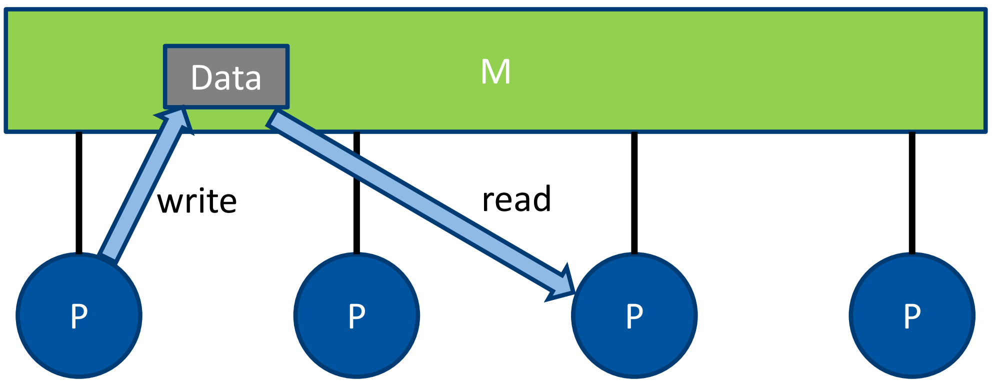
Parallel Architectures

- **Shared Memory**
  → All processing elements (P) have direct access to the main memory block (M)



Parallel Architectures

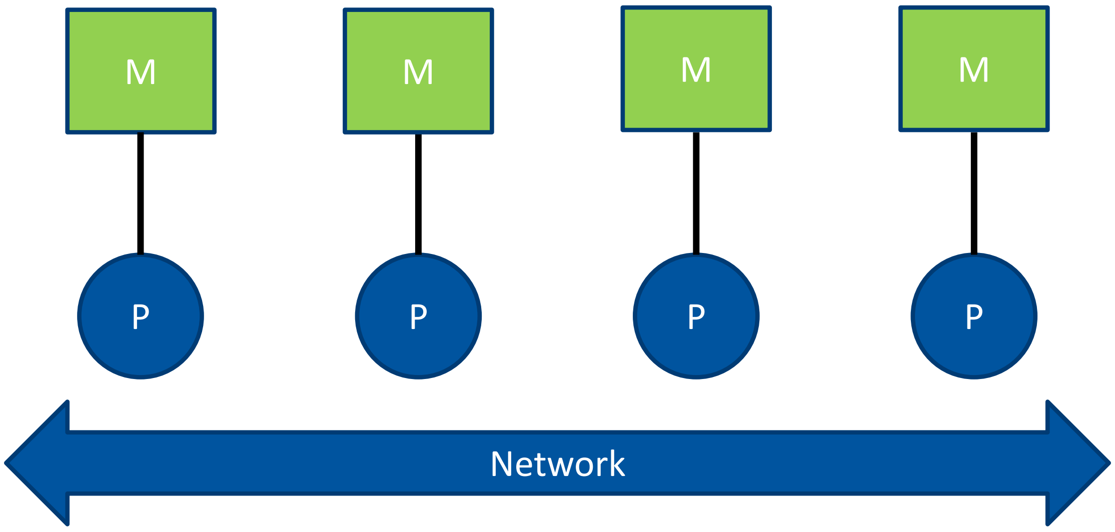
- **Shared Memory**
  - → All processing elements (P) have direct access to the main memory block (M)



  - → Data exchange is achieved through read/write operations on shared variables located in the global address space

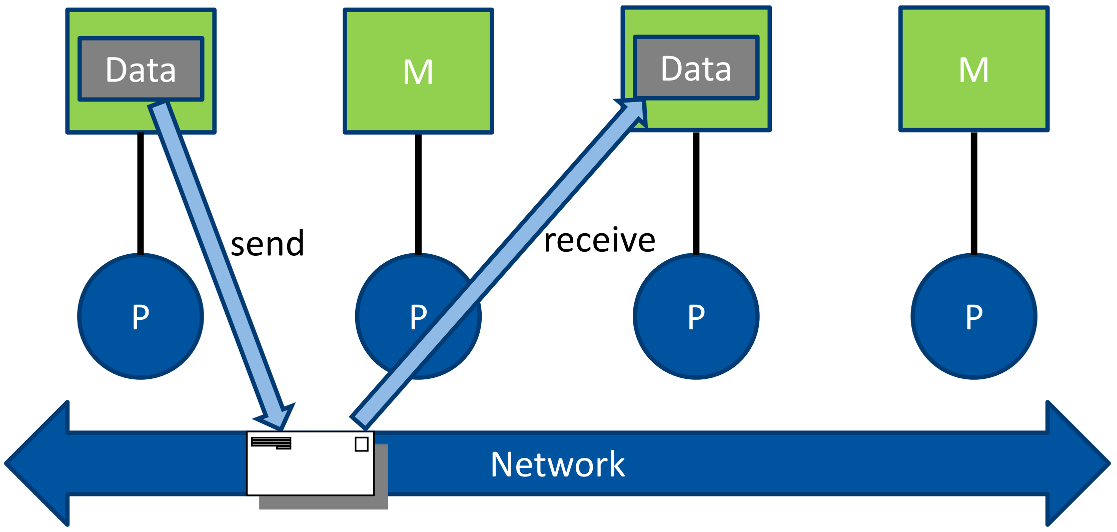
Parallel Architectures

- **Distributed Memory**
  - → Each processing element (P) has its own main memory block (M)



Parallel Architectures

- **Distributed Memory**
  - → Each processing element (P) has its own main memory block (M)



  - → Data exchange is achieved through message passing over the network

Parallel Architectures

- **Distributed Memory**
  - → Each processing element (P) has its own main memory block (M)
  - → Data exchange is achieved through message passing over the network
  - → Message passing could be either explicit (MPI) or implicit (PGAS)
  - → Programs typically implemented as a set of OS entities that have their own (virtual) address spaces – *processes*
  - → No shared variables
    - →No data races
    - →Explicit synchronisation mostly unneeded
      - →Results as “side effect” of the send-receive semantics

Processes

- **A process is a running in-memory instance of an executable**
  - → Executable code: e.g. binary machine instructions
  - → One or more threads of execution
  - → Memory: data, heap, stack, processor state (CPU registers and flags)
  - → Operating system context (e.g. signals, I/O handles, etc.)
  - → PID
- **Isolation and protection**
  - → A process cannot interoperate with other processes or access their context (even on the same node) without the help of the operating system
  - → No direct inter-process data exchange (virtual address spaces)
  - → No direct inter-process synchronisation

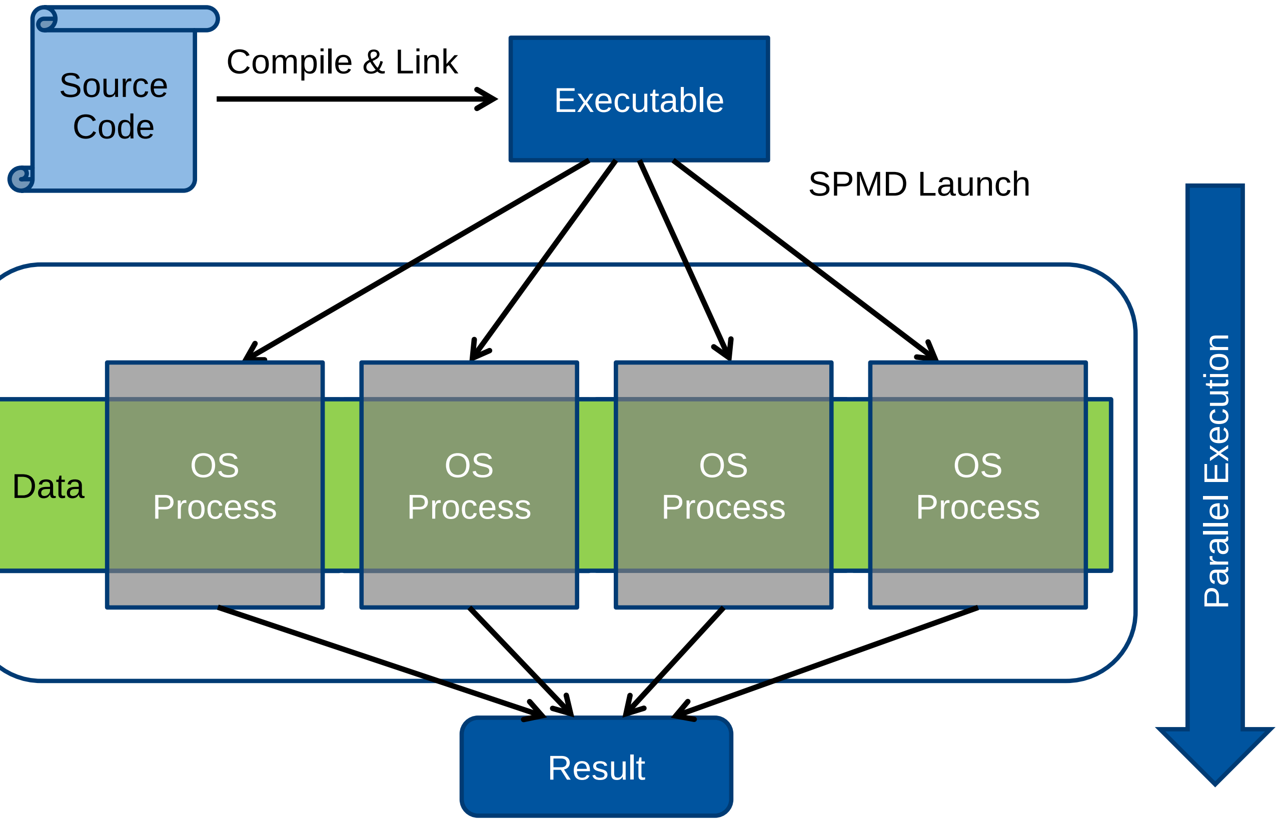
SPMD Model

- **Abstractions make programming and understanding easier**
- **Single Program Multiple Data**
  - → Multiple instruction flows (instances) from a Single Program working on Multiple (different parts of) Data
  - → Instances could be threads (OpenMP) and/or processes (MPI)
  - → Each instance receives a unique ID – can be used for flow control

```
if (myID == specificID)
{
    do something
}
else
{
    do something different
}
```

SPMD Model

- **SPMD Program Lifecycle – multiple processes (e.g. MPI)**



SPMD Environments

- **Provide dynamic identification of all peers**
  - → Who else is also working on this problem?
- **Provide robust mechanisms to exchange data**
  - → Whom to send data to / From whom to receive the data?
  - → How much data?
  - → What kind of data?
  - → Has the data arrived?
- **Provide synchronisation mechanisms**
  - → Have all processes reached same point in the program execution flow?
- **Provide methods to launch and control a set of processes**
  - → How do we start multiple processes and get them to work together?
- **Portability**

MPI

- **Message Passing Interface**
  - → The de-facto standard API for explicit message passing nowadays
  - → A moderately large standard (v3.1 is a 868 pages long)
  - → Maintained by the Message Passing Interface Forum

    http://www.mpi-forum.org/
- **Many concrete implementations of the MPI standard**
  - → Open MPI, MPICH, Intel MPI, MVAPICH, MS-MPI, etc.
- **MPI is used to describe the interaction (communication) in programs for computers with distributed memory**
- **MPI provides source level portability of parallel applications between different implementations and hardware platforms**

MPI

- **A language-independent specification (LIS) of a set of communication and I/O operations**
  - → Standard bindings for C and Fortran
    - → Concrete function prototypes / interfaces
  - → Non-standard bindings for other languages exist:

→C++	Boost.MPI
→Java	Open MPI, MPJ Express
→Python	mpi4py

- **Unlike e.g. OpenMP, MPI implementations are libraries (+ specialised runtimes) and make use of existing languages and compilers**

MPI History

- **Version 1.0 (1994): FORTRAN 77 and C bindings**
- **Version 1.1 (1995): Minor corrections and clarifications**
- **Version 1.2 (1997): Further corrections and clarifications**
- **Version 2.0 (1997): MPI-2 – Major extensions**
  - → One-sided communication
  - → Parallel I/O
  - → Dynamic process creation
  - → Fortran 90 and C++ bindings
  - → Language interoperability
- **Version 2.1 (2008): Merger of MPI-1 and MPI-2**
- **Version 2.2 (2009): Minor corrections and clarifications**
  - → C++ bindings deprecated
- **Version 3.0 (2012): Major enhancements**
  - → Non-blocking collective operations
  - → Modern Fortran 2008 bindings
  - → C++ deleted from the standard
- **Version 3.1 (2015): Corrections and clarifications**
  - → Portable operation with address variables
  - → Non-blocking collective I/O

More Information & Documentation

- The MPI Forum document archive (free standards for everyone!)
  → http://www.mpi-forum.org/docs/
- The MPI home page at Argonne National Lab
  → http://www-unix.mcs.anl.gov/mpi/
  → http://www.mcs.anl.gov/research/projects/mpi/www/
- Open MPI
  → http://www.open-mpi.org/
- Our MPI-related WEB page with further links (German only)
  → http://www.rz.rwth-aachen.de/mpi/
- Manual pages
  → man MPI
  → man MPI_Xxx_yyy_zzz (for all MPI calls)

General Structure of an MPI Program

- **Start-up, initialisation, finalisation, and shutdown – C**

C

```
(1)  #include <mpi.h>

(2)  int main(int argc, char **argv)
     {
       … some code …
(3)    MPI_Init(&argc, &argv);



(4)    … computation & communication …



(5)    MPI_Finalize();
(6)    … wrap-up …
       return 0;
     }
```

1. Inclusion of the MPI header file
2. Pre-initialisation mode: uncoordinated
   - **No MPI function calls allowed with few exceptions**
   - **All program instances run exactly the same code**
3. Initialisation of the MPI environment
   Implicit synchronisation
4. Parallel MPI code
   Typically computation and communication
5. Finalisation of the MPI environment
   Internal buffers are flushed
6. Post-finalisation mode: uncoordinated
   - **No MPI function calls allowed with few exceptions**

General Structure of an MPI Program

- **Start-up, initialisation, finalisation, and shutdown – Fortran**

Fortran

```
PROGRAM example
  USE mpi                                 ①
                                          ②
   … some code …
                                          ③
  CALL MPI_Init(ierr)
                                          ④
  … computation  & communication …
                                          ⑤
  CALL MPI_Finalize(ierr)
                                          ⑥
  … wrap-up …
END
```

① Inclusion of the MPI module

② Pre-initialisation mode: uncoordinated
- **No MPI function calls allowed with few exceptions**
- **All program instances run exactly the same code**

③ Initialisation of the MPI environment
Implicit synchronisation

④ Parallel MPI code
Typically computation and communication

⑤ Finalisation of the MPI environment
Internal buffers are flushed

⑥ Post-finalisation mode: uncoordinated
- **No MPI function calls allowed with few exceptions**

General Structure of an MPI Program

- How many processes are there in total?
- Who am I?

C

```
#include <mpi.h>

int main(int argc, char **argv)
{
  … some code …
  MPI_Init(&argc, &argv);
  … other code …
① MPI_Comm_size(MPI_COMM_WORLD,
       &numberOfProcs);
② MPI_Comm_rank(MPI_COMM_WORLD,
       &rank);
  … computation  & communication …
  MPI_Finalize();
  … wrap-up …
  return 0;
}
```

① Obtains the number of processes (ranks) in the MPI program

Example: if the job was started with 4 processes, then **numberOfProcs** will be set to 4 by the call

② Obtains the identity of the calling process within the MPI program
NB: MPI processes are numbered starting from 0

Example: if there are 4 processes in the job, then **rank** receive value of 0 in the first process, 1 in the second process, and so on

General Structure of an MPI Program

- **How many processes are there in total?**
- **Who am I?**

Fortran

```
PROGRAM example
  USE mpi
  INTEGER :: rank, numberOfProcs, ierr
  … some code …
  CALL MPI_Init(ierr)
  … other code …
① CALL MPI_Comm_size(MPI_COMM_WORLD,&
       numberOfProcs, ierr)
② CALL MPI_Comm_rank(MPI_COMM_WORLD,&
       rank, ierr)
  … computation  & communication …
  CALL MPI_Finalize(ierr)
  … wrap-up …
END PROGRAM example
```

① Obtains the number of processes (ranks) in the MPI program

Example: if the job was started with 4 processes, then **numberOfProcs** will be set to 4 by the call

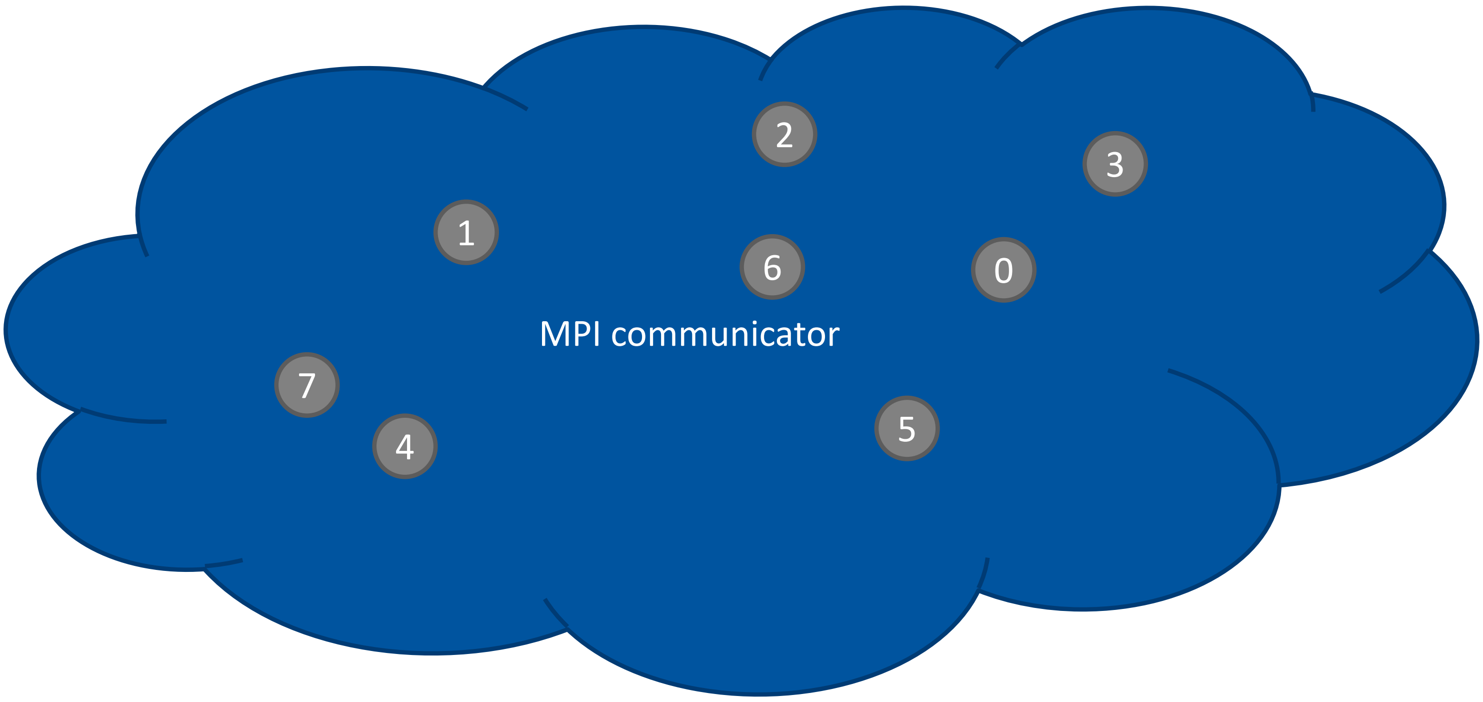
② Obtains the identity of the calling process within the MPI program

NB: MPI processes are numbered starting from 0

Example: if there are 4 processes in the job, then **rank** receive value of 0 in the first process, 1 in the second process, and so on

Ranks

- The processes in any MPI program are initially indistinguishable
- MPI_Init assigns each process a unique identity – rank



Ranks

- **The processes in any MPI program are initially indistinguishable**
- **MPI_Init assigns each process a unique identity – rank**
  - → Without personality, the started MPI processes cannot do coordinated parallel work in the pre-initialisation mode
  - → Ranks range from 0 up to the total number of processes minus 1
- **Ranks are associated with the so-called communicators**
  - → Logical contexts where communication takes place
  - → Represent groups of MPI processes with some additional information
  - → The most important one is the world communicator **MPI_COMM_WORLD**
    - → Contains all processes launched *initially* as part of the MPI program
  - → Ranks are always provided in MPI calls in combination with the corresponding communicator

Basic MPI Use

- **Initialisation:**

```
C:       MPI_Init(&argc, &argv);
Fortran: CALL MPI_Init(ierr)
```

→ Initialises the MPI library and makes the process member of the world communicator

→ [C] Modern MPI implementations allow both arguments to be NULL, otherwise they *must* point to the arguments of **main()**

→ May not be called more than once for the duration of the program execution

- **Finalisation:**

```
C:       MPI_Finalize();
Fortran: CALL MPI_Finalize(ierr)
```

→ Cleans up the MPI library and prepares the process for termination

→ Must be called once before the process terminates

→ Having other code after the finalisation call is not recommended

Basic MPI Use

- **Number of processes in the MPI program:**

```
C:       MPI_Comm_size(MPI_COMM_WORLD, &size);
Fortran: CALL MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

→ Obtains the number of processes initially started in the MPI program (the size of the world communicator)

→ **size** is an integer variable

→ **MPI_COMM_WORLD** is a predefined constant *MPI handle* that represents the world communicator

- **Process identification:**

```
C:       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
Fortran: CALL MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
```

→ Determines the rank (unique ID) of the process within the world communicator

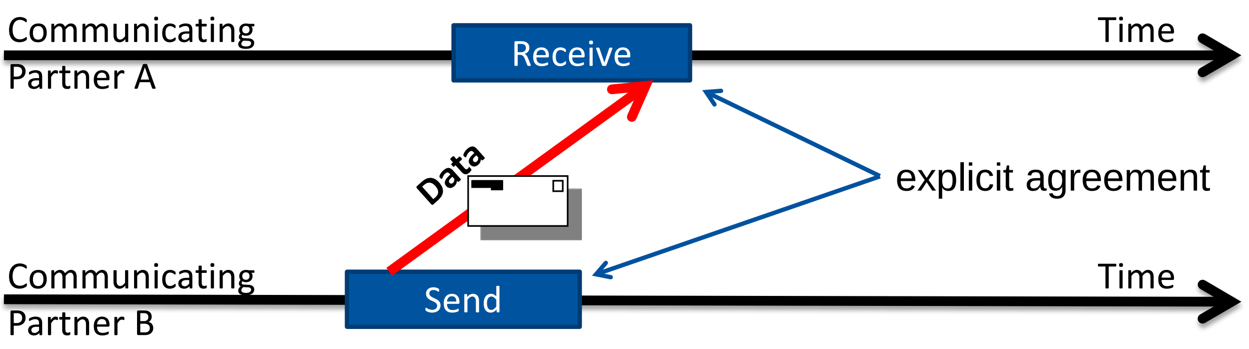
→ **rank** is an integer variable; receives value between 0 and #processes - 1

Basic MPI Use

- **Most C MPI calls return an integer error code:**
  - → **int MPI_Comm_size(…)**
- **Most Fortran MPI calls are subroutines with an extra INTEGER output argument (<u>always last one in the list</u>) for the error code:**
  - → **SUBROUTINE MPI_Comm_size (…, ierr)**
- **Error codes indicate the success of the operation:**
  - → Failure is indicated by error codes other than **MPI_SUCCESS**
  - → C: **if (MPI_SUCCESS != MPI_Init(NULL, NULL)) …**
  - → Fortran: **CALL MPI_Init(ierr)**
    IF (ierr /= MPI_SUCCESS) …
- **If an error occurs, an MPI error handler is called first before the call returns. The default error handler for non-I/O calls aborts the entire MPI program!**
- **NB: MPI error code values are implementation specific**

Message Passing

- **The goal is to enable communication between processes that share no memory space**



- **Explicit message passing requires:**
  - → Send and receive primitives (operations)
  - → Known addresses of both the sender and the receiver
  - → Specification of what has to be sent/received

Sending Data

- **Sending a message:**

What?

```
MPI_Send (void *data, int count, MPI_Datatype type,
          int dest, int tag, MPI_Comm comm)
```

To whom?

C

→ **data:**	location in memory of the data to be sent
→ **count:**	number of data elements to be sent (MPI is array-oriented)
→ **type:**	*MPI datatype* of the buffer content
→ **dest:**	rank of the receiver
→ **tag:**	additional identification of the message
	ranges from 0 to UB (impl. dependant but not less than 32767)
→ **comm:**	communication context (communicator)

```
MPI_Send (data, count, type, dest, tag, comm, ierr)
```

Fortran

Receiving Data

- **Receiving a message:**

What?

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

C

From whom?

→ **data:** location of the receive buffer

→ **count:** size of the receive buffer in data elements

→ **type:** MPI datatype of the data elements

→ **source:** rank of the sender or **MPI_ANY_SOURCE** (wildcard)

→ **tag:** message tag or **MPI_ANY_TAG** (wildcard)

→ **comm:** communication context

→ **status:** status of the receive operation or **MPI_STATUS_IGNORE**

```
MPI_Recv (data, count, type, src, tag, comm, status, ierr)
```

Fortran

MPI Datatypes

- **MPI is a library – it cannot infer the type of elements in the supplied buffer at run time and that's why it has to be told what it is**

- **MPI datatypes tell MPI how to:**
  - → read binary values from the send buffer
  - → write binary values into the receive buffer
  - → correctly apply value alignments
  - → convert between machine representations in heterogeneous environments

- **MPI datatype must match the language type(s) in the data buffer**
- **MPI datatypes are handles and cannot be used to declare variables**

MPI Datatypes

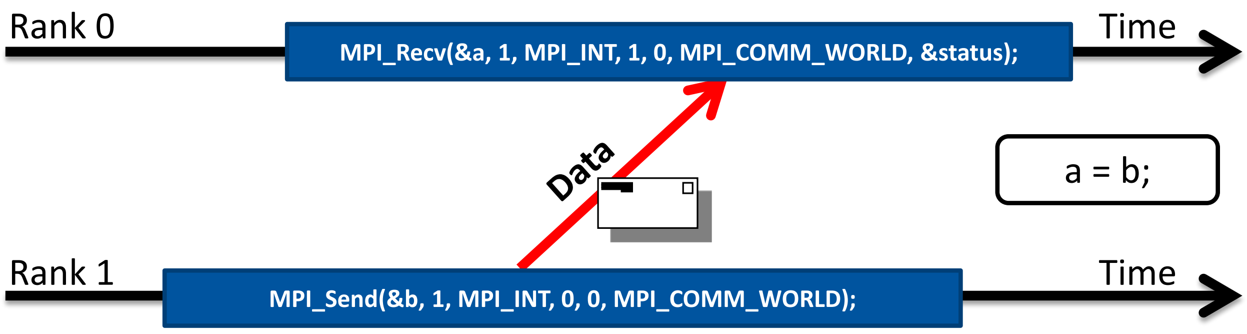
- MPI provides many predefined datatypes for each language binding:
  - → Fortran

MPI data type	Fortran data type
MPI_INTEGER	INTEGER
MPI_REAL MPI_REAL8	REAL REAL(KIND=8)
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
…	…
MPI_BYTE	-

8 binary digits
no conversion

Message Passing as Assignment

- **Message passing in MPI is explicit:**

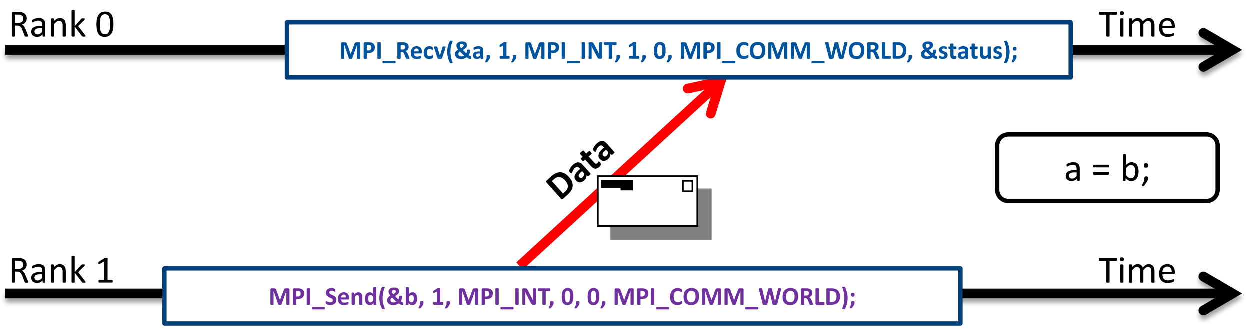

Rank 0
MPI_Recv(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
Time
Data
a = b;
Rank 1
MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
Time

- **The value of variable *b* in rank 1 is copied into variable *a* in rank 0**

- **For now, assume that *comm* is always MPI_COMM_WORLD**
  → We will talk about other communicators later on

Message Passing as Assignment

- Message passing in MPI is explicit:


Rank 0
MPI_Recv(&a, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
Time
Data
a = b;
Rank 1
MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
Time

```
if (rank == 0) {
  MPI_Recv(&a, 1, MPI_INT, 1, 0,
           MPI_COMM_WORLD, &status);
}
else if (rank == 1) {
  MPI_Send(&b, 1, MPI_INT, 0, 0,
           MPI_COMM_WORLD);
}
```

Complete MPI Example

C

```
#include <mpi.h>

int main(int argc, char **argv)
{
  int nprocs, rank, data;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,
      &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,
      &rank);
  if (rank == 0)
    MPI_Recv(&data, 1, MPI_INT, 1, 0,
      MPI_COMM_WORLD, &status);
  else if (rank == 1)
    MPI_Send(&data, 1, MPI_INT, 0, 0,
      MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

1. Initialise the MPI library
2. Identify current process
3. Behave differently based on the rank
4. Communicate
5. Clean up the MPI library

Compiler Wrappers

- MPI is a typical library with C header files, Fortran modules, etc.
- Some MPI vendors provide convenience compiler wrappers:

$$\begin{tikzcd}
\boxed{\texttt{cc}} \arrow[r] & \boxed{\texttt{mpicc}} \\
\boxed{\texttt{c++}} \arrow[r] & \boxed{\texttt{mpic++}} \\
\boxed{\texttt{f90}} \arrow[r] & \boxed{\texttt{mpif90}}
\end{tikzcd}$$

- On RWTH Compute Cluster (depending on the loaded modules):

$$\begin{tikzcd}
\boxed{\texttt{\$CC}} \arrow[r] & \boxed{\texttt{\$MPICC}} \\
\boxed{\texttt{\$CXX}} \arrow[r] & \boxed{\texttt{\$MPICXX}} \\
\boxed{\texttt{\$FC}} \arrow[r] & \boxed{\texttt{\$MPIFC}}
\end{tikzcd}$$

Execution of MPI Programs

- **Most MPI implementations provide a special launcher program:**

```
mpiexec -n nprocs … program <arg1> <arg2> <arg3> …
```

  → launches **nprocs** instances of **program** with command-line arguments **arg1, arg2, …** and provides the MPI library with enough information in order to establish network connections between the processes

  → Sometimes called **mpirun**

- **The launcher often performs more than simply launching processes:**

  → Helps MPI processes find each other and establish the world communicator

  → Redirects the standard output of all ranks to the terminal

  → Redirects the terminal input to the standard input of rank 0

  → Forwards received signals (Unix-specific)

6.1.1 MPI parallel programs in the CCA

To run a basic parallel “Hello World!” program in the CCA machines, you would need to follow these steps:

1. Write the hello-world.f90 code:

```
program hello_world
  use mpi
  implicit none

  integer error
  integer, parameter :: master = 0
  integer num_procs
  integer world_id

  call MPI_Init ( error )
  call MPI_Comm_size ( MPI_COMM_WORLD, num_procs, error )
  call MPI_Comm_rank ( MPI_COMM_WORLD, world_id, error )

  if ( world_id == master ) then
     print*, "I’m the master of ", num_procs, "processes"
  end if

  print*, "Process ", world_id, ’ says "Hello, world!"’

  call MPI_Finalize ( error )

end program hello_world
```

2. Compile it using the mpif90 wrapper

```
[angelv@opel]$ mpif90 -o hello-world hello-world.f90
```

3. Then you just execute it, selecting the number of processes to create with the -np option

```
[angelv@opel]$ mpirun -np 4 ./hello-world
I’m the master of            4 processes
Process           0  says "Hello, world!"
Process           1  says "Hello, world!"
Process           2  says "Hello, world!"
Process           3  says "Hello, world!"
[angelv@opel]$
```

Exercise #1: Hello World

- Write a minimal MPI program which prints “hello world”
- Run it on several processors in parallel
- Modify your program so that only the process ranked 2 in MPI_COMM_WORLD prints out “hello world”
- Modify your program so that each process prints out its rank and the total number of processors

Message Envelope and Matching

- Reception of MPI messages is done by matching their envelope
- Send operation

```
MPI_Send (void *data, int count, MPI_Datatype type,
          int dest, int tag, MPI_Comm comm)
```

- Message Envelope:

	Sender	Receiver
Source	Implicit	Explicit, wildcard possible (MPI_ANY_SOURCE)
Destination	Explicit	Implicit
Tag	Explicit	Explicit, wildcard possible (MPI_ANY_TAG)
Communicator	Explicit	Explicit

Message Envelope

- Receive operation

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

Message Envelope and Matching

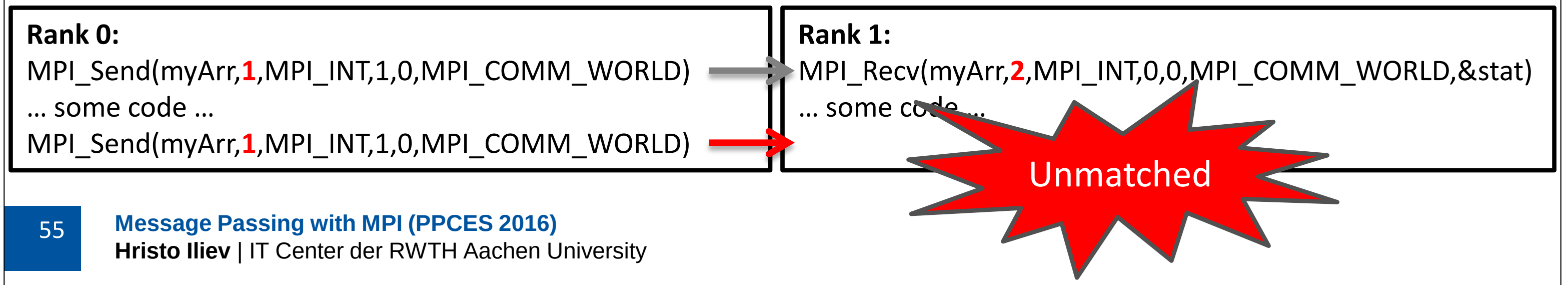
- Reception of MPI messages is also dependent on the data.
- Recall:

```
MPI_Send (void *data, int count, MPI_Datatype type,
          int dest, int tag, MPI_Comm comm)
```

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

- The standard expects datatypes at both ends to match
  → Not enforced by most implementations
- Matching sends and receives must always come in pairs

- NB: messages do not aggregate



Rank 0:

```
MPI_Send(myArr,1,MPI_INT,1,0,MPI_COMM_WORLD)
… some code …
MPI_Send(myArr,1,MPI_INT,1,0,MPI_COMM_WORLD)
```

Rank 1:

```
MPI_Recv(myArr,2,MPI_INT,0,0,MPI_COMM_WORLD,&stat)
… some code …
```

Message Reception and Status

- **The receive buffer must be able to fit the entire message**

→ send count ≤ receive count	OK (but check status)
→ send count > receive count	ERROR (message truncation)

- **The MPI status object holds information about the received message**

- **Fortran: INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status**

→ **status(MPI_SOURCE)**	message source rank
→ **status(MPI_TAG)**	message tag
→ **status(MPI_ERROR)**	receive status code

Inquiry Operations

- **Blocks until a matching message appears:**

```
MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status *status)
```

→ Message is not received, one must call **MPI_Recv** to receive it

→ Information about the message is stored in the status field

```
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status);
```

→ Checks for any message in the given communicator

- **Message size inquiry:**

```
MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)
```

→ Calculates how many integral **datatype** elements can be formed from the data in the message referenced by **status**

→ If the number is not integral, **count** is set to **MPI_UNDEFINED**

→ Can be used with the status from **MPI_Recv** too

Operation Completion

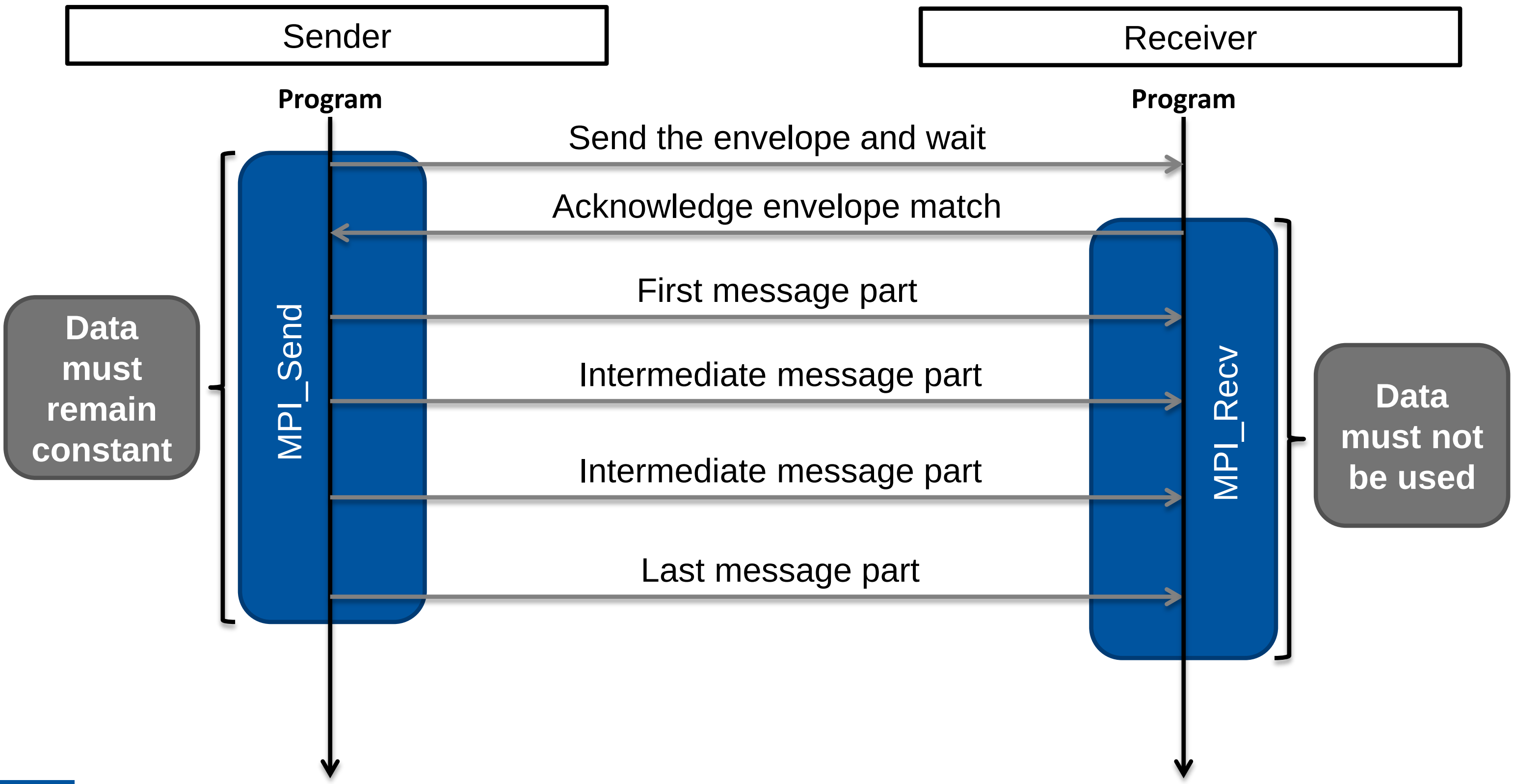
- **MPI operations complete then, when the message buffer is no longer in use by the MPI library and is free for reuse**

- **Send operations complete:**
  - → once the message is constructed ***and***
    - →sent completely to the network ***or***
    - →buffered completely (by MPI, the OS, the network, …)
- **Receive operations complete:**
  - → once the entire message has arrived and has been placed into the buffer

- **Blocking MPI calls only return once the operation has completed**
  - → **MPI_Send** and **MPI_Recv** are blocking

Blocking Calls

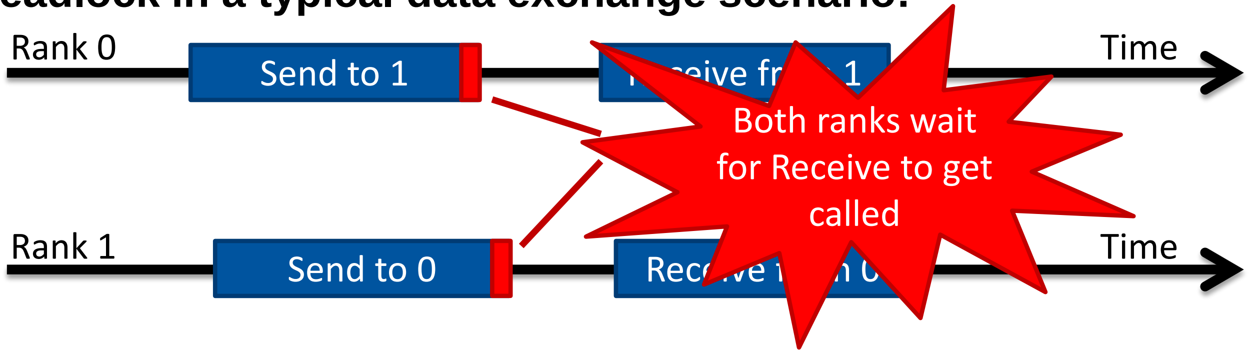
- **Blocking send (w/o buffering) and receive calls:**


Sender
Receiver
Program
Program
Send the envelope and wait
Acknowledge envelope match
First message part
Intermediate message part
Intermediate message part
Last message part
MPI_Send
MPI_Recv
Data must remain constant
Data must not be used

Deadlocks

- **Both MPI_Send and MPI_Recv calls are blocking:**
  - → The receive operation only returns after a matching message has arrived
  - → The send operation ***might*** be buffered *(implementation-specific!!!)* and therefore return before the message is actually sent to the network
  - → Larger messages are usually sent only when both the send and the receive operations are active (synchronously)
  - → **Never rely on any implementation-specific behaviour!!!**

- **Deadlock in a typical data exchange scenario:**



Message Ordering

- **Order is preserved in a given communicator for point-to-point operations between any pair of processes**
  - → Sends in same communicator and to the same rank are non-overtaking
  - → Probe/receive returns the earliest matching message
- **Order is not preserved for**
  - → messages sent in different communicators
  - → messages from different senders

```
MPI_Status status;

MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
… allocate buffer based on message size …
MPI_Recv(buffer, size, MPI_INT, status.MPI_SOURCE, 0,
         MPI_COMM_WORLD, &status);
```

Also applies to sequences of wildcard receives

Common Pitfalls – Fortran 90

- **Non-contiguous array sections should not be passed to non-blocking MPI calls**

```
INTEGER, DIMENSION(10,10) :: mat

! Probably OK
CALL MPI_Isend(mat(:,1:3), …

! NOT OK
CALL MPI_Isend(mat(1:3,:), …
! NOT OK
CALL MPI_Isend(mat(1:3,1:3), …
```

A temporary contiguous array is created and passed to MPI. It might get destroyed on return from the call before the actual send is complete!

- **Solved in MPI-3.0 with the introduction of the new Fortran 2008 interface *mpi_f08*, which allows array sections to be passed**

Sample Program #1 - Fortran

```
      PROGRAM p2p
C Run with two processes
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      real data(100)
      real value(200)
      integer status(MPI_STATUS_SIZE)
      integer count
      CALL MPI_INIT(err)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
      if (rank.eq.1) then
         data=3.0
         call MPI_SEND(data,100,MPI_REAL,0,55,MPI_COMM_WORLD,err)
       else
         call MPI_RECV(value,200,MPI_REAL,MPI_ANY_SOURCE,55,
     &                MPI_COMM_WORLD,status,err)
         print *, "P:",rank," got data from processor ",
     &                status(MPI_SOURCE)
         call MPI_GET_COUNT(status,MPI_REAL,count,err)
         print *, "P:",rank," got ",count," elements"
         print *, "P:",rank," value(5)=",value(5)
      end if
      CALL MPI_FINALIZE(err)
      END
```

```
           Program Output
P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=3.
```

6.1.2 MPI Parallel code to calculate an integral using the trapezoidal rule

```
!  trap.f -- Parallel Trapezoidal Rule, first version
!
! Slightly modified by Angel de Vicente
!
!  Input: None.
!  Output:  Estimate of the integral from a to b of f(x)
!     using the trapezoidal rule and n trapezoids.
!
!  Algorithm:
!     1.  Each process calculates "its" interval of
!         integration.
!     2.  Each process estimates the integral of f(x)
!         over its interval using the trapezoidal rule.
!     3a. Each process != 0 sends its integral to 0.
!     3b. Process 0 sums the calculations received from
!         the individual processes and prints the result.
!
!  Notes:
!     1.  f(x), a, b, and n are all hardwired.
!     2.  Assumes number of processes (p) evenly divides
!         number of trapezoids (n = 1024)
!
! See section 3.2 de "MPI User's Guide in Fortran"
!
PROGRAM trapezoidal
  USE MPI
  INTEGER :: n=1024, dest=0, tag=0
  REAL :: a=0.0, b=1.0
  INTEGER :: my_rank, p, local_n, source, status(MPI_STATUS_SIZE), ierr
  REAL :: h, local_a, local_b, integral, total

  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, p, ierr)

  h = (b-a)/n
  local_n = n/p

  local_a = a + my_rank*local_n*h
  local_b = local_a + local_n*h
  integral = Trap(local_a, local_b, local_n, h)

  IF (my_rank .EQ. 0) THEN
     total = integral
     DO source = 1, p-1
        CALL MPI_RECV(integral, 1, MPI_REAL, source, tag, MPI_COMM_WORLD, status, ierr)
        total = total + integral
     END DO
  ELSE
     CALL MPI_SEND(integral, 1, MPI_REAL, dest, tag, MPI_COMM_WORLD, ierr)
  END IF

  IF (my_rank .EQ. 0) THEN
     PRINT*, "With n = ", n, " trapezoids, our estimate"
     PRINT*, "of the integral from ", a, " to ", b, " = ", total
  END IF

  CALL MPI_FINALIZE(ierr)

CONTAINS
  REAL FUNCTION f(x)
    REAL :: x

    f = x*x
  END FUNCTION f


  REAL FUNCTION Trap(local_a, local_b, local_n, h)
    REAL ::   local_a, local_b, h, integral, x, i
    INTEGER :: local_n

    integral = (f(local_a) + f(local_b))/2.0
    x = local_a
    DO i = 1, local_n-1
       x = x + h
       integral = integral + f(x)
    END DO
    Trap = integral*h
  END FUNCTION Trap

END PROGRAM trapezoidal
```

6.2 Basic MPI Exercises - Point-to-Point

Below there are some exercises to familiarize yourself with the basic features of MPI, using the most basic point-to-point routines (those in which only two processes take part in the communication), as seen in section 6.1.

You can find solutions to these exercises in section A.3, but you are *highly encouraged* to try to solve them first on your own.

6.2.1 Simple modification to "Hello World!"

In this exercise, we will make a simple modification to the "Hello World!" problem.

In this case, we want that each process with rank>0 sends a message to the process with rank==0. The message is just its rank, and process 0 will just print a message saying that it received the message. (You will have to use the routines MPI_Send and MPI_Recv).

Thus, for example, if we run the code with 4 processes, we could get the following output:

```
Greetings from process 1 !
Greetings from process 2 !
Greetings from process 3 !
```

Remember that the code is meant to run with any number of processes, not only 4. Also, make sure you understand if the messages should be printed in consecutive order or not, and in any case why.

6.2.2 Ring data passing

Write a parallel program (valid for any number N of processes), in which:

- rank 0 reads a number
- each process with rank==r sends to the process with rank==r+1 the number received (read in the case of rank=0) multiplied by r+1, except the last process (rank==N-1), who sends is to the process with rank=0
- rank 0 prints the number that it receives from process with rank==N-1

6.2.3 Parallel sum

Write a parallel program to compute the sum of a 1-D array:

- rank 0 reads the array 1-D array size
- rank 0 allocates an array with the given size and reads the data
- rank 0 calculates the number of data that each rank will be responsible for (we can assume that the array size is divisible by the number of processes running the code)
- rank 0 sends to each process the number of items it will responsible for and the actual data (which each process will have to store in an temporary array,
- each process will calculate the partial sum of its array chunk and will send the partial result to rank 0
- rank 0 will calculate the total sum and print the result.

6.2.4 Matrix distribution

This exercise was taken from the ”Training MPI” course by CINECA-SCAI. For details see `http://www.hpc.cineca.it/content/exercise-5`

Distribute a global square NxN matrix (with N fixed) over P processors, so that each task has a local portion of it in its memory. Initialize such portion with the task’s rank.

NOTE: the exercise does not state that P has to be a divisor of N. How to deal with the matrix distribution if the number of rows/columns isn’t the same for all tasks?

Each task sends its first and last columns (if Fortran) or rows (if C) to its neighbours (i.e.: the first column/row to the left processor, the last column/row to the right processor). Note that each task has to actually allocate a larger portion of its submatrix (ghost cells), with two extra columns/rows at the extremities to hold the received values.

6.2.5 Identity matrix distribution

This exercise was taken from the ”Training MPI” course by CINECA-SCAI. For details see `http://www.hpc.cineca.it/content/exercise-6`

Write a program that performs a data distribution over the processes of the identity matrix. Given the number of processes and the dimension of the identity matrix, each process must allocate its own portion of the matrix. Distribute the matrix by columns (FORTRAN).

First, obtain the number, N, of columns you need to allocate for each process, np. In case N is not divisible by np, take care of the remainder with the module operator (function).

Now you need to implement a transformation from global to local (task) coordinates. You can use a a varable, iglob, that shifts the values to be initialized to 1 according to the global coordinates.

To check if everything is correct make the process 0 collect all matrix blocks initialized by the other processes and print out the matrix.

Chapter 7

MPI collective operations

This chapter builds on chapter 6. The concepts are just the same, but we learn how to exchange messages not only from one process to another, but "collective" messages, those involving all processes in a communicator (in our case, always the communicator *MPI_COMM_WORLD*, i.e. all processes).

7.1 MPI collective routines

Collective Communication

*includes sample C and Fortran programs

Collective Communication

- Communications involving a group of processes
- Called by *all* processes in a communicator
- Examples:
  - Broadcast, scatter, gather (Data Distribution)
  - Global sum, global maximum, etc. (Collective Operations)
  - Barrier synchronization

Characteristics of Collective Communication

- Collective communication will not interfere with point-to-point communication and vice-versa
- All processes must call the collective routine
- Synchronization not guaranteed (except for barrier)
- No non-blocking collective communication
- No tags
- Receive buffers must be exactly the right size

Barrier Synchronization

- Red light for each processor: turns green when all processors have arrived
- Slower than hardware barriers (example: SGI/Cray T3E)

C:

```
int MPI_Barrier (MPI_Comm comm)
```

Fortran:

```
CALL MPI_BARRIER (COMM,IERROR)
INTEGER COMM,IERROR
```

Broadcast

- One-to-all communication: same data sent from root process to all the others in the communicator

- C:

```
int MPI_Bcast (void *buffer, int, count,
  MPI_Datatype datatype,int root, MPI_Comm comm)
```

- Fortran:

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM IERROR)

<type> BUFFER (*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

- All processes must specify same root rank and communicator

Sample Program #5 - Fortran

```fortran
PROGRAM broadcast
INCLUDE 'mpif.h'
INTEGER err, rank, size
real param
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
if(rank.eq.5) param=23.0
call MPI_BCAST(param,1,MPI_REAL,5,MPI_COMM_WORLD,err)
print *,"P:",rank," after broadcast param is ",param
CALL MPI_FINALIZE(err)
END
```

```
             Program Output
P:1 after broadcast parameter is 23.
P:3 after broadcast parameter is 23.
P:4 after broadcast parameter is 23
P:0 after broadcast parameter is 23
P:5 after broadcast parameter is 23.
P:6 after broadcast parameter is 23.
P:7 after broadcast parameter is 23.
P:2 after broadcast parameter is 23.
```

Scatter

- One-to-all communication: different data sent to each process in the communicator (in rank order)

C:

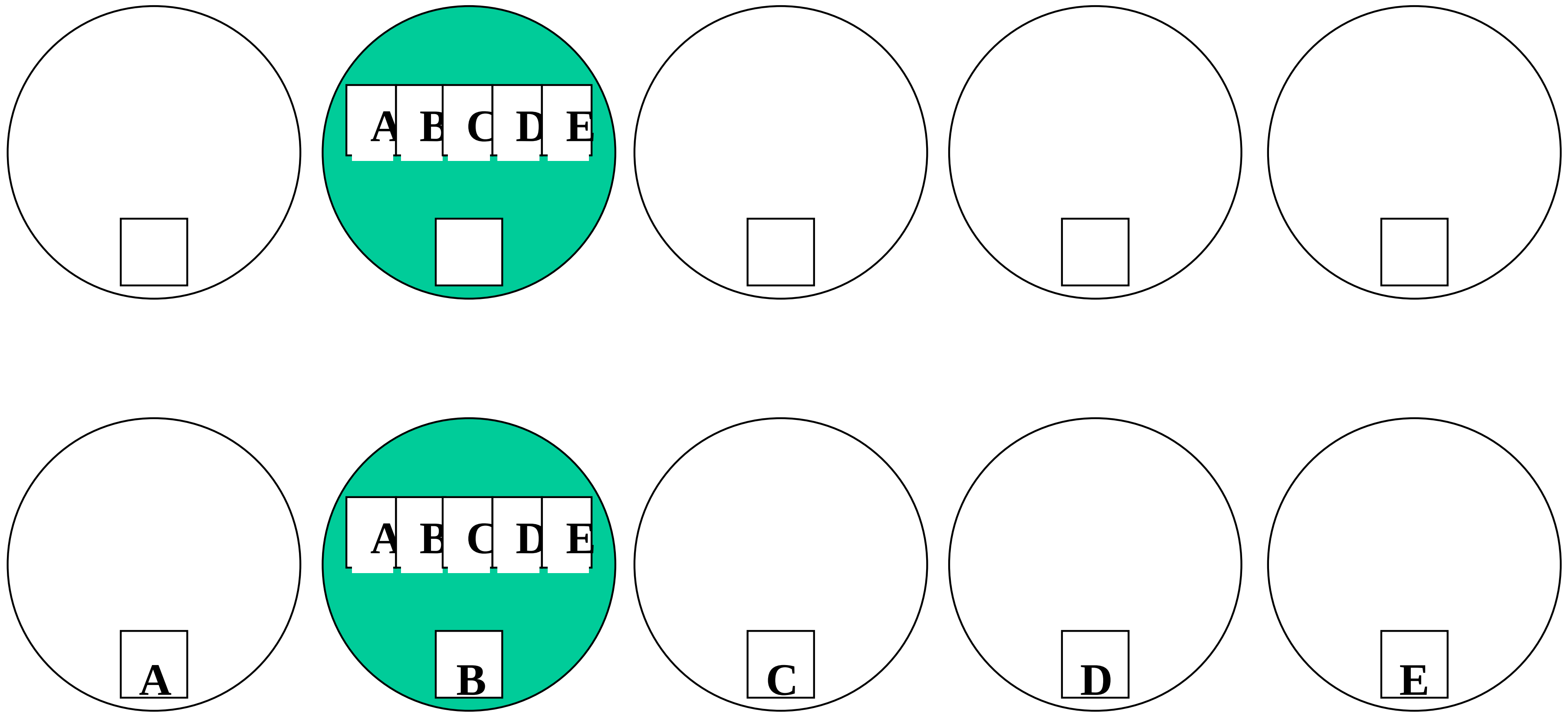
```
int MPI_Scatter(void* sendbuf, int sendcount,
        MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Fortran:

```
CALL MPI_SCATTER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
        RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
```

- `sendcount` is the number of elements sent to each process, not the "total" number sent
  - send arguments are significant only at the root process

Scatter Example



Sample Program #6 - Fortran

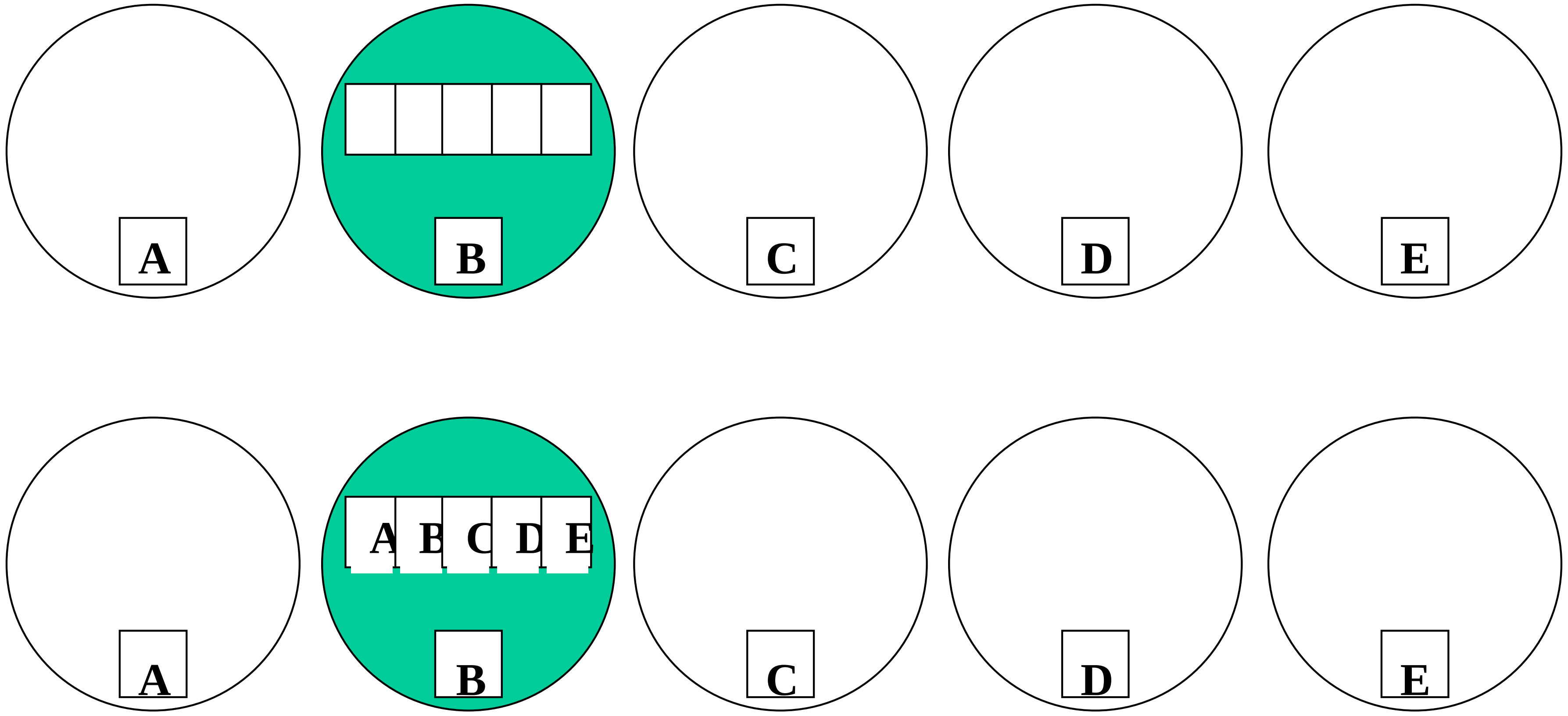
```
 PROGRAM scatter
 INCLUDE 'mpif.h'
 INTEGER err, rank, size
 real param(4), mine
 integer sndcnt,rcvcnt
 CALL MPI_INIT(err)
 CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
 CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
 rcvcnt=1
 if(rank.eq.3) then
    do i=1,4
       param(i)=23.0+i
    end do
    sndcnt=1
 end if
 call MPI_SCATTER(param,sndcnt,MPI_REAL,mine,rcvcnt,MPI_REAL,
&            3,MPI_COMM_WORLD,err)
 print *,"P:",rank," mine is ",mine
 CALL MPI_FINALIZE(err)
 END
```

```
   Program Output
P:1 mine is 25.
P:3 mine is 27.
P:0 mine is 24.
P:2 mine is 26.
```

Gather

- All-to-one communication: different data collected by root process
  - Collection done in rank order
- `MPI_GATHER` & `MPI_Gather` have same arguments as matching scatter routines
- Receive arguments only meaningful at the root process

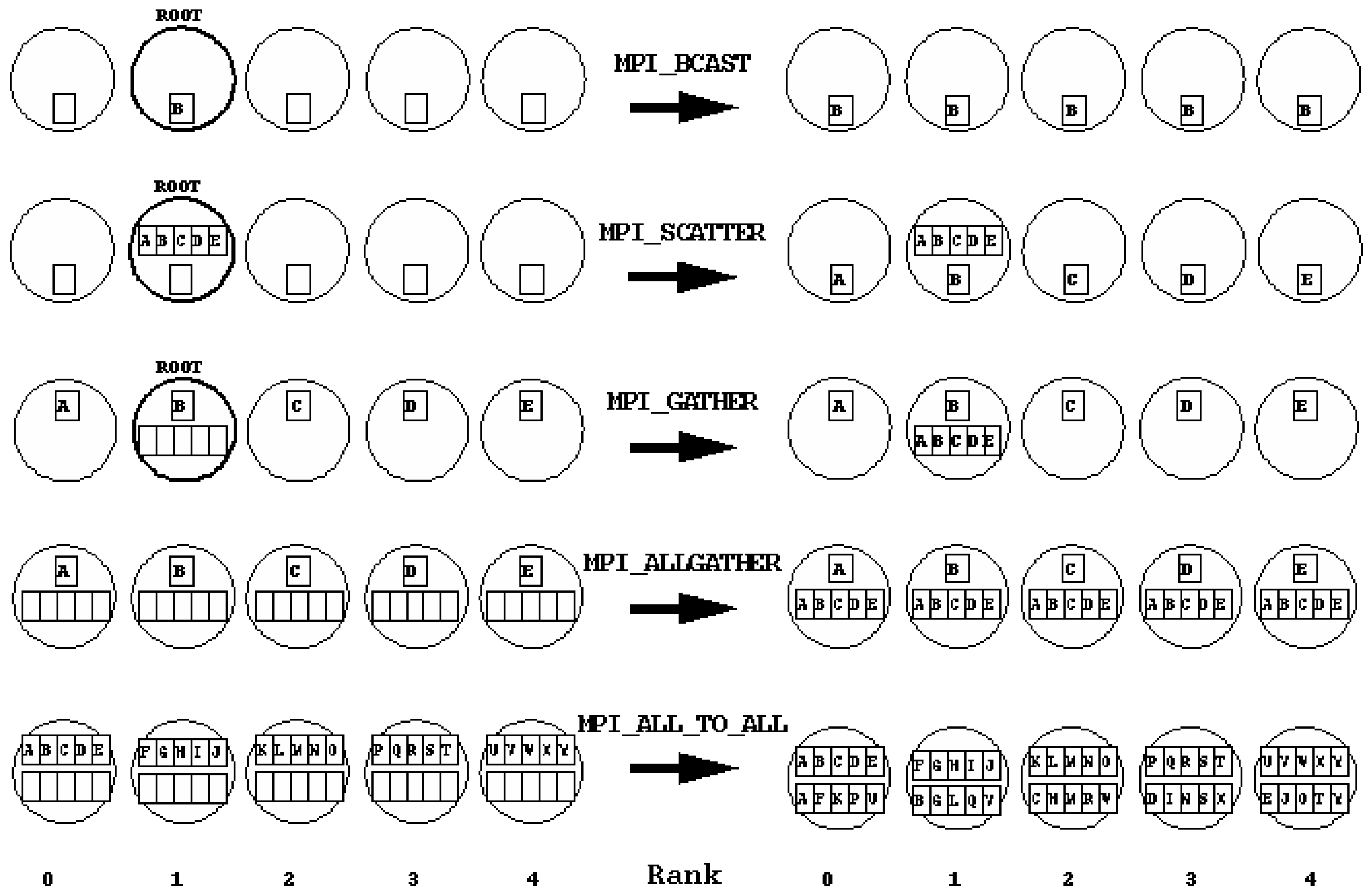
Gather Example



Gather/Scatter Variations

- `MPI_Allgather`
- `MPI_Alltoall`
- No root process specified: all processes get gathered or scattered data
- Send and receive arguments significant for all processes

Summary


ROOT
MPI_BCAST
MPI_SCATTER
MPI_GATHER
MPI_ALLGATHER
MPI_ALL_TO_ALL
Rank
0
1
2
3
4

Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes
- Examples:
  - Global sum or product
  - Global maximum or minimum
  - Global user-defined operation

Example of Global Reduction

- Sum of all the `x` values is placed in `result` only on processor 0

C:

```
MPI_Reduce (&x, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```

Fortran:

```
CALL MPI_REDUCE (x,result,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,IERROR)
INTEGER COMM,IERROR
```

Predefined Reduction Operations

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

General Form

- count is the number of "*ops*" done on consecutive elements of sendbuf (it is also size of recvbuf)

- op is an associative operator that takes two operands of type datatype and returns a result of the same type

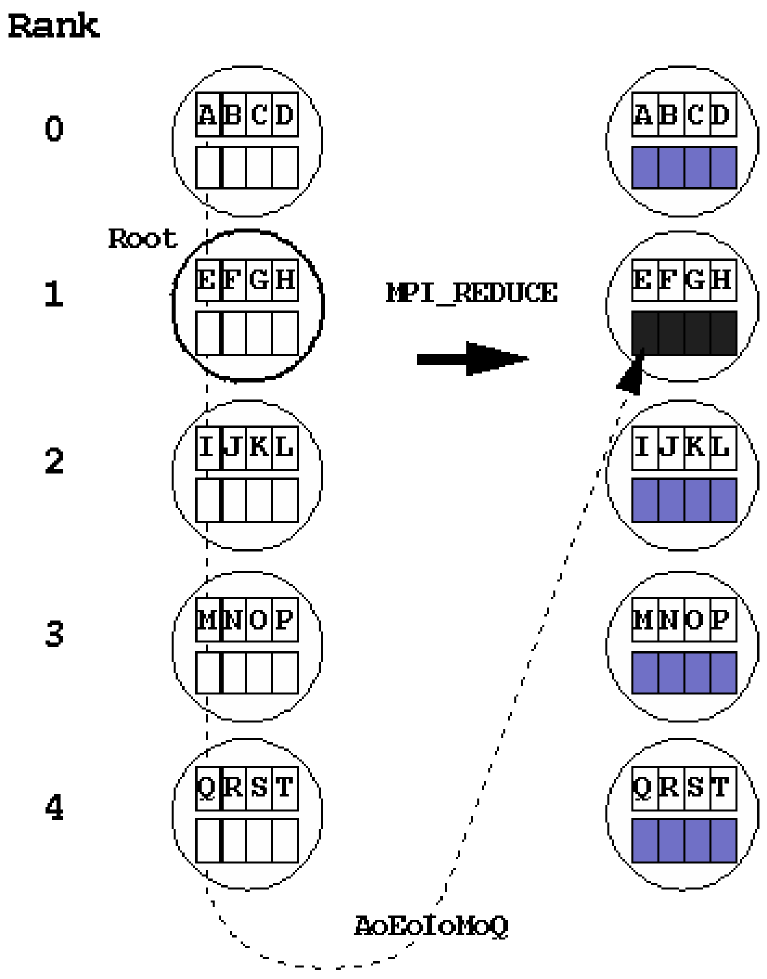
C:

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, int root,
             MPI_Comm comm)
```

Fortran:

```
CALL MPI_REDUCE(SENDBUF,RECVBUF,COUNT,DATATYPE,OP,ROOT,COMM,IERROR)
<type> SENDBUF(*), RECVBUF(*)
```

MPI_Reduce


Rank
0
1
2
3
4
Root
A B C D
E F G H
I J K L
M N O P
Q R S T
MPI_REDUCE
A B C D
E F G H
I J K L
M N O P
Q R S T
AoEoIoMoQ

Minloc and Maxloc

- Designed to compute a global minimum/maximum and and index associated with the extreme value
  - Common application: index is the processor rank (see sample program)
- If more than one extreme, get the first
- Designed to work on operands that consist of a value and index pair
- `MPI_Datatypes` include:

C:

```
MPI_FLOAT_INT, MPI_DOUBLE_INT, MPI_LONG_INT, MPI_2INT,
  MPI_SHORT_INT, MPI_LONG_DOUBLE_INT
```

Fortran:

```
MPI_2REAL, MPI_2DOUBLEPRECISION, MPI_2INTEGER
```

Sample Program #7 - Fortran

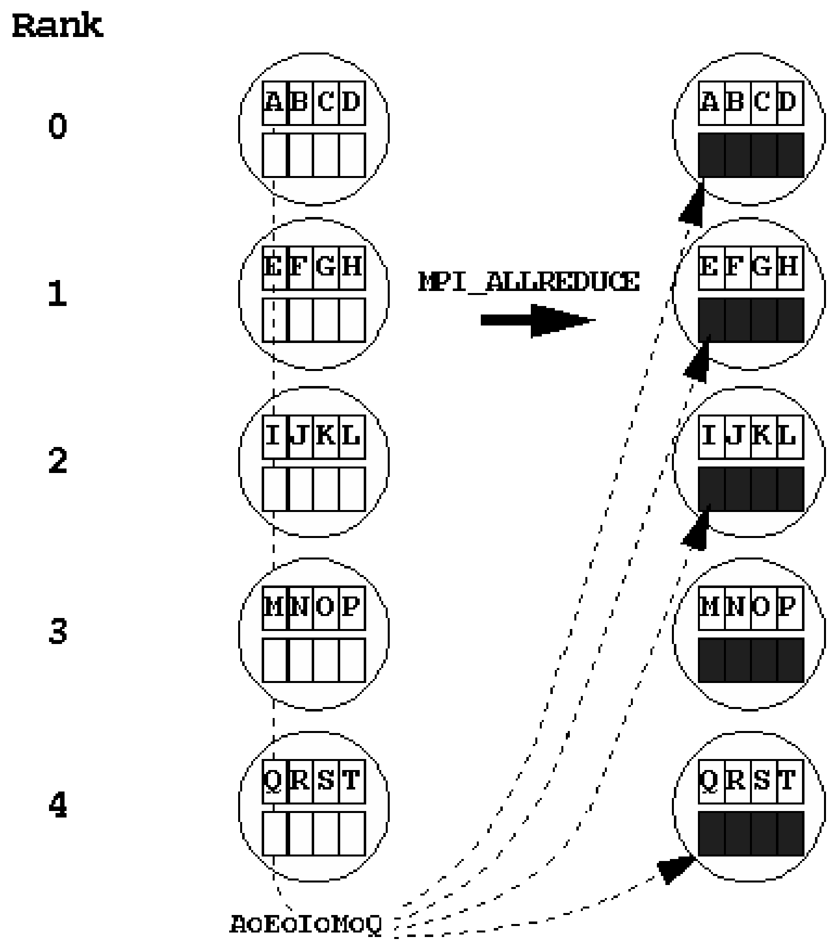
```
      PROGRAM MaxMin
C
C Run with 8 processes
C
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      integer in(2),out(2)
      CALL MPI_INIT(err)
      CALL MPI_COMM_RANK(MPI_WORLD_COMM,rank,err)
      CALL MPI_COMM_SIZE(MPI_WORLD_COMM,size,err)
      in(1)=rank+1
      in(2)=rank
      call MPI_REDUCE(in,out,1,MPI_2INTEGER,MPI_MAXLOC,
     &                7,MPI_COMM_WORLD,err)

      if(rank.eq.7) print *,"P:",rank," max=",out(1)," at rank ",out(2)

      call MPI_REDUCE(in,out,1,MPI_2INTEGER,MPI_MINLOC,
     &                2,MPI_COMM_WORLD,err)

      if(rank.eq.2) print *,"P:",rank," min=",out(1)," at rank ",out(2)

      CALL MPI_FINALIZE(err)
      END
```

```
    Program Output
P:2 min=1 at rank 0
P:7 max=8 at rank 7
```

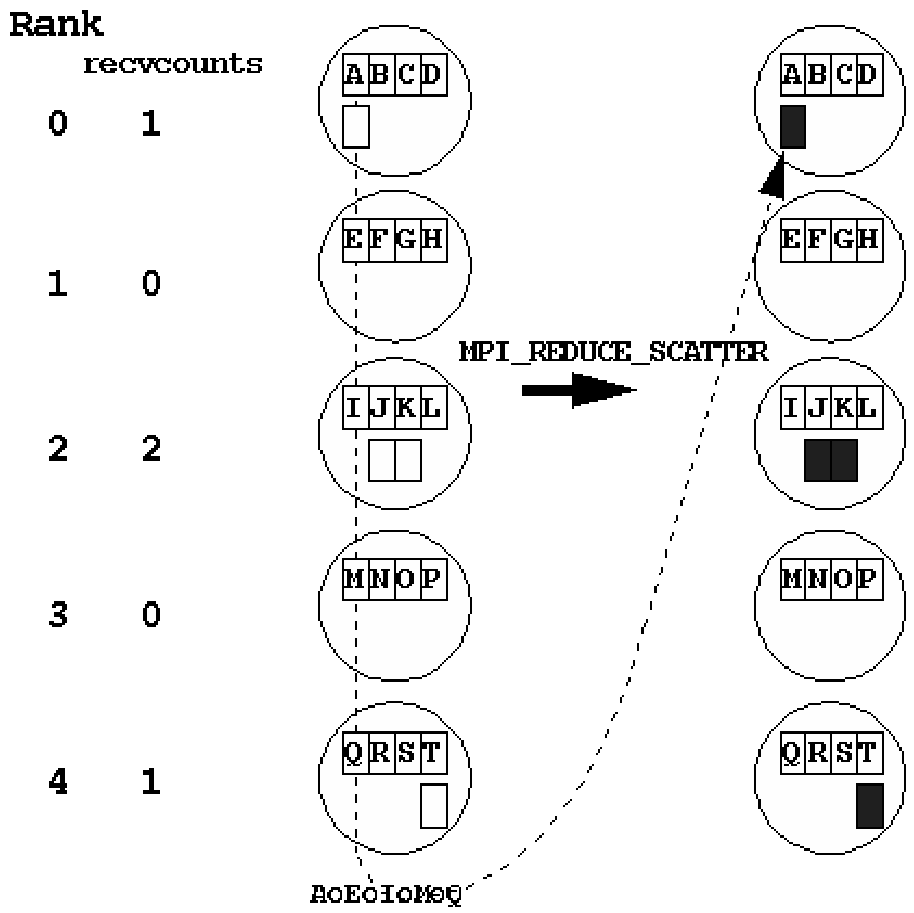
Variants of MPI_REDUCE

- `MPI_ALLREDUCE` -- no root process (all get results)
- `MPI_REDUCE_SCATTER` -- multiple results are scattered
- `MPI_SCAN` -- “parallel prefix”

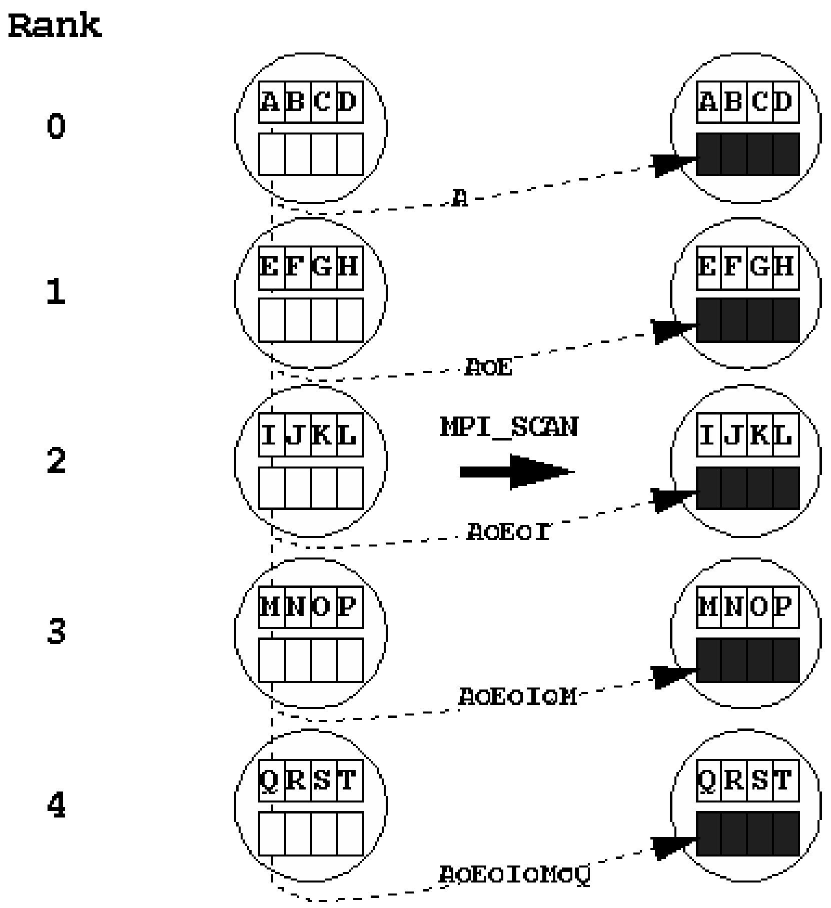
MPI_ALLREDUCE



MPI_REDUCE_SCATTER

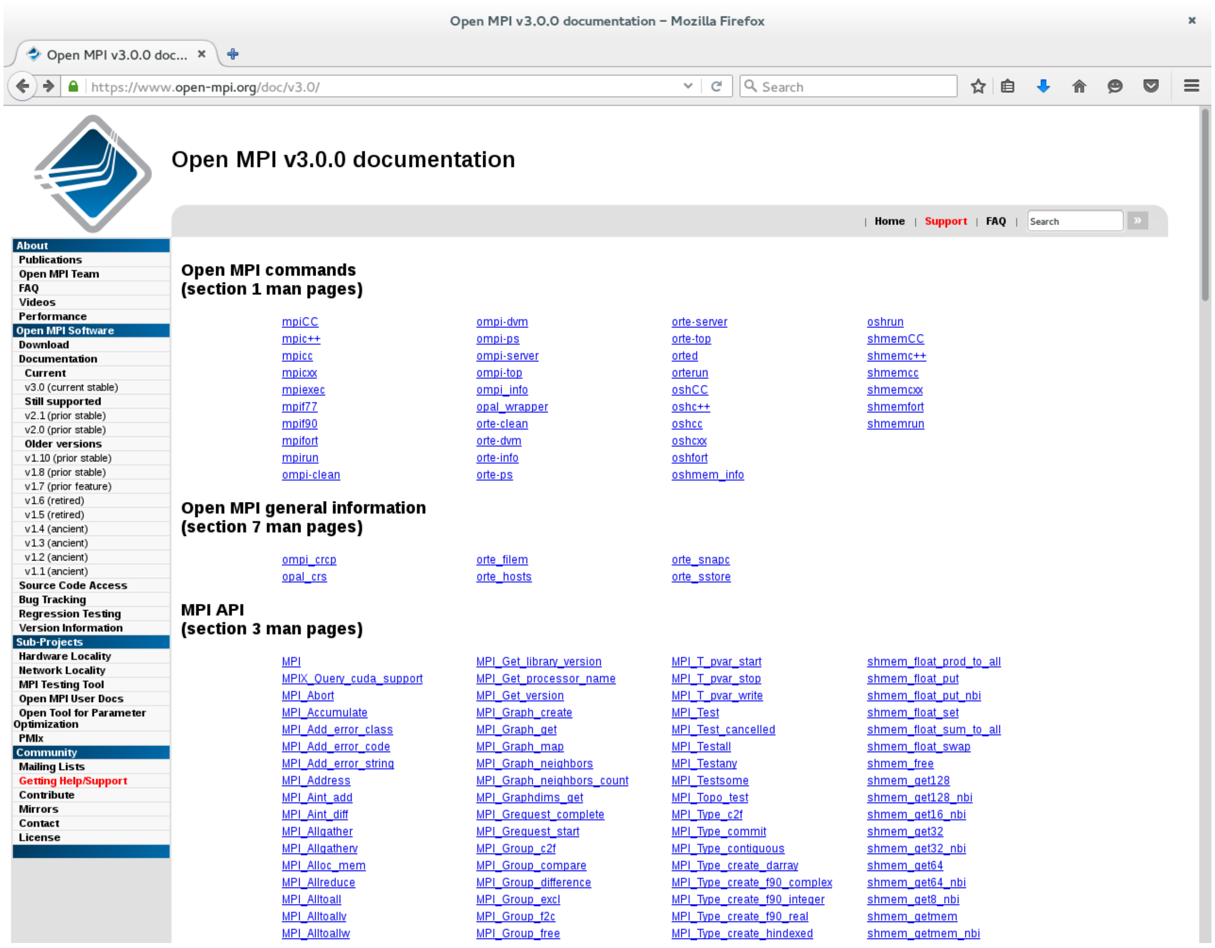


MPI_SCAN



Rank
0
1
2
3
4
A B C D
E F G H
I J K L
M N O P
Q R S T
MPI_SCAN
A
AoE
AoEoI
AoEoIoM
AoEoIoMoQ

MPI has MANY other routines. You can check all of them, together with their syntax and some usage examples at the OpenMPI webpage `https://www.open-mpi.org/doc/v4.0/`


Open MPI v3.0.0 documentation – Mozilla Firefox
Open MPI v3.0.0 doc...
https://www.open-mpi.org/doc/v3.0/
Search
Home | Support | FAQ | Search

Open MPI v3.0.0 documentation

About
- Publications
- Open MPI Team
- FAQ
- Videos
- Performance

Open MPI Software
- Download
- Documentation
  - Current
  - v3.0 (current stable)
  - Still supported
  - v2.1 (prior stable)
  - v2.0 (prior stable)
  - Older versions
  - v1.10 (prior stable)
  - v1.8 (prior stable)
  - v1.7 (prior feature)
  - v1.6 (retired)
  - v1.5 (retired)
  - v1.4 (ancient)
  - v1.3 (ancient)
  - v1.2 (ancient)
  - v1.1 (ancient)
- Source Code Access
- Bug Tracking
- Regression Testing
- Version Information

Sub-Projects
- Hardware Locality
- Network Locality
- MPI Testing Tool
- Open MPI User Docs
- Open Tool for Parameter Optimization
- PMIx

Community
- Mailing Lists
- Getting Help/Support
- Contribute
- Mirrors
- Contact
- License

Open MPI commands (section 1 man pages)

mpiCC	ompi-dvm	orte-server	oshrun
mpic++	ompi-ps	orte-top	shmemCC
mpicc	ompi-server	orted	shmemc++
mpicxx	ompi-top	orterun	shmemcc
mpiexec	ompi_info	oshCC	shmemcxx
mpif77	opal_wrapper	oshc++	shmemfort
mpif90	orte-clean	oshcc	shmemrun
mpifort	orte-dvm	oshcxx	
mpirun	orte-info	oshfort	
ompi-clean	orte-ps	oshmem_info	

Open MPI general information (section 7 man pages)

ompi_crcp	orte_filem	orte_snapc
opal_crs	orte_hosts	orte_sstore

MPI API (section 3 man pages)

MPI	MPI_Get_library_version	MPI_T_pvar_start	shmem_float_prod_to_all
MPIX_Query_cuda_support	MPI_Get_processor_name	MPI_T_pvar_stop	shmem_float_put
MPI_Abort	MPI_Get_version	MPI_T_pvar_write	shmem_float_put_nbi
MPI_Accumulate	MPI_Graph_create	MPI_Test	shmem_float_set
MPI_Add_error_class	MPI_Graph_get	MPI_Test_cancelled	shmem_float_sum_to_all
MPI_Add_error_code	MPI_Graph_map	MPI_Testall	shmem_float_swap
MPI_Add_error_string	MPI_Graph_neighbors	MPI_Testany	shmem_free
MPI_Address	MPI_Graph_neighbors_count	MPI_Testsome	shmem_get128
MPI_Aint_add	MPI_Graphdims_get	MPI_Topo_test	shmem_get128_nbi
MPI_Aint_diff	MPI_Grequest_complete	MPI_Type_c2f	shmem_get16_nbi
MPI_Allgather	MPI_Grequest_start	MPI_Type_commit	shmem_get32
MPI_Allgatherv	MPI_Group_c2f	MPI_Type_contiguous	shmem_get32_nbi
MPI_Alloc_mem	MPI_Group_compare	MPI_Type_create_darray	shmem_get64
MPI_Allreduce	MPI_Group_difference	MPI_Type_create_f90_complex	shmem_get64_nbi
MPI_Alltoall	MPI_Group_excl	MPI_Type_create_f90_integer	shmem_get8_nbi
MPI_Alltoallv	MPI_Group_f2c	MPI_Type_create_f90_real	shmem_getmem
MPI_Alltoallw	MPI_Group_free	MPI_Type_create_hindexed	shmem_getmem_nbi

Figure 7.1: OpenMPI v3.0 documentation page: `https://www.open-mpi.org/doc/v3.0/`

7.2 MPI collective communication exercises

Below there are some exercises to familiarize yourself with the collective communication features of MPI, as seen in section 7.1.

You can find solutions to these exercises in section A.4, but you are *highly encouraged* to try to solve them first on your own.

7.2.1 Broadcast

This exercise was taken from the ”Training MPI” course by CINECA-SCAI. For details see `http://www.hpc.cineca.it/content/exercise-7`

Task 0 initializes a variable to a given value,then modifies the variable (for example, by calculating the square of its value) and finally broadcasts it to all the others tasks.

7.2.2 Trapezoidal rule integral with I/O

Modify the program for the calculation of an integral using the trapezodial rule (see 6.1.2), so that a,b and n are read by rank 0 and they are distributed (with collective calls) to the other processes.

7.2.3 Divide and update array

This exercise was taken from the ”Training MPI” course by CINECA-SCAI. For details see `http://www.hpc.cineca.it/content/exercise-8`

Task 0 initializes a one-dimensional array assigning to each cell the value of its index. This array is then divided into chunks and sent to other processes. After having received the proper chunk, each process updates it by adding its rank and then sends it back to root process. (Analyze the cases for equal and not equal chunks separately).

Start by assuming that the length of the array is divisible by the number of processes. To make it more general, instead of using the routines MPI_GATHER y MPI_SCATTER you should check the documentation for these two routines: MPI_GATHERV (https://www.open-mpi.org/doc/v3.0/man3/MPI_Gatherv.3.php) and MPI_SCATTER_V (https://www.open-mpi.org/doc/v3.0/man3/MPI_Scatterv.3.php).

7.2.4 Reduce operations

Ejercicio 9 del curso ”Training MPI” de CINECA-SCAI. Ver detalles en `http://www.hpc.cineca.it/content/exercise-9`

Each process initializes a one-dimensional array by giving to all the elements the value of its rank+1. Then the root process (task 0) performs two reduce operations (sum and then product) to the arrays of all the processes. Finally, each process generates a random number and root process finds (and prints) the maximum value among these random values.

Modify (optional) the code to perform a simple scalability test using MPI_Wtime. Notice what happens when you go up with the number of processes involved.

In Fortran, in order to generate a “random” number, you can do:

```
REAL :: x
CALL RANDOM_SEED()
CALL RANDOM_NUMBER(x)
```

7.2.5 Heat equation - 2D

Parallelize the following serial code, which follows the pattern that you would have to use in order to solve the 2D equation.

```
PROGRAM heat2D
  IMPLICIT NONE

  INTEGER :: side, i, j, t=0
  REAL :: min,max,diff
  REAL, DIMENSION( : , : ), ALLOCATABLE :: datu

  READ*, side
  ALLOCATE(datu(0:side+1,0:side+1))
```

```
    datu = 0.0
    DO i=1,side
       READ*, datu(i,1:side)
    END DO

    min=MINVAL(datu(1:side,1:side))
    max=MAXVAL(datu(1:side,1:side))
    diff=max-min
    PRINT*, "t = ",t, "diff = ",diff

    DO WHILE (diff .GE. 1)
       t=t+1
       datu(1:side,1:side) = 0.99*datu(1:side,1:side) + 0.01*((datu(0:side-1,1:side) + &
            datu(2:side+1,1:side) + datu(1:side,0:side-1) + datu(1:side,2:side+1)) / 4)
       min=MINVAL(datu(1:side,1:side))
       max=MAXVAL(datu(1:side,1:side))
       diff=max-min
       IF (MOD(t,1000) .EQ. 0) PRINT*, "t = ",t, "diff = ",diff
    END DO
    PRINT*, "Final t is: ", t

  END PROGRAM heat2D
```

This code simply reads an array (let's imagine it represents temperature), which is updated in each loop. For each of these loops (representing a time step), the array is updated at each point with this simple expression:

```
ti,j = 0.99*ti,j + 0.01*((ti-1,j+ti+1,j+ti,j-1+ti,j+1)/4)
```

This is computed while the difference between the minimum and maximum values of the array is >=1. We also print this difference for every 1000 time steps, and at the end we print the total number of time steps given. To verify that your code is working, you can use the following array (first line gives the array side, so in this case we are reading a 10x10 array):

```
10
1 5 5 5 5 5 5 5 5 5
5 1 5 5 5 5 5 5 5 5
5 5 1 5 5 5 5 5 5 5
5 5 5 1 5 5 5 5 5 5
5 5 5 5 1 5 5 5 5 5
5 5 5 5 5 1 5 5 5 5
5 5 5 5 5 5 1 5 5 5
5 5 5 5 5 5 5 1 5 5
5 5 5 5 5 5 5 5 1 5
5 5 5 5 5 5 5 5 5 1
```

With this array, the serial program gives:

```
angelv$ ./heat2D < temp.in
 t =            0 diff =    4.00000000
 t =         1000 diff =    3.61902761
 t =         2000 diff =    2.80383563
 t =         3000 diff =    1.92610180
 t =         4000 diff =    1.29235637
 Final t is:         4636
```

Your goal is to parallelize this code. Some comments that can help you with this task:

- To simplify, we can assume that your code will always run in 4 processes and that the array side is multiple of 2 (so, when dividing the array in 4 chunks, each chunk has the same number of elements). But you have to make sure that the code works for any array size (as far as the side is multiple of 2).

- Besides the basic initialization and finalization MPI calls, you can solve this problem using only: MPI_BCAST, MPI_ALLREDUCE, MPI_Send and MPI_Recv
- Don't try to modify all the code in one go (parallel programs are much more difficult to debug than serial ones. Try to change the code in a number of steps and making sure that each step gives the correct results:
  - rank 0 just reads the array and distributes it to the other processes
  - now write the part of the program before the loop, where you will determine which chunk of the array each process will be responsible for, and calculate the initial difference between minimum and maximum values.
  - now go for the loop. The first thing to do (and probably the most complicated part of the program) is to determine which information you will have to communicate from process to process. Only after you have a clear idea of who sends what, then try to write the right MPI routines to perform that communication.

Chapter 8

Barnes-Hut code in parallel

In order to implement the Barnes-Hut algorithm in parallel, we can have a number of options. Let's explore two of them.

1. The first solution we could implement is similar to the parallel version not using Barnes-Hut. We can assign to each process a number of bodies for which it will be responsible. The tree that we build with the Barnes-Hut algorithm can be built by all the processes (or perhaps we could do that only one builds it and then sends it to the other processes, but that would be more inefficient), and then each process calculates how all the bodies affect the particles it is responsible for. This result can then be sent to the other processes, so then all processes will have the new updated positions for all bodies and then can continue with the create tree ; calculate forces loop.
2. The second solution is more efficient, since we can parallelize both the forces calculation and the building of the tree. The initial box is divided in 8 octants (and then later recursively each octant in another 8, etc.), so we can implement this solution for 8 processes, where each process will take care of one octant. Thus, each process will build a partial tree, for the bodies that belong to its octant. When the tree is built, we can calculate how the bodies in my tree affect all particles. Then, by using the function MPI_ALLREDUCE we can add up the accelerations for all particles produced by the 8 different trees belonging to each of the octants. This solution is slightly more difficult than the first one (and meant for a fixed number of processes), but we perform in parallel both the forces calculation and the tree building, so its performance is much better than solution 1.

8.1 Solution 1

barnes-hut.parallel.solution1.f90

```
PROGRAM tree
  USE MPI
  IMPLICIT NONE

 !! Variables MPI (rango, numero procesadores, etc.
 !!                y para determinar mi bloque de trabajo)
 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  INTEGER :: my_rank, p, error
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
  INTEGER :: my_n, my_start, my_end

!! Variables del problema Nbody
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  INTEGER :: i,j,k,n
  REAL :: dt, t_end, t, dt_out, t_out, rs, r2, r3
  REAL, PARAMETER :: theta = 1
  REAL, DIMENSION(:), ALLOCATABLE :: m
  REAL, DIMENSION(:,:), ALLOCATABLE :: r,v,a
  REAL, DIMENSION(3) :: rji

  TYPE RANGE
     REAL, DIMENSION(3) :: min,max
  END TYPE RANGE

  TYPE CPtr
     TYPE(CELL), POINTER :: ptr
  END TYPE CPtr

  TYPE CELL
     TYPE (RANGE) :: range
     REAL, DIMENSION(3) :: part
     INTEGER :: pos
     INTEGER :: type !! 0 = no particle; 1 = particle; 2 = conglomerado
     REAL :: mass
     REAL, DIMENSION(3) :: c_o_m
     TYPE (CPtr), DIMENSION(2,2,2) :: subcell
  END TYPE CELL

  TYPE (CELL), POINTER :: head, temp_cell

!! Inicialización de MPI
!!!!!!!!!!!!!!!!!!!!!!!!
  CALL MPI_INIT ( error )
  CALL MPI_COMM_SIZE ( MPI_COMM_WORLD, p, error )
  CALL MPI_COMM_RANK ( MPI_COMM_WORLD, my_rank, error )


!! Inicialización de matrices
!! El master lee de fichero y hace un broadcast de
!! todas las variables a todo el resto de slaves
!!
  IF ( my_rank == 0 ) THEN

     READ*, dt
     READ*, dt_out
     READ*, t_end
     READ*, n

     CALL MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(dt,1,MPI_REAL,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(dt_out,1,MPI_REAL,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(t_end,1,MPI_REAL,0,MPI_COMM_WORLD,error)

     ALLOCATE(m(n))
     ALLOCATE(r(n,3))
     ALLOCATE(v(n,3))
     ALLOCATE(a(n,3))

     DO i = 1, n
        READ*, m(i), r(i,:),v(i,:)
     END DO

     CALL MPI_BCAST(m,n,MPI_REAL,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(r,n*3,MPI_REAL,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(v,n*3,MPI_REAL,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(a,n*3,MPI_REAL,0,MPI_COMM_WORLD,error)
  ELSE

     CALL MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(dt,1,MPI_REAL,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(dt_out,1,MPI_REAL,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(t_end,1,MPI_REAL,0,MPI_COMM_WORLD,error)

     ALLOCATE(m(n))
     ALLOCATE(r(n,3))
     ALLOCATE(v(n,3))
     ALLOCATE(a(n,3))

     CALL MPI_BCAST(m,n,MPI_REAL,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(r,n*3,MPI_REAL,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(v,n*3,MPI_REAL,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(a,n*3,MPI_REAL,0,MPI_COMM_WORLD,error)

  END IF

!! Calculamos el bloque que me toca calcular.
!! A partir de aquí ya no hay distinción master / slave excepto para imprimir.
!! Por lo demás todos los procesadores colaboran por igual.
!! Se asume que n es divisible por p, y calculamos el comienzo de mi
!! bloque (my_start) y el final de mi bloque (my_end)
!!
  my_n = n / p
  my_start = (my_n * my_rank) + 1
  my_end = my_start + my_n - 1


!! Inicialización head node
!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ALLOCATE(head)

  CALL Calculate_ranges(head)
  head%type = 0
  CALL Nullify_Pointers(head)


!! Creación del árbol inicial
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  DO i = 1,n
     CALL Find_Cell(head,temp_cell,r(i,:))
     CALL Place_Cell(temp_cell,r(i,:),i)
  END DO

  CALL Borrar_empty_leaves(head)
  CALL Calculate_masses(head)

!! Calcular aceleraciones iniciales
!! Ésto es similar a la versión serie, pero ahora las operaciones
!!sobre matrices ahora las hacemos con un rango.
!!    Por ejemplo: a(my_start:my_end,:)
!!    Es decir, sólo me encargo de trabajar sobre mi bloque.
!!    Los otros procesadores hacen lo propio con su bloque.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  a(my_start:my_end,:) = 0.0
  CALL Calculate_forces(head,my_start,my_end)

!! Bucle principal
!!!!!!!!!!!!!!!!!!
  t_out = 0.0
  DO t = 0.0, t_end, dt
     v(my_start:my_end,:) = v(my_start:my_end,:) + a(my_start:my_end,:) * dt/2
     r(my_start:my_end,:) = r(my_start:my_end,:) + v(my_start:my_end,:) * dt
     CALL MPI_ALLGATHER(r(my_start,1),my_n,MPI_REAL,r(1,1),my_n,MPI_REAL,MPI_COMM_WORLD,error)
     CALL MPI_ALLGATHER(r(my_start,2),my_n,MPI_REAL,r(1,2),my_n,MPI_REAL,MPI_COMM_WORLD,error)
     CALL MPI_ALLGATHER(r(my_start,3),my_n,MPI_REAL,r(1,3),my_n,MPI_REAL,MPI_COMM_WORLD,error)

     !! Las posiciones han cambiado, por lo que tenemos que borrar
     !! y reinicializar el arbol
     CALL Borrar_tree(head)

     CALL Calculate_ranges(head)
     head%type = 0
     CALL Nullify_Pointers(head)

     DO i = 1,n
        CALL Find_Cell(head,temp_cell,r(i,:))
        CALL Place_Cell(temp_cell,r(i,:),i)
     END DO

     CALL Borrar_empty_leaves(head)
     CALL Calculate_masses(head)

     a(my_start:my_end,:) = 0.0
     CALL Calculate_forces(head,my_start,my_end)
     v(my_start:my_end,:) = v(my_start:my_end,:) + a(my_start:my_end,:) * dt/2

     !! Sólo imprimimos si somos el master
     !!
     IF (my_rank == 0) THEN
        t_out = t_out + dt
        IF (t_out >= dt_out) THEN
           DO i = 1,10
              PRINT*, r(i,:)
           END DO
           PRINT*, "-----------------------------------"
           PRINT*, ""
           t_out = 0.0
        END IF
     END IF

  END DO

  CALL MPI_Finalize ( error )

CONTAINS

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calculate_Ranges                       !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE Calculate_Ranges(goal)
    TYPE(CELL),POINTER :: goal

    REAL, DIMENSION(3) :: mins,maxs,medios
    REAL :: span

    mins = MINVAL(r,DIM=1)
    maxs = MAXVAL(r,DIM=1)
    span = MAXVAL(maxs - mins) * 1.1 ! Le sumo un 10% para que las particulas no caigan justo en el bo
    medios = (maxs + mins) / 2.0
    goal%range%min = medios - span/2.0
    goal%range%max = medios + span/2.0
  END SUBROUTINE Calculate_Ranges

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Find_Cell                              !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
  RECURSIVE SUBROUTINE Find_Cell(root,goal,part)
    REAL, DIMENSION(3) :: part
    TYPE(CELL),POINTER :: root,goal,temp
    INTEGER :: i,j,k

    SELECT CASE (root%type)
       CASE (2)
          out: DO i = 1,2
             DO j = 1,2
                DO k = 1,2
                   IF (Belongs(part,root%subcell(i,j,k)%ptr)) THEN
                      CALL Find_Cell(root%subcell(i,j,k)%ptr,temp,part)
                      goal => temp
                      EXIT out
                   END IF
                END DO
             END DO
          END DO out
       CASE DEFAULT
          goal => root
     END SELECT
  END SUBROUTINE Find_Cell

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Place_Cell                             !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  RECURSIVE SUBROUTINE Place_Cell(goal,part,n)
    TYPE(CELL),POINTER :: goal,temp
    REAL, DIMENSION(3) :: part
    INTEGER :: n

    SELECT CASE (goal%type)
       CASE (0)
          goal%type = 1
          goal%part = part
          goal%pos = n
       CASE (1)
          CALL Crear_Subcells(goal)
          CALL Find_Cell(goal,temp,part)
          CALL Place_Cell(temp,part,n)
       CASE DEFAULT
          print*,"SHOULD NOT BE HERE. ERROR!"
    END SELECT
  END SUBROUTINE Place_Cell

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Crear_Subcells                         !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE Crear_Subcells(goal)
    TYPE(CELL), POINTER :: goal
    REAL,DIMENSION(3) :: part
    INTEGER :: i,j,k,n
    INTEGER, DIMENSION(3) :: octant

    part = goal%part
    goal%type=2

    DO i = 1,2
       DO j = 1,2
          DO k = 1,2
             octant = (/i,j,k/)
             ALLOCATE(goal%subcell(i,j,k)%ptr)
             goal%subcell(i,j,k)%ptr%range%min = Calcular_Range (0,goal,octant)
             goal%subcell(i,j,k)%ptr%range%max = Calcular_Range (1,goal,octant)
```

```
              IF (Belongs(part,goal%subcell(i,j,k)%ptr)) THEN
                 goal%subcell(i,j,k)%ptr%part = part
                 goal%subcell(i,j,k)%ptr%type = 1
                 goal%subcell(i,j,k)%ptr%pos = goal%pos
              ELSE
                 goal%subcell(i,j,k)%ptr%type = 0
              END IF
              CALL Nullify_Pointers(goal%subcell(i,j,k)%ptr)
           END DO
        END DO
     END DO
  END SUBROUTINE Crear_Subcells


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Nullify_Pointers                       !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE Nullify_Pointers(goal)
    TYPE(CELL), POINTER :: goal
    INTEGER :: i,j,k

    DO i = 1,2
       DO j = 1,2
          DO k = 1,2
             NULLIFY(goal%subcell(i,j,k)%ptr)
          END DO
       END DO
    END DO
  END SUBROUTINE Nullify_Pointers


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Belongs                                !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  FUNCTION Belongs (part,goal)
    REAL, DIMENSION(3) :: part
    TYPE(CELL), POINTER :: goal
    LOGICAL :: Belongs

    IF (part(1) >= goal%range%min(1) .AND. &
         part(1) <= goal%range%max(1) .AND. &
         part(2) >= goal%range%min(2) .AND. &
         part(2) <= goal%range%max(2) .AND. &
         part(3) >= goal%range%min(3) .AND. &
         part(3) <= goal%range%max(3)) THEN
       Belongs = .TRUE.
    ELSE
       Belongs = .FALSE.
    END IF
  END FUNCTION Belongs

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calcular_Range                         !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  FUNCTION Calcular_Range (what,goal,octant)
    INTEGER :: what,n
    TYPE(CELL), POINTER :: goal
    INTEGER, DIMENSION(3) :: octant
    REAL, DIMENSION(3) :: Calcular_Range, valor_medio

    valor_medio = (goal%range%min + goal%range%max) / 2.0

    SELECT CASE (what)
    CASE (0)
       WHERE (octant == 1)
```

```
            Calcular_Range = goal%range%min
         ELSEWHERE
            Calcular_Range = valor_medio
         ENDWHERE
      CASE (1)
         WHERE (octant == 1)
            Calcular_Range = valor_medio
         ELSEWHERE
            Calcular_Range = goal%range%max
         ENDWHERE
      END SELECT
    END FUNCTION Calcular_Range

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !! Borrar_empty_leaves                     !!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    RECURSIVE SUBROUTINE Borrar_empty_leaves(goal)
      TYPE(CELL),POINTER :: goal
      INTEGER :: i,j,k

      IF (ASSOCIATED(goal%subcell(1,1,1)%ptr)) THEN
         DO i = 1,2
            DO j = 1,2
               DO k = 1,2
                  CALL Borrar_empty_leaves(goal%subcell(i,j,k)%ptr)
                  IF (goal%subcell(i,j,k)%ptr%type == 0) THEN
                     DEALLOCATE (goal%subcell(i,j,k)%ptr)
                  END IF
               END DO
            END DO
         END DO
      END IF
    END SUBROUTINE Borrar_empty_leaves

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !! Borrar_tree                             !!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    RECURSIVE SUBROUTINE Borrar_tree(goal)
      TYPE(CELL),POINTER :: goal
      INTEGER :: i,j,k

         DO i = 1,2
            DO j = 1,2
               DO k = 1,2
                  IF (ASSOCIATED(goal%subcell(i,j,k)%ptr)) THEN
                     CALL Borrar_tree(goal%subcell(i,j,k)%ptr)
                     DEALLOCATE (goal%subcell(i,j,k)%ptr)
                  END IF
               END DO
            END DO
         END DO
    END SUBROUTINE Borrar_tree

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !! Calculate_masses                        !!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    RECURSIVE SUBROUTINE Calculate_masses(goal)
      TYPE(CELL),POINTER :: goal
      INTEGER :: i,j,k
      REAL :: mass
      REAL, DIMENSION(3) :: c_o_m

      goal%mass = 0
      goal%c_o_m = 0
      SELECT CASE (goal%type)
         CASE (1)
            goal%mass = m(goal%pos)
            goal%c_o_m = r(goal%pos,:)
         CASE (2)
            DO i = 1,2
               DO j = 1,2
                  DO k = 1,2
                     IF (ASSOCIATED(goal%subcell(i,j,k)%ptr)) THEN
                        CALL Calculate_masses(goal%subcell(i,j,k)%ptr)
                        mass = goal%mass
                        goal%mass = goal%mass + goal%subcell(i,j,k)%ptr%mass
                        goal%c_o_m = (mass * goal%c_o_m + goal%subcell(i,j,k)%ptr%mass * goal%subcell(i,
                     END IF
                  END DO
               END DO
            END DO
      END SELECT
    END SUBROUTINE Calculate_masses


  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !! Calculate_forces                       !!
  !!                                        !!
  !! Muy parecida a la version en serie,    !!
  !! simplemente le paso el comienzo y el   !!
  !! final de mi bloque y solo llamare a la !!
  !! funcion aux en caso de que la particula!!
  !! a tratar sea una de ellas              !!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    RECURSIVE SUBROUTINE Calculate_forces(head,start,end)
      TYPE(CELL),POINTER :: head
      INTEGER :: i,j,k,start,end

      DO i= start,end
         CALL Calculate_forces_aux(i,head)
      END DO

    END SUBROUTINE Calculate_forces


  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !! Calculate_forces_aux                   !!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    RECURSIVE SUBROUTINE Calculate_forces_aux(goal,tree)
      TYPE(CELL),POINTER :: tree
      INTEGER :: i,j,k,goal
      REAL :: l,D

      SELECT CASE (tree%type)
         CASE (1)
            IF (goal .NE. tree%pos) THEN
               rji = tree%c_o_m - r(goal,:)
               r2 = SUM(rji**2)
               r3 = r2 * SQRT(r2)
               a(goal,:) = a(goal,:) + m(tree%pos) * rji / r3
            END IF
         CASE (2)
            l = tree%range%max(1) - tree%range%min(1) !! El rango tiene el mismo span en las 3 dimensior
            rji = tree%c_o_m - r(goal,:)
            r2 = SUM(rji**2)
            D = SQRT(r2)
            IF (l/D < theta) THEN
               !! Si conglomerado, tenemos que ver si se cumple l/D < @
               r3 = r2 * D
```

```
            a(goal,:) = a(goal,:) + tree%mass * rji / r3
         ELSE
            DO i = 1,2
               DO j = 1,2
                  DO k = 1,2
                     IF (ASSOCIATED(tree%subcell(i,j,k)%ptr)) THEN
                        CALL Calculate_forces_aux(goal,tree%subcell(i,j,k)%ptr)
                     END IF
                  END DO
               END DO
            END DO
         END IF
      END SELECT

    END SUBROUTINE Calculate_forces_aux

  END PROGRAM tree
```

8.2 Solution 2

barnes-hut.parallel.solution2.f90

```
PROGRAM tree
  IMPLICIT NONE

!! Usamos el "include" en lugar del "USE", pues con el "USE" no podiamos utilizar MPI_ALLREDUCE
!! USE MPI
  include 'mpif.h'

 !! Variables MPI (rango, numero procesadores, etc.
 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  INTEGER :: my_rank, p, error
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

!! Variables del problema Nbody
!!
!! Una variable nueva con respecto a la Solución 1 es
!! total_a. En el array "a" vamos a calcular como las
!! partículas de nuestro árbol afectan a todas las partículas.
!! Con un MPI_ALLREDUCE sumaremos todas las "a"s locales en
!! "total_a", con lo que calcularemos la aceleración total para
!! todas la partículas (afectadas por los árboles de los 8 octantes)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  INTEGER :: i,j,k,n
  REAL :: dt, t_end, t, dt_out, t_out, rs, r2, r3
  REAL, PARAMETER :: theta = 1
  REAL, DIMENSION(:), ALLOCATABLE :: m
  REAL, DIMENSION(:,:), ALLOCATABLE :: r,v,a,total_a
  REAL, DIMENSION(3) :: rji

  TYPE RANGE
     REAL, DIMENSION(3) :: min,max
  END TYPE RANGE

  TYPE CPtr
     TYPE(CELL), POINTER :: ptr
  END TYPE CPtr

  TYPE CELL
     TYPE (RANGE) :: range
     REAL, DIMENSION(3) :: part
     INTEGER :: pos
```

```
         INTEGER :: type !! 0 = no particle; 1 = particle; 2 = conglomerado
         REAL :: mass
         REAL, DIMENSION(3) :: c_o_m
         TYPE (CPtr), DIMENSION(2,2,2) :: subcell
      END TYPE CELL

      TYPE (CELL), POINTER :: head, temp_cell

    !! Inicialización de MPI
    !!!!!!!!!!!!!!!!!!!!!!!!
      CALL MPI_INIT ( error )
      CALL MPI_COMM_SIZE ( MPI_COMM_WORLD, p, error )
      CALL MPI_COMM_RANK ( MPI_COMM_WORLD, my_rank, error )


    !! Inicialización de matrices
    !! El master lee de fichero y hace un broadcast de
    !! todas las variables a todo el resto de slaves
    !!
      IF ( my_rank == 0 ) THEN

         READ*, dt
         READ*, dt_out
         READ*, t_end
         READ*, n

         CALL MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,error)
         CALL MPI_BCAST(dt,1,MPI_REAL,0,MPI_COMM_WORLD,error)
         CALL MPI_BCAST(dt_out,1,MPI_REAL,0,MPI_COMM_WORLD,error)
         CALL MPI_BCAST(t_end,1,MPI_REAL,0,MPI_COMM_WORLD,error)

         ALLOCATE(m(n))
         ALLOCATE(r(n,3))
         ALLOCATE(v(n,3))
         ALLOCATE(a(n,3))
         ALLOCATE(total_a(n,3))

         DO i = 1, n
            READ*, m(i), r(i,:),v(i,:)
         END DO

         CALL MPI_BCAST(m,n,MPI_REAL,0,MPI_COMM_WORLD,error)
         CALL MPI_BCAST(r,n*3,MPI_REAL,0,MPI_COMM_WORLD,error)
         CALL MPI_BCAST(v,n*3,MPI_REAL,0,MPI_COMM_WORLD,error)

      ELSE

         CALL MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,error)
         CALL MPI_BCAST(dt,1,MPI_REAL,0,MPI_COMM_WORLD,error)
         CALL MPI_BCAST(dt_out,1,MPI_REAL,0,MPI_COMM_WORLD,error)
         CALL MPI_BCAST(t_end,1,MPI_REAL,0,MPI_COMM_WORLD,error)

         ALLOCATE(m(n))
         ALLOCATE(r(n,3))
         ALLOCATE(v(n,3))
         ALLOCATE(a(n,3))
         ALLOCATE(total_a(n,3))

         CALL MPI_BCAST(m,n,MPI_REAL,0,MPI_COMM_WORLD,error)
         CALL MPI_BCAST(r,n*3,MPI_REAL,0,MPI_COMM_WORLD,error)
         CALL MPI_BCAST(v,n*3,MPI_REAL,0,MPI_COMM_WORLD,error)
      END IF


    !! Inicialización head node
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!
  ALLOCATE(head)

  CALL Calculate_ranges(head)
  head%type = 0
  CALL Nullify_Pointers(head)


!! Creación del árbol inicial
!!
!! Nota: sólo tenemos en cuenta
!! para la creación del árbol las
!! partículas que pertenecen al rango
!! de la cabeza del árbol (head), dado
!! por la subrutina Calculate_ranges
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  DO i = 1,n
     IF (Belongs(r(i,:),head)) THEN
        CALL Find_Cell(head,temp_cell,r(i,:))
        CALL Place_Cell(temp_cell,r(i,:),i)
     END IF
  END DO

  CALL Borrar_empty_leaves(head)
  CALL Calculate_masses(head)

  a = 0.0
  CALL Calculate_forces(head)
  CALL MPI_ALLREDUCE(a,total_a,n*3,MPI_REAL,MPI_SUM,MPI_COMM_WORLD,error)

!! Bucle principal
!!!!!!!!!!!!!!!!!
  t_out = 0.0
  DO t = 0.0, t_end, dt
     v = v + total_a * dt/2
     r = r + v * dt

     !! Las posiciones han cambiado, por lo que tenemos que borrar
     !! y reinicializar el arbol
     CALL Borrar_tree(head)

     CALL Calculate_ranges(head)
     head%type = 0
     CALL Nullify_Pointers(head)

     DO i = 1,n
        IF (Belongs(r(i,:),head)) THEN
           CALL Find_Cell(head,temp_cell,r(i,:))
           CALL Place_Cell(temp_cell,r(i,:),i)
        END IF
     END DO

     CALL Borrar_empty_leaves(head)
     CALL Calculate_masses(head)

     a = 0.0
     CALL Calculate_forces(head)
     CALL MPI_ALLREDUCE(a,total_a,n*3,MPI_REAL,MPI_SUM,MPI_COMM_WORLD,error)

     v = v + total_a * dt/2

     !! Sólo imprimimos si somos el master
     !!
     IF (my_rank == 0) THEN
        t_out = t_out + dt
```

```
          IF (t_out >= dt_out) THEN
             DO i = 1,10
                PRINT*, r(i,:)
             END DO
             PRINT*, "-----------------------------------"
             PRINT*, ""
             t_out = 0.0
          END IF
       END IF

    END DO

    CALL MPI_Finalize ( error )

  CONTAINS

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !! Calculate_Ranges                       !!
  !!                                        !!
  !! Parecida a la subrutina en la solución !!
  !! primera, pero modificamos los rangos   !!
  !! de la cabeza del árbol para que a cada !!
  !! uno de los 8 procesadores les toque    !!
  !! su parte correspondiente.              !!
  !!                                        !!
  !! Utiliza: Convert_rank_octant           !!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    SUBROUTINE Calculate_Ranges(goal)
      TYPE(CELL),POINTER :: goal

      REAL, DIMENSION(3) :: mins,maxs,medios,new_range_min,new_range_max
      REAL :: span
      INTEGER, DIMENSION(3) :: octant

      mins = MINVAL(r,DIM=1)
      maxs = MAXVAL(r,DIM=1)
      span = MAXVAL(maxs - mins) * 1.1 ! Le sumo un 10% para que las particulas no caigan justo en el bo
      medios = (maxs + mins) / 2.0
      goal%range%min = medios - span/2.0
      goal%range%max = medios + span/2.0

      octant = Convert_rank_octant(my_rank)

      new_range_min = Calcular_Range(0,goal,octant)
      new_range_max = Calcular_Range(1,goal,octant)

      goal%range%min = new_range_min
      goal%range%max = new_range_max

    END SUBROUTINE Calculate_Ranges

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !! Convert_rank_octant                    !!
  !!                                        !!
  !! Usada por Calculate_ranges             !!
  !! Simplemente nos hace un "mapping" del  !!
  !! rango a octantes.                      !!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    FUNCTION Convert_rank_octant(rank)
      INTEGER :: rank
      INTEGER, DIMENSION(3) :: Convert_rank_octant

      SELECT CASE (rank)
      CASE (0)
         Convert_rank_octant = (/1,1,1/)
      CASE (1)
         Convert_rank_octant = (/1,1,2/)
      CASE (2)
         Convert_rank_octant = (/1,2,1/)
      CASE (3)
         Convert_rank_octant = (/1,2,2/)
      CASE (4)
         Convert_rank_octant = (/2,1,1/)
      CASE (5)
         Convert_rank_octant = (/2,1,2/)
      CASE (6)
         Convert_rank_octant = (/2,2,1/)
      CASE (7)
         Convert_rank_octant = (/2,2,2/)
      END SELECT
    END FUNCTION Convert_rank_octant


  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !! Find_Cell                              !!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    RECURSIVE SUBROUTINE Find_Cell(root,goal,part)
      REAL, DIMENSION(3) :: part
      TYPE(CELL),POINTER :: root,goal,temp
      INTEGER :: i,j,k

      SELECT CASE (root%type)
         CASE (2)
            out: DO i = 1,2
               DO j = 1,2
                  DO k = 1,2
                     IF (Belongs(part,root%subcell(i,j,k)%ptr)) THEN
                        CALL Find_Cell(root%subcell(i,j,k)%ptr,temp,part)
                        goal => temp
                        EXIT out
                     END IF
                  END DO
               END DO
            END DO out
         CASE DEFAULT
            goal => root
       END SELECT
    END SUBROUTINE Find_Cell

  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !! Place_Cell                             !!
  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    RECURSIVE SUBROUTINE Place_Cell(goal,part,n)
      TYPE(CELL),POINTER :: goal,temp
      REAL, DIMENSION(3) :: part
      INTEGER :: n

      SELECT CASE (goal%type)
         CASE (0)
            goal%type = 1
            goal%part = part
            goal%pos = n
         CASE (1)
            CALL Crear_Subcells(goal)
            CALL Find_Cell(goal,temp,part)
            CALL Place_Cell(temp,part,n)
         CASE DEFAULT
            print*,"SHOULD NOT BE HERE. ERROR!"
      END SELECT
    END SUBROUTINE Place_Cell
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Crear_Subcells                         !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE Crear_Subcells(goal)
    TYPE(CELL), POINTER :: goal
    REAL,DIMENSION(3) :: part
    INTEGER :: i,j,k,n
    INTEGER, DIMENSION(3) :: octant

    part = goal%part
    goal%type=2

    DO i = 1,2
       DO j = 1,2
          DO k = 1,2
             octant = (/i,j,k/)
             ALLOCATE(goal%subcell(i,j,k)%ptr)
             goal%subcell(i,j,k)%ptr%range%min = Calcular_Range (0,goal,octant)
             goal%subcell(i,j,k)%ptr%range%max = Calcular_Range (1,goal,octant)

             IF (Belongs(part,goal%subcell(i,j,k)%ptr)) THEN
                goal%subcell(i,j,k)%ptr%part = part
                goal%subcell(i,j,k)%ptr%type = 1
                goal%subcell(i,j,k)%ptr%pos = goal%pos
             ELSE
                goal%subcell(i,j,k)%ptr%type = 0
             END IF
             CALL Nullify_Pointers(goal%subcell(i,j,k)%ptr)
          END DO
       END DO
    END DO
  END SUBROUTINE Crear_Subcells


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Nullify_Pointers                       !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  SUBROUTINE Nullify_Pointers(goal)
    TYPE(CELL), POINTER :: goal
    INTEGER :: i,j,k

    DO i = 1,2
       DO j = 1,2
          DO k = 1,2
             NULLIFY(goal%subcell(i,j,k)%ptr)
          END DO
       END DO
    END DO
  END SUBROUTINE Nullify_Pointers


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Belongs                                !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  FUNCTION Belongs (part,goal)
    REAL, DIMENSION(3) :: part
    TYPE(CELL), POINTER :: goal
    LOGICAL :: Belongs

    IF (part(1) >= goal%range%min(1) .AND. &
         part(1) <= goal%range%max(1) .AND. &
         part(2) >= goal%range%min(2) .AND. &
         part(2) <= goal%range%max(2) .AND. &
         part(3) >= goal%range%min(3) .AND. &
```

```
        part(3) <= goal%range%max(3)) THEN
      Belongs = .TRUE.
   ELSE
      Belongs = .FALSE.
   END IF
 END FUNCTION Belongs

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calcular_Range                          !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 FUNCTION Calcular_Range (what,goal,octant)
   INTEGER :: what,n
   TYPE(CELL), POINTER :: goal
   INTEGER, DIMENSION(3) :: octant
   REAL, DIMENSION(3) :: Calcular_Range, valor_medio

   valor_medio = (goal%range%min + goal%range%max) / 2.0

   SELECT CASE (what)
   CASE (0)
      WHERE (octant == 1)
         Calcular_Range = goal%range%min
      ELSEWHERE
         Calcular_Range = valor_medio
      ENDWHERE
   CASE (1)
      WHERE (octant == 1)
         Calcular_Range = valor_medio
      ELSEWHERE
         Calcular_Range = goal%range%max
      ENDWHERE
   END SELECT
 END FUNCTION Calcular_Range

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Borrar_empty_leaves                     !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 RECURSIVE SUBROUTINE Borrar_empty_leaves(goal)
   TYPE(CELL),POINTER :: goal
   INTEGER :: i,j,k

   IF (ASSOCIATED(goal%subcell(1,1,1)%ptr)) THEN
      DO i = 1,2
         DO j = 1,2
            DO k = 1,2
               CALL Borrar_empty_leaves(goal%subcell(i,j,k)%ptr)
               IF (goal%subcell(i,j,k)%ptr%type == 0) THEN
                  DEALLOCATE (goal%subcell(i,j,k)%ptr)
               END IF
            END DO
         END DO
      END DO
   END IF
 END SUBROUTINE Borrar_empty_leaves

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Borrar_tree                             !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
 RECURSIVE SUBROUTINE Borrar_tree(goal)
   TYPE(CELL),POINTER :: goal
   INTEGER :: i,j,k

      DO i = 1,2
         DO j = 1,2
            DO k = 1,2
```

```
               IF (ASSOCIATED(goal%subcell(i,j,k)%ptr)) THEN
                  CALL Borrar_tree(goal%subcell(i,j,k)%ptr)
                  DEALLOCATE (goal%subcell(i,j,k)%ptr)
               END IF
            END DO
         END DO
      END DO
  END SUBROUTINE Borrar_tree

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calculate_masses                        !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  RECURSIVE SUBROUTINE Calculate_masses(goal)
    TYPE(CELL),POINTER :: goal
    INTEGER :: i,j,k
    REAL :: mass
    REAL, DIMENSION(3) :: c_o_m

    goal%mass = 0
    goal%c_o_m = 0

    SELECT CASE (goal%type)
       CASE (1)
          goal%mass = m(goal%pos)
          goal%c_o_m = r(goal%pos,:)
       CASE (2)
          DO i = 1,2
             DO j = 1,2
                DO k = 1,2
                   IF (ASSOCIATED(goal%subcell(i,j,k)%ptr)) THEN
                      CALL Calculate_masses(goal%subcell(i,j,k)%ptr)
                      mass = goal%mass
                      goal%mass = goal%mass + goal%subcell(i,j,k)%ptr%mass
                      goal%c_o_m = (mass * goal%c_o_m + goal%subcell(i,j,k)%ptr%mass * &
                           goal%subcell(i,j,k)%ptr%c_o_m) /  goal%mass
                   END IF
                END DO
             END DO
          END DO
    END SELECT
  END SUBROUTINE Calculate_masses


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calculate_forces                        !!
!!                                         !!
!! Desviación de la Solución 1. En este    !!
!! caso no vamos a calcular las fuerzas    !!
!! ejercidas sobre un número reducido de   !!
!! partículas, sino que vamos a calcular   !!
!! como nuestro árbol (que sólo representa!!
!! las partículas que caen en nuestro      !!
!! octante) afecta a todas las partículas !!
!!                                         !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  RECURSIVE SUBROUTINE Calculate_forces(head)
    TYPE(CELL),POINTER :: head
    INTEGER :: i,j,k,start,end

    DO i = 1,n
       CALL Calculate_forces_aux(i,head)
    END DO

  END SUBROUTINE Calculate_forces
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Calculate_forces_aux                    !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  RECURSIVE SUBROUTINE Calculate_forces_aux(number,tree)
    TYPE(CELL),POINTER :: tree
    INTEGER :: i,j,k,number
    REAL :: l,D

    SELECT CASE (tree%type)
       CASE (1)
          IF (number .NE. tree%pos) THEN
             rji = tree%c_o_m - r(number,:)
             r2 = SUM(rji**2)
             r3 = r2 * SQRT(r2)
             a(number,:) = a(number,:) + m(tree%pos) * rji / r3
          END IF
       CASE (2)
          l = tree%range%max(1) - tree%range%min(1) !! El rango tiene el mismo span en las 3 dimensior
          rji = tree%c_o_m - r(number,:)
          r2 = SUM(rji**2)
          D = SQRT(r2)
          IF (l/D < theta) THEN
             !! Si conglomerado, tenemos que ver si se cumple l/D < @
             r3 = r2 * D
             a(number,:) = a(number,:) + tree%mass * rji / r3
          ELSE
             DO i = 1,2
                DO j = 1,2
                   DO k = 1,2
                      IF (ASSOCIATED(tree%subcell(i,j,k)%ptr)) THEN
                         CALL Calculate_forces_aux(number,tree%subcell(i,j,k)%ptr)
                      END IF
                   END DO
                END DO
             END DO
          END IF
    END SELECT

  END SUBROUTINE Calculate_forces_aux

END PROGRAM tree
```

8.3 Parallel version performance

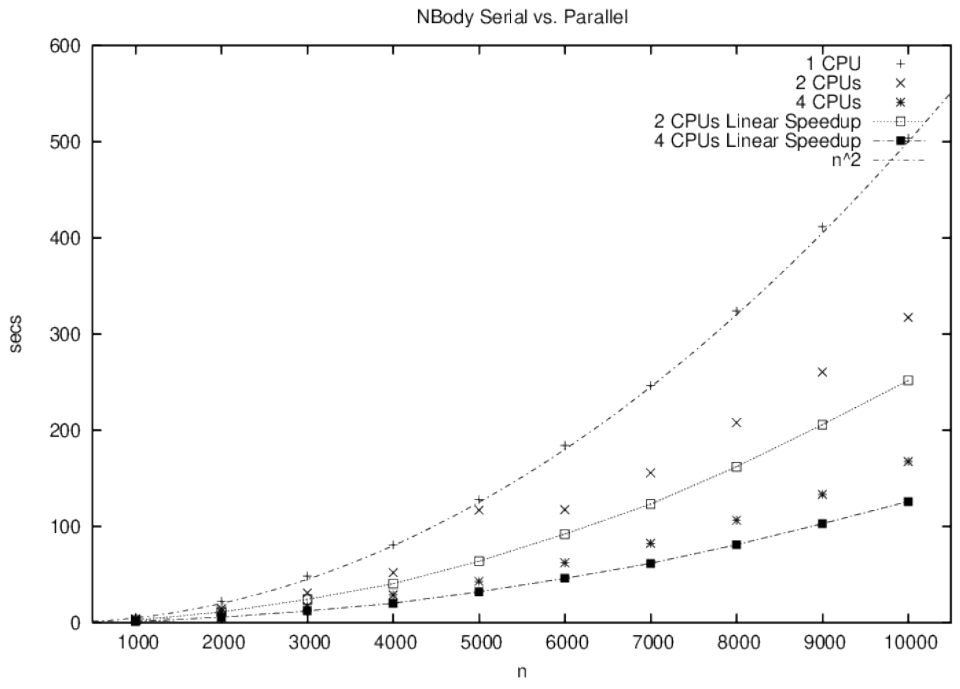


Figure 8.1: Performance Barnes Hut - serial vs. parallel

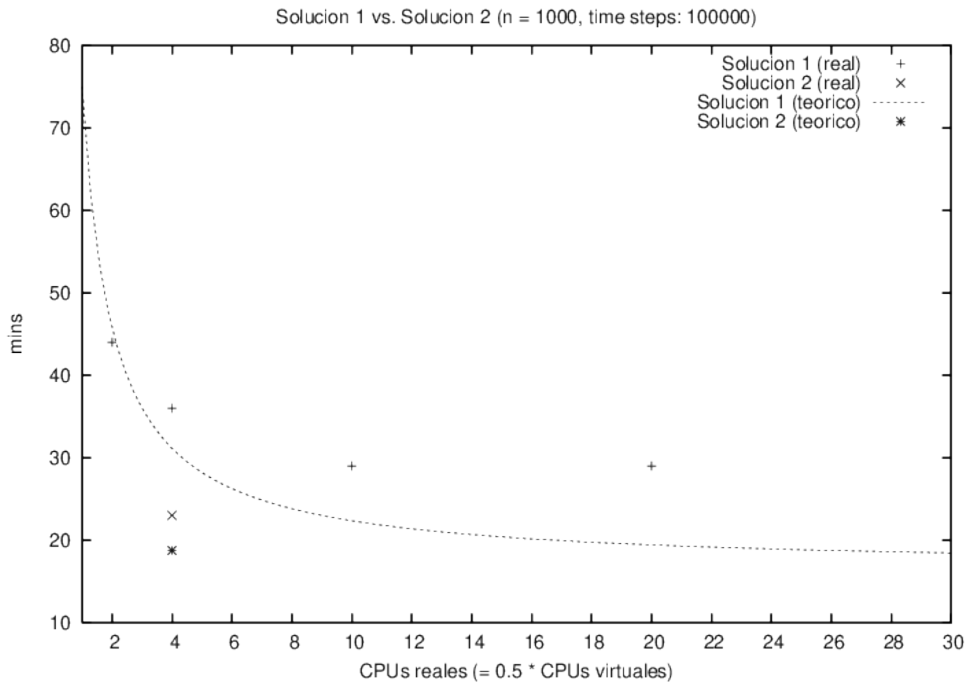


Figure 8.2: Performance Parallel Barnes Hut - solution 1 vs. solution 2

PART IV

APPENDICES

Appendix A

Solutions to exercises

A.1 Exercises in section 2.2

A.1.1 Solution to exercise 2.2.1

ex1.f90

```
PROGRAM REVERSE
  IMPLICIT NONE

  INTEGER :: num
  INTEGER, DIMENSION(:), ALLOCATABLE :: dat

  PRINT*, "Enter number of data to read:"
  READ*, num
  ALLOCATE(dat(num))
  PRINT*, "Enter (in one line)", num, "integers"
  READ*, dat

  PRINT*, "The data in reverse order are:"
  PRINT*, dat(num:1:-1)

END PROGRAM REVERSE
```

A.1.2 Solution to exercise 2.2.2

ex2.f90

```
PROGRAM leapyear
  IMPLICIT NONE

  INTEGER :: i, start, end, base
  PRINT*, "Enter starting and end years"
  READ*, start,end
  PRINT*, "Leap years between years", start, "and", end, "are:"

  base = start / 4
  IF (MOD(start,4) .NE. 0) start = (base+1)*4

  DO i=start,end,4
     IF ((MOD(i,100) .NE. 0) .OR. (MOD(i,400) .EQ. 0)) PRINT*, i
  END DO

END PROGRAM leapyear
```

A.1.3 Solution to exercise 2.2.3

ex3.f90

```
!! To verify: http://kinetigram.com/mck/LinearAlgebra/JPaisMatrixMult04/classes/JPaisMatrixMult04.html

PROGRAM matmul
  IMPLICIT NONE

  INTEGER :: m,n,p
  INTEGER :: i,j
  REAL, DIMENSION(:,:), ALLOCATABLE :: A,B,C


  READ*, m,n,p

  ALLOCATE(A(m,n))
  ALLOCATE(B(n,p))
  ALLOCATE(C(m,p))

  DO i=1,m
     READ*, A(i,:)
  END DO

  DO i=1,n
     READ*, B(i,:)
  END DO

  DO i=1,m
     DO j=1,p
        C(i,j) = SUM(A(i,:) * B(:,j))
     END DO
  END DO

  DO i=1,m
     PRINT*, C(i,:)
  END DO

END PROGRAM matmul
```

A.1.4 Solution to exercise 2.2.4

ex4.f90

```
PROGRAM PALINDROMIC
  IMPLICIT NONE

  INTEGER :: num, temp, digits, i
  INTEGER, DIMENSION(:), ALLOCATABLE :: data
  LOGICAL :: palin

  PRINT*, "Enter num"
  READ*, num
  temp = num
  digits = 0

  DO WHILE (temp .GT. 0)
     temp = temp / 10
     digits = digits + 1
  END DO

  ALLOCATE(data(digits))

  temp = num
  digits=0
  DO WHILE (temp .GT. 0)
     digits = digits + 1
     data(digits) = MOD(temp,10)
     temp = temp / 10
  END DO

  palin = .TRUE.

  DO i=1,digits/2
     IF (data(i) .NE. data(digits-i+1)) THEN
        palin = .FALSE.
        EXIT
     END IF
  END DO
  IF (palin) THEN
     PRINT*, "Palindromic"
  ELSE
     PRINT*, "NOT Palindromic"
  END IF

END PROGRAM PALINDROMIC
```

A.1.5 Solution to exercise 2.2.5

ex5.f90

```
PROGRAM SUMMATION
  IMPLICIT NONE

  INTEGER :: n,w,i, sumi
  REAL, DIMENSION(:), ALLOCATABLE :: data
  REAL :: bsum = 0, psum

  READ*, n
  READ*, w

  ALLOCATE(data(n))
  DO i=1,n
     READ*, data(i)
  END DO

  DO i=w,n
     psum = SUM(data(i-w+1:i))
     IF (psum > bsum) THEN
        bsum = psum
        sumi = i
     END IF
  END DO

  PRINT*, "Greatest sum is:", bsum, "given by the following numbers:"
  DO i=sumi-w+1,sumi
     PRINT*, data(i)
  END DO

END PROGRAM SUMMATION
```

A.1.6 Solution to exercise 2.2.6

ex6.f90

```
PROGRAM SALARIES_COST
  IMPLICIT NONE

  INTEGER :: n,nc,i

  REAL, DIMENSION(:), ALLOCATABLE :: salaries
  INTEGER, DIMENSION(:), ALLOCATABLE :: categories
  REAL, DIMENSION(:), ALLOCATABLE :: payrise

  REAL :: curr_cost, new_cost

  READ*, n
  READ*, nc

  ALLOCATE(salaries(n))
  ALLOCATE(categories(n))
  ALLOCATE(payrise(nc))

  DO i=1,n
     READ*, salaries(i)
  END DO

  DO i=1,n
     READ*, categories(i)
  END DO

  DO i=1,nc
     READ*, payrise(i)
  END DO
  payrise = 1 + payrise/100  ! Convert it to a factor

  curr_cost = SUM(salaries)

  new_cost = 0
  DO i=1,n
     new_cost = new_cost + salaries(i) * payrise(categories(i))
  END DO

  PRINT*, "Current cost:",curr_cost," New cost:",new_cost,"Difference:", new_cost - curr_cost
END PROGRAM SALARIES_COST
```

A.1.7 Solution to exercise 2.2.7

ex7.f90

```
PROGRAM TSP
  IMPLICIT NONE

  INTEGER :: n,i,j,k,l,m
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: distances
  INTEGER, DIMENSION(:), ALLOCATABLE :: route
  INTEGER :: dist_travelled, min_dist = HUGE(0)

  READ*, n
  IF (n .NE. 5) STOP "This version only works with n == 5"
  ALLOCATE(distances(n,n))
  ALLOCATE(route(n))

  DO i=1,n
     READ*, distances(i,:)
  END DO

  LI: DO I = 1, n
     LJ: DO J = 1, n
        IF (J==I) CYCLE
        LK: DO K = 1, n
           IF (K==J .OR. K==I) CYCLE
           LL: DO L = 1, n
              IF (L==J .OR. L==K .OR. L==I ) CYCLE
              LM: DO M = 1, n
                 IF (M==L .OR. M==K .OR. M==J .OR. M==I) CYCLE
                 dist_travelled = distances(I,J) + distances(J,K) + distances(K,L) + &
                      distances(L,M) + distances(M,I)
                 IF (dist_travelled < min_dist) THEN
                    min_dist = dist_travelled
                    route(1) = I
                    route(2) = J
                    route(3) = K
                    route(4) = L
                    route(5) = M
                 END IF
              END DO LM
           END DO LL
        END DO LK
     END DO LJ
  END DO LI
  PRINT*,'Shortest route travelled is :', route
  PRINT*,'Distance travelled = ',min_dist

END PROGRAM TSP
```

A.1.8 Solution to exercise 2.2.8

ex8.f90

```
PROGRAM leapfrog
  IMPLICIT NONE
  INTEGER :: i,j,k
  INTEGER :: n
  REAL :: dt, t_end, t, dt_out, t_out
  REAL :: rs, r2, r3

  REAL, DIMENSION(:), ALLOCATABLE :: m
  REAL, DIMENSION(:,:), ALLOCATABLE :: r,v,a
  REAL, DIMENSION(3) :: rji

  READ*, dt
  READ*, dt_out
  READ*, t_end
  READ*, n

  ALLOCATE(m(n))
  ALLOCATE(r(n,3))
  ALLOCATE(v(n,3))
  ALLOCATE(a(n,3))

  DO i = 1, n
     READ*, m(i), r(i,:),v(i,:)
  END DO

  a = 0.0
  DO i = 1,n
     DO j = i+1,n
        rji = r(j,:) - r(i,:)
        r2 = SUM(rji**2)
        r3 = r2 * SQRT(r2)
        a(i,:) = a(i,:) + m(j) * rji / r3
        a(j,:) = a(j,:) - m(i) * rji / r3
     END DO
  END DO

  t_out = 0.0
  DO t = 0.0, t_end, dt
     v = v + a * dt/2
     r = r + v * dt

     a = 0.0
     DO i = 1,n
        DO j = i+1,n
           rji = r(j,:) - r(i,:)
           r2 = SUM(rji**2)
```

```
            r3 = r2 * SQRT(r2)
            a(i,:) = a(i,:) + m(j) * rji / r3
            a(j,:) = a(j,:) - m(i) * rji / r3
         END DO
      END DO

      v = v + a * dt/2

      t_out = t_out + dt
      IF (t_out >= dt_out) THEN
         DO i = 1,n
            PRINT*, r(i,:)
         END DO
         t_out = 0.0
      END IF

   END DO

END PROGRAM leapfrog
```

A.1.8.1 *Execution example*

The code needs only a few input values. For example, we can run it with a special three-body configuration with the following data:

```
0.001                                                   dt (time step)
0.1                                                     dt_im (printing time steps)
100                                                     t (total time)
3                                                       n (number of bodies)
1.0 .9700436 -.24308753 0.0 .466203685 0.43236573 0.0   m x y z vx vy vz
1.0 -.9700436 .24308753 0.0 .466203685 0.43236573 0.0   m x y z vx vy vz
1.0 0.0 0.0 0.0   -0.93249737 -0.86473146 0.0           m x y z vx vy vz
```

As output, the code will print, after every dt_im, the positions of each body, with the following format:

```
-0.965142 0.163212 -0.153059         x  y  z
0.965142 0.163212 0.153059           x  y  z
-0.965142 -0.163212 0.153059         x  y  z
```

As before, be can redirect the standard input (so that instead of typing the input data, the code will just read them from a file), but we can also redirect the standard output (so the output will end up in a file instead of being printed to the terminal).

```
$ ./leapfrog < nbody.in > nbody.out
```

We can verify that the generated output is correct by plotting (for example, with gnuplot) all the positions over time of the three bodies. If correct, the three bodies will follow an infinite sign pattern, as can be seen in figure A.1.

```
$ gnuplot
gnuplot> plot ``nbody.out''
gnuplot> quit
```

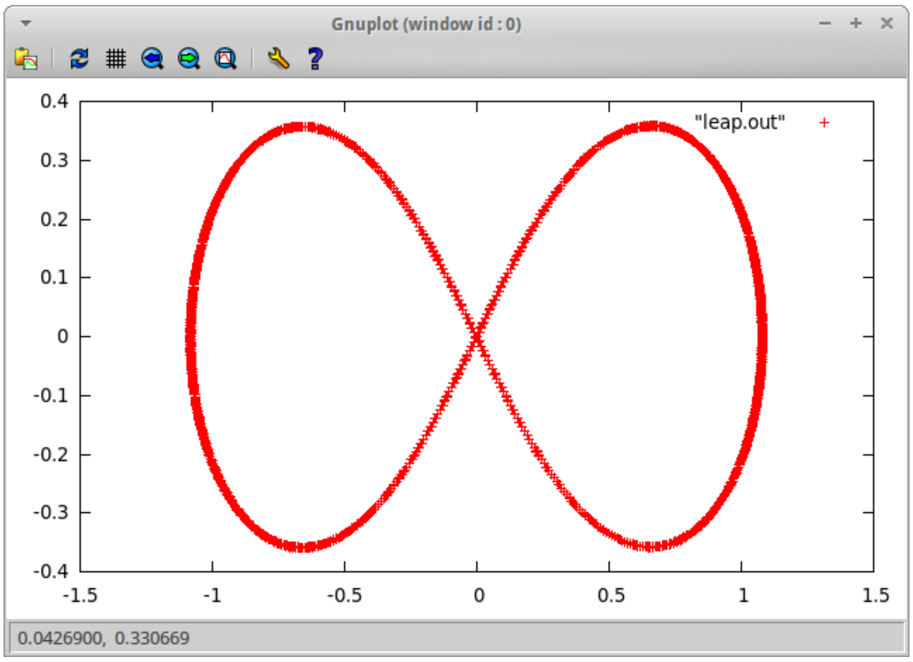


Figure A.1: Execution example

A.2 Exercises in section 4.2

A.2.1 Solution to exercise 4.2.1

rec1.f90

```
PROGRAM Fibo
  IMPLICIT NONE

  INTEGER, PARAMETER :: min = 35, max = 40
  INTEGER :: i

  DO i = min,max
     PRINT*, i, "-->", fibonacci(i)
  END DO

CONTAINS

  RECURSIVE FUNCTION FIBONACCI(N) RESULT (FIBO_RESULT)
    INTEGER, INTENT(IN)        :: N
    INTEGER                    :: FIBO_RESULT
    IF ( N <= 2 ) THEN
       FIBO_RESULT = 1
    ELSE
       FIBO_RESULT = FIBONACCI(N-1) + FIBONACCI(N-2)
    END IF
  END FUNCTION FIBONACCI

END PROGRAM Fibo
```

A.2.2 Solution to exercise 4.2.2

rec2.f90

```
PROGRAM Fibonacci_iterative
  IMPLICIT NONE

  INTEGER, PARAMETER :: min = 30000, max = 45000
  INTEGER :: i

  DO i = min,max
     PRINT*, i, "-->", fibonacci(i)
  END DO

CONTAINS

  INTEGER FUNCTION FIBONACCI(N)
    INTEGER, INTENT(IN)      :: N
    INTEGER :: a,b,t,i

    a=1 ; b=1

    if ( N <= 2 ) THEN
       fibonacci = 1
    else
       do i=3,n
          t=a
          a=b
          b=b+t
       end do
       fibonacci = b
    end if
  end FUNCTION FIBONACCI

END PROGRAM Fibonacci_iterative
```

A.2.3 Solution to exercise 4.2.3

rec3_counter.f90

```
PROGRAM Fibo_counter
  IMPLICIT NONE

  INTEGER, PARAMETER :: min = 35, max = 40
  INTEGER :: i

  DO i = min,max
     PRINT*, i, "-->", fibonacci_c(i)
  END DO

CONTAINS

  RECURSIVE FUNCTION FIBONACCI_C(N) RESULT (FIBO_RESULT)
    INTEGER, INTENT(IN)      :: N
    INTEGER                  :: FIBO_RESULT
    IF ( N <= 2 ) THEN
       FIBO_RESULT = 1
    ELSE
       FIBO_RESULT = 1 + FIBONACCI_C(N-1) + FIBONACCI_C(N-2)
    END IF
  END FUNCTION FIBONACCI_C

END PROGRAM Fibo_Counter
```

rec3_memo.f90

```
PROGRAM Fibonacci_memoization
```

```
  IMPLICIT NONE

  INTEGER, PARAMETER :: min = 30000, max = 45000
  INTEGER :: i
  INTEGER, DIMENSION(:), ALLOCATABLE :: fibs

  ALLOCATE(fibs(max))
  fibs = -1

  DO i = min,max
     PRINT*, i, "-->", fibonacci(i)
  END DO

CONTAINS

  RECURSIVE FUNCTION FIBONACCI(N) RESULT (FIBO_RESULT)
    INTEGER, INTENT(IN)      :: N
    INTEGER                  :: FIBO_RESULT, fibs_1, fibs_2
    IF ( N <= 2 ) THEN
       FIBO_RESULT = 1
       fibs(n) = 1
    ELSE
       if (fibs(n-1) .ne. -1) then
          fibs_1 = fibs(n-1)
       else
          fibs_1 = fibonacci(N-1)
          fibs(n-1) = fibs_1
       end if

       if (fibs(n-2) .ne. -1) then
          fibs_2 = fibs(n-2)
       else
          fibs_2 = fibonacci(N-2)
          fibs(n-2) = fibs_2
       end if

       FIBO_RESULT = fibs_1 + fibs_2
    END IF
  END FUNCTION FIBONACCI

END PROGRAM Fibonacci_memoization
```

A.2.4 Solution to exercise 4.2.4

rec4.txt

```
RECURSIVE SUBROUTINE print_sorted (tree)
  IF (left_branch_exists(tree)) print_sorted(left_branch(tree))
  PRINT*, node_value(tree)
  IF (right_branch_exists(tree)) print_sorted(right_branch(tree))
END print_sorted
```

A.2.5 Solution to exercise 4.2.5

rec5.f90

```
PROGRAM Hanoi
  IMPLICIT NONE

  INTEGER :: pieces
  PRINT*, "Enter number of pieces"
  READ*, pieces

  CALL solve(pieces,1,3,2)

CONTAINS

  RECURSIVE SUBROUTINE solve(pieces,from,to,aux)
    INTEGER :: pieces,from,to,aux

    IF (pieces .EQ. 1) THEN
       PRINT*, pieces, "(",from," -> ",to,")"
    ELSE
       CALL solve(pieces-1,from,aux,to)
       PRINT*, pieces, "(",from," -> ",to,")"
       CALL solve(pieces-1,aux,to,from)
    END IF
  END SUBROUTINE solve
END PROGRAM Hanoi
```

A.2.6 Solution to exercise 4.2.6

rec6.f90

```
PROGRAM TP
  IMPLICIT NONE

  INTEGER :: n,i
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: distances
  INTEGER, DIMENSION(:), ALLOCATABLE :: route, route_temp
  LOGICAL, DIMENSION(:), ALLOCATABLE :: visited
  INTEGER :: dist_travelled, min_dist = HUGE(0)

  READ*, n
  ALLOCATE(distances(n,n))
  ALLOCATE(route(n),route_temp(n),visited(n))

  DO i=1,n
     READ*, distances(i,:)
  END DO

  visited = .FALSE.
  dist_travelled = 0

  CALL tsp(1)

  PRINT*, "Shortest route travelled is :", route
  PRINT*, "Distance travelled = ", min_dist
CONTAINS

  RECURSIVE SUBROUTINE tsp (level)
    INTEGER :: level, city, cur_dist

    IF (level < n) THEN
       DO city=1,n
          IF (.NOT. visited(city)) THEN
             visited(city) = .TRUE.
             route_temp(level) = city
             cur_dist = dist_travelled
             IF (level > 1) THEN
                dist_travelled = dist_travelled + distances(route_temp(level-1),city)
             END IF
```

```
!             PRINT*, "At level", level, " Route", route_temp(1:level)," Distance",dist_travelled

             CALL tsp(level+1)

             dist_travelled = cur_dist
             visited(city) = .FALSE.
          END IF
       END DO
    ELSE
       DO city=1,n
          IF (.NOT. visited(city)) THEN
             visited(city) = .TRUE.
             route_temp(level) = city
             cur_dist = dist_travelled
             dist_travelled = dist_travelled + distances(route_temp(level-1),city) + &
                  distances(city,route_temp(1))

!             PRINT*, "CIRCUIT: ", route_temp(1:level)," Distance: ",dist_travelled

             IF (dist_travelled < min_dist) THEN
                min_dist = dist_travelled
                route = route_temp
!                PRINT*, "--- IMPROVEMENT: ",dist_travelled
             END IF

             dist_travelled = cur_dist
             visited(city) = .FALSE.
          END IF
       END DO


    END IF

  END SUBROUTINE tsp

END PROGRAM TP
```

A.2.7 Solution to exercise 4.2.7

rec7.f90

```
PROGRAM CON_DIGITS
  IMPLICIT NONE

  INTEGER :: n1,n2

  DO
     READ*, n1,n2
     IF (n1 .EQ. 0) EXIT
     PRINT*, contained_digits(n1,n2)
  END DO

CONTAINS

  RECURSIVE FUNCTION contained_digits(n1,n2) RESULT(res)
    INTEGER :: n1,n2,n2t,rem
    LOGICAL :: res

    n2t = n2
    res = .FALSE.

    IF (n1 / 10 < 1) THEN
```

```
        ! Just one digit. No need for recursion
        DO WHILE (n2t > 0)
           rem = MOD(n2t,10)
           n2t = n2t / 10
           IF (n1 == rem) THEN
              res = .TRUE.
              EXIT
           END IF
        END DO
     ELSE
        IF (contained_digits(MOD(n1,10),n2)) THEN
           res = contained_digits(n1/10,n2)
        END IF
     END IF
  END FUNCTION contained_digits

END PROGRAM CON_DIGITS
```

A.2.8 Solution to exercise 4.2.8

rec8.f90

```
PROGRAM PERMUTATIONS
  IMPLICIT NONE

  INTEGER :: n,naux
  INTEGER :: dn
  INTEGER, DIMENSION(:),ALLOCATABLE :: an,perm
  LOGICAL, DIMENSION(:), ALLOCATABLE :: used

  READ*,n

  naux=n
  dn = 0
  DO WHILE(naux > 0)
     dn = dn + 1
     naux = naux / 10
  END DO

  ALLOCATE(an(dn))

  naux=n
  dn=0
  DO WHILE(naux > 0)
     dn = dn + 1
     an(dn) = MOD(naux,10)
     naux = naux / 10
  END DO
  an(dn:1:-1) = an(1:dn)

  ALLOCATE(used(dn),perm(dn))
  used = .FALSE.

  CALL PERMS(1)

CONTAINS

  RECURSIVE SUBROUTINE PERMS(level)
    INTEGER :: level,i

    IF (level .EQ. dn) THEN
       ! Base case. If here, everytime we add a new number to the last position in perm
       !  we have a new permutation and have to print it.
```

```
        DO i=1,dn
           IF (.NOT. used(i)) THEN
              perm(level) = an(i)
              PRINT*,perm
           END IF
        END DO
     ELSE
        DO i=1,dn
           IF (.NOT. used(i)) THEN
              perm(level) = an(i)
              used(i) = .TRUE.
              CALL PERMS(level+1)
              used(i) = .FALSE.
           END IF
        END DO
     END IF

   END SUBROUTINE PERMS

 END PROGRAM PERMUTATIONS
```

A.2.9 Solution to exercise 4.2.9

rec9.f90

```
PROGRAM QUEENS
  IMPLICIT NONE

  LOGICAL, DIMENSION(:,:), ALLOCATABLE :: board
  INTEGER :: n,solutions

  PRINT*, "Board size (side)"
  READ*,n

  ALLOCATE(board(n,n))
  board = .FALSE.   ! .FALSE. = empty cell, .TRUE. = a cell with a queen
  solutions = 0

  CALL add_queen(1,n)
  PRINT*, "Number of solutions (not taking into account rotation and reflection): ", solutions

CONTAINS

  RECURSIVE SUBROUTINE add_queen(row,n)
    INTEGER, INTENT(IN) :: row,n
    INTEGER :: col

    IF (row .EQ. n) THEN
       ! Base case. If we are able to place a queen in this row, then print board

       DO col=1,n
          board(row,col) = .TRUE.
          IF (valid_board(row,col,n)) THEN
             CALL print_board(n)
             solutions = solutions + 1
          END IF
          board(row,col) = .FALSE.
       END DO
    ELSE
       DO col=1,n
          board(row,col) = .TRUE.
          IF (valid_board(row,col,n)) THEN
             CALL add_queen(row+1,n)
```

```
        END IF
        board(row,col) = .FALSE.
     END DO
  END IF
END SUBROUTINE add_queen

SUBROUTINE print_board(n)
  INTEGER :: n,row

  PRINT*,""
  PRINT*, "=================="
  DO row=1,n
     PRINT*, board(row,:)
  END DO
END SUBROUTINE print_board

! This function will just make sure that if we add a queen in [row,col]
!  this new queen doesn't attack any previously located queens (i.e. those
!  positions in the board == .TRUE.
FUNCTION valid_board(row,col,n)
  LOGICAL :: valid_board
  INTEGER :: row,col,n,ri,ci,count,diff

  valid_board = .TRUE.

  ! Horizontally
  count = 0
  DO ci=1,n
     IF (board(row,ci)) count = count + 1
  END DO
  IF (count > 1) THEN
     valid_board = .FALSE.
     RETURN
  END IF

  ! Vertically
  count = 0
  DO ri=1,n
     IF (board(ri,col)) count = count + 1
  END DO
  IF (count > 1) THEN
     valid_board = .FALSE.
     RETURN
  END IF

  ! SE diagonal
  count = 0
  diff = MIN(row,col) - 1
  ri = row-diff ; ci = col-diff
  DO
     IF (ri > n .OR. ci > n) EXIT
     IF (board(ri,ci)) count = count + 1
     ri = ri + 1 ; ci = ci + 1
  END DO
  IF (count > 1) THEN
     valid_board = .FALSE.
     RETURN
  END IF

  ! NE diagonal
  count = 0
  diff = MIN(n-row,col-1)
  ri = row+diff ; ci = col-diff
  DO
     IF (ri < 1 .OR. ci > n) EXIT
```

```
         IF (board(ri,ci)) count = count + 1
         ri = ri - 1 ; ci = ci + 1
      END DO
      IF (count > 1) THEN
         valid_board = .FALSE.
         RETURN
      END IF


    END FUNCTION valid_board

  END PROGRAM QUEENS
```

A.2.10 Solution to exercise 4.4.1

pointer1.f90

```
  PROGRAM POINTERdef

    REAL, TARGET, DIMENSION(1:10)   :: VECTOR
    REAL, POINTER, DIMENSION(:)     :: POINTER1
    REAL, POINTER                   :: POINTER2

    VECTOR = (/ 1,2,3,4,5,6,7,8,9,10 /)

    POINTER1 => VECTOR
    POINTER2 => VECTOR(7)

    PRINT*, "vector", POINTER1
    PRINT*, "scalar", POINTER2

  END PROGRAM POINTERdef
```

A.2.11 Solution to exercise 4.4.2

pointer2.f90

```
  PROGRAM pointerswap
    IMPLICIT NONE

    INTEGER, TARGET :: s1,s2
    INTEGER, POINTER :: p1,p2

    s1=4 ; s2=6
    p1=>s1 ; p2=>s2

    CALL swap_with_pointers(p1,p2)

    PRINT*, "Via pointers:" ,p1,p2
    PRINT*, "Via variables:",s1,s2

  CONTAINS

    SUBROUTINE swap_with_pointers(pt1,pt2)
      INTEGER, INTENT(IN), POINTER :: pt1,pt2
      INTEGER :: swap

      swap = pt1
      pt1 = pt2
      pt2 = swap
    END SUBROUTINE swap_with_pointers

  END PROGRAM pointerswap
```

A.2.12 Solution to exercise 4.4.3

pointer3.f90

```
PROGRAM array_alias
  IMPLICIT NONE

  INTEGER, TARGET, DIMENSION(1:10) :: VECTOR
  INTEGER, POINTER, DIMENSION(:)    :: ODD, EVEN

  ODD => VECTOR(1:10:2)
  EVEN => VECTOR(2:10:2)

  EVEN = 13
  ODD = 17

  PRINT*, "whole vector:", VECTOR
  PRINT*, "odd elements:", ODD
  PRINT*, "even elements:", EVEN

END PROGRAM array_alias
```

A.2.13 Solution to exercise 4.5.1

bidirectional.f90

```
PROGRAM bidirectional
  IMPLICIT NONE

  TYPE CELL
     INTEGER :: val
     TYPE (CELL), POINTER :: prev
     TYPE (CELL), POINTER :: next
  END TYPE CELL

  TYPE (CELL), TARGET  :: head
  TYPE (CELL), POINTER :: curr, temp
  INTEGER              :: n,k,i

  head%val = 0
  NULLIFY(head%prev)
  NULLIFY(head%next)
  curr => head

  PRINT*, "Input number of elements in the list"
  READ*, n

  PRINT*, "Now enter", n, " elements"
  DO i=1,n
     READ*, k
     ALLOCATE(temp)
     temp%val = k
     NULLIFY(temp%prev)
     NULLIFY(temp%next)
     curr%next => temp
     temp%prev => curr
     curr => temp
  END DO

PRINT*, " Backwards..."
CALL Print (curr,0)

PRINT*, " Forward..."
curr => head
CALL Print (curr,1)

CONTAINS
  RECURSIVE SUBROUTINE Print (ptr,forward)
    TYPE (CELL), POINTER :: ptr
    INTEGER :: forward

    PRINT*, ptr%val
    IF (forward == 1 .AND. ASSOCIATED(ptr%next)) CALL Print (ptr%next,1)
    IF (forward == 0 .AND. ASSOCIATED(ptr%prev)) CALL Print (ptr%prev,0)

  END SUBROUTINE Print

END PROGRAM bidirectional
```

A.2.14 Solution to exercise 4.5.2

sortedbidirectional.f90

```
PROGRAM sortedbidirectional
  IMPLICIT NONE

  TYPE CELL
     INTEGER :: val
     TYPE (CELL), POINTER :: prev
     TYPE (CELL), POINTER :: next
  END TYPE CELL

  TYPE (CELL), TARGET  :: head
  TYPE (CELL), POINTER :: curr, temp
  INTEGER              :: n,k,i

  head%val = 0
  NULLIFY(head%prev)
  NULLIFY(head%next)

  PRINT*, "Input number of elements in the list"
  READ*, n

  PRINT*, "Now enter", n, " elements"
  DO i=1,n
     READ*, k
     ALLOCATE(temp)
     temp%val = k
     NULLIFY(temp%prev)
     NULLIFY(temp%next)

     curr => head
     DO
        IF (ASSOCIATED(curr%next)) THEN
           IF (curr%next%val .GE. k) THEN
              EXIT
           ELSE
              curr => curr%next
           END IF
```

```
            ELSE
               EXIT
            END IF
         END DO

         IF (ASSOCIATED(curr%next)) THEN
            curr%next%prev => temp
         END IF
         temp%next => curr%next
         temp%prev => curr
         curr%next => temp

      END DO

    PRINT*, " Backwards..."
    curr => head
    DO
       IF (.NOT. ASSOCIATED(curr%next)) EXIT
       curr => curr%next
    END DO
    CALL Print (curr,0)

    PRINT*, " Forward..."
    curr => head
    CALL Print (curr,1)

    CONTAINS
      RECURSIVE SUBROUTINE Print (ptr,forward)
        TYPE (CELL), POINTER :: ptr
        INTEGER :: forward

        PRINT*, ptr%val
        IF (forward == 1 .AND. ASSOCIATED(ptr%next)) CALL Print (ptr%next,1)
        IF (forward == 0 .AND. ASSOCIATED(ptr%prev)) CALL Print (ptr%prev,0)

      END SUBROUTINE Print

    END PROGRAM sortedbidirectional
```

A.2.15 Solution to exercise 4.5.3

bst.f90

```
    PROGRAM bst
      IMPLICIT NONE

      TYPE CELL
         INTEGER :: val
         TYPE (CELL), POINTER :: left,right
      END TYPE CELL

      TYPE (CELL), POINTER :: head
      INTEGER              :: n,k,i

      PRINT*, "Input number of elements in the list"
      READ*, n
      PRINT*, "Now enter", n, " elements"

      READ*, k
      ALLOCATE(head)
      head%val = k
      NULLIFY(head%left)
      NULLIFY(head%right)
      DO i=2,n
         READ*, k
         CALL place_number(head,k)
      END DO

      CALL Print(head)

    CONTAINS
      RECURSIVE SUBROUTINE place_number(node,number)
        TYPE (CELL), POINTER :: node, temp
        INTEGER :: number

        IF (number < node%val) THEN
           IF (ASSOCIATED(node%left)) THEN
              CALL place_number(node%left,number)
           ELSE
              ALLOCATE(temp)
              node%left => temp
              temp%val = number
              NULLIFY(temp%left)
              NULLIFY(temp%right)
           END IF
        ELSE IF (number > node%val) THEN
           IF (ASSOCIATED(node%right)) THEN
              CALL place_number(node%right,number)
           ELSE
              ALLOCATE(temp)
              node%right => temp
              temp%val = number
              NULLIFY(temp%left)
              NULLIFY(temp%right)
           END IF
        ELSE
           PRINT*, "Repeated numbers not allowed, ignoring: ",number, " !"
        END IF
      END SUBROUTINE place_number

      RECURSIVE SUBROUTINE Print (node)
        TYPE (CELL), POINTER :: node

        IF (ASSOCIATED(node%left)) CALL Print (node%left)
        PRINT*, node%val
        IF (ASSOCIATED(node%right)) CALL Print (node%right)

      END SUBROUTINE Print

    END PROGRAM bst
```

A.2.16 Solution to exercise 4.5.4

```
    RECURSIVE FUNCTION numelem(node) RESULT (num)
      TYPE (CELL), POINTER :: node
      INTEGER :: num

      num = 1

      IF (ASSOCIATED(node%left)) num = num + numelem(node%left)
      IF (ASSOCIATED(node%right)) num = num + numelem(node%right)
    END FUNCTION numelem
```

A.2.17 Solution to exercise 4.5.5

```
RECURSIVE FUNCTION depth(node) RESULT (num)
  TYPE (CELL), POINTER :: node
  INTEGER :: num, dleft, dright

  IF (ASSOCIATED(node%left)) THEN
     dleft = depth(node%left)
  ELSE
     dleft = 0
  END IF

  IF (ASSOCIATED(node%right)) THEN
     dright = depth(node%right)
  ELSE
     dright = 0
  END IF

  IF (dleft .GT. dright) THEN
     num = 1+dleft
  ELSE
     num = 1+dright
  END IF
END FUNCTION depth
```

A.2.18 Solution to exercise 4.5.6

```
RECURSIVE FUNCTION in_order_successor(node) RESULT (tnode)
  TYPE (CELL), POINTER :: node,tnode

  IF (ASSOCIATED(node%left)) THEN
     tnode => in_order_successor(node%left)
  ELSE
     tnode => node
  END IF

END FUNCTION in_order_successor

RECURSIVE FUNCTION get_parent (node,number) RESULT (tnode)
  TYPE (CELL), POINTER :: node,tnode
  INTEGER :: number

  IF (number .LT. node%val) THEN
     IF (node%left%val .EQ. number) THEN
        tnode => node
     ELSE
        tnode => get_parent(node%left,number)
     END IF
  ELSE
     IF (node%right%val .EQ. number) THEN
        tnode => node
     ELSE
        tnode => get_parent(node%right,number)
     END IF
  END IF
END FUNCTION get_parent

RECURSIVE SUBROUTINE delete(head,number)
  TYPE (CELL), POINTER :: head, parent, node, successor
  INTEGER :: number, exchange

  parent => get_parent(head,number)

  !! Find the node to actually delete
```

```
    IF (ASSOCIATED(parent%left)) THEN
       IF (parent%left%val .EQ. number) THEN
          node => parent%left
       ELSE
          node => parent%right
       END IF
    ELSE
       node => parent%right
    END IF

    !! Do the actual delete
    IF (.NOT. ASSOCIATED(node%left) .AND. .NOT. ASSOCIATED(node%right)) THEN
       !! The easiest. This is a leaf
       IF (ASSOCIATED(parent%left)) THEN
          IF (parent%left%val .EQ. number) THEN
             NULLIFY(parent%left)
          ELSE
             NULLIFY(parent%right)
          END IF
       ELSE
          NULLIFY(parent%right)
       END IF
       DEALLOCATE(node)
    ELSEIF (.NOT. ASSOCIATED(node%left)) THEN
       !! There is right branch
       IF (ASSOCIATED(parent%left)) THEN
          IF (parent%left%val .EQ. number) THEN
             parent%left => node%right
          ELSE
             parent%right => node%right
          END IF
       ELSE
          parent%right => node%right
       END IF
       DEALLOCATE(node)
    ELSEIF (.NOT. ASSOCIATED(node%right)) THEN
       !! There is left branch
       IF (ASSOCIATED(parent%left)) THEN
          IF (parent%left%val .EQ. number) THEN
             parent%left => node%left
          ELSE
             parent%right => node%left
          END IF
       ELSE
          parent%right => node%left
       END IF
       DEALLOCATE(node)
    ELSE
       !! most general case
       successor => in_order_successor(node%right)
       exchange = successor%val
       CALL delete(head,exchange)
       node%val = exchange
    END IF

  END SUBROUTINE delete
```

A.3 Exercises in section 6.2

A.3.1 Solution to exercise 6.2.1

ex1.f90

```
program hello_world
  use mpi
  implicit none

  integer :: my_rank, p, source, dest, tag, error, numb
  integer, dimension(MPI_STATUS_SIZE) :: status

  call MPI_Init ( error )
  call MPI_Comm_size ( MPI_COMM_WORLD, p, error )
  call MPI_Comm_rank ( MPI_COMM_WORLD, my_rank, error )

  if ( my_rank == 0 ) then
     do source = 1, p-1
        call MPI_Recv(numb,1,MPI_INTEGER,source,0,MPI_COMM_WORLD,status,error)
        print*, "Greetings from process ", numb, "!"
     end do

  else
     call MPI_Send(my_rank,1,MPI_INTEGER,0,0,MPI_COMM_WORLD,error)
  end if

  call MPI_Finalize ( error )

end program hello_world
```

A.3.2 Solution to exercise 6.2.2

ex2.f90

```
PROGRAM mpi1
  USE mpi
  IMPLICIT none

  INTEGER :: procs, rank, error, number, send_to
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

  CALL MPI_Init ( error )
  CALL MPI_Comm_size ( MPI_COMM_WORLD, procs, error )
  CALL MPI_Comm_rank ( MPI_COMM_WORLD, rank, error )

  IF ( rank .EQ. procs - 1) THEN
     send_to = 0
  ELSE
     send_to = rank + 1
  END IF

  IF ( rank .EQ. 0 ) THEN
     READ*, number
     CALL MPI_Send(number,1,MPI_INTEGER,send_to,0,MPI_COMM_WORLD,error)
     CALL MPI_Recv(number,1,MPI_INTEGER,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,status,error)
     PRINT*, number
  ELSE
     CALL MPI_Recv(number,1,MPI_INTEGER,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,status,error)
     number = number * (rank + 1)
     CALL MPI_Send(number,1,MPI_INTEGER,send_to,0,MPI_COMM_WORLD,error)
  END IF
  CALL MPI_Finalize ( error )

END PROGRAM mpi1
```

A.3.3 Solution to exercise 6.2.3

ex3.f90

```
PROGRAM mpi2
  USE mpi
  IMPLICIT none

  INTEGER :: procs, rank, error, send_to
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

  INTEGER :: number, number_p, total, partial, start
  INTEGER, DIMENSION( : ), ALLOCATABLE :: data, data_p

  CALL MPI_Init ( error )
  CALL MPI_Comm_size ( MPI_COMM_WORLD, procs, error )
  CALL MPI_Comm_rank ( MPI_COMM_WORLD, rank, error )



  IF ( rank .EQ. 0 ) THEN
     READ*, number
     ALLOCATE (data(number))
     READ*, data

     number_p = number / procs
     total = SUM(data(1:number_p))

     DO send_to = 1, procs - 1
        start = (number_p * send_to) + 1
        CALL MPI_Send(number_p,1,MPI_INTEGER,send_to,1,MPI_COMM_WORLD,error)
        CALL MPI_Send(data(start),number_p,MPI_INTEGER,send_to,0,MPI_COMM_WORLD,error)
     END DO

     DO send_to = 1, procs - 1
        CALL MPI_Recv(partial,1,MPI_INTEGER,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,status,error)
        total = total + partial
     END DO

     PRINT*, "The total is ", total
  ELSE
     CALL MPI_Recv(number_p,1,MPI_INTEGER,0,1,MPI_COMM_WORLD,status,error)
     ALLOCATE (data_p(number_p))
     CALL MPI_Recv(data_p,number_p,MPI_INTEGER,0,0,MPI_COMM_WORLD,status,error)
     partial = SUM(data_p)
     CALL MPI_Send(partial,1,MPI_INTEGER,0,0,MPI_COMM_WORLD,error)
  END IF

  CALL MPI_Finalize ( error )

END PROGRAM mpi2
```

A.3.4 Solution to exercise 6.2.4

ex4.f90

```
program ghost_cells
  use mpi
  implicit none

  integer, parameter :: N=20
  integer :: my_id, num_procs, ierr
  integer :: i,j
  integer :: rem, num_local_col
  integer :: proc_right, proc_left
  integer, allocatable :: matrix(:,:)
  integer status1(MPI_Status_size), status2(MPI_Status_size)

  call mpi_init(ierr)
  call mpi_comm_rank(MPI_COMM_WORLD,my_id,ierr)
  call mpi_comm_size(MPI_COMM_WORLD,num_procs,ierr)

  !  number of columns for each mpi task

  rem= mod(N,num_procs)
  num_local_col = (N - rem)/num_procs

  if(my_id < rem) num_local_col = num_local_col+1

  allocate(matrix(N,num_local_col+2))

  ! inizialization of the local matrix
  matrix = my_id

  proc_right = my_id+1
  proc_left = my_id-1
  if(proc_right .eq. num_procs) proc_right = 0
  if(proc_left < 0) proc_left = num_procs-1

  ! check printings
  write(*,*) "my_id, proc right, proc left ", my_id, proc_right, proc_left
  write(*,*) "my_id, num_local_col ", my_id, num_local_col
  write(*,*) "my_id, matrix(1,1), matrix(1,num_local_col+2), matrix(N,num_local_col+2)", &
       my_id, matrix(1,1), matrix(1,num_local_col+2), &
       matrix(N,num_local_col+2)

  ! send receive of the ghost regions
  call mpi_sendrecv(matrix(:,2),N,MPI_INTEGER,proc_left,10, &
       matrix(:,(num_local_col+2)),N,MPI_INTEGER,proc_right,10, &
       MPI_COMM_WORLD,status1,ierr)

  call mpi_sendrecv(matrix(:,(num_local_col+1)),N,MPI_INTEGER,proc_right, &
       11,matrix(:,1),N,MPI_INTEGER,proc_left,11,MPI_COMM_WORLD,status2,ierr)

  ! check printings
  write(*,*) "my_id ", my_id, " colonna arrivata da sinistra: ", matrix(:,1)
  write(*,*) "my_id ", my_id, " colonna arrivata da destra: ", &
       matrix(:,num_local_col+2)

  deallocate(matrix)

  call mpi_finalize(ierr)

end program ghost_cells
```

A.3.5 Solution to exercise 6.2.5

ex5.f90

```fortran
program distribuite

  use mpi

  implicit none

  character(LEN=50) :: stringa

  integer, parameter :: N = 10

  integer ierr, error
  integer status(MPI_STATUS_SIZE)

  integer k, i, j, rem, iglob, Ncol, Nrow, sup
  integer me, nprocs

  integer, allocatable, dimension(:,:) :: a

  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierr)

  !$ Number of rows that are assigned to each processor, taking care of the remainder
  Nrow = N
  Ncol = N / nprocs

  rem = MOD(N, nprocs)
  if (me < rem) then
     Ncol = Ncol + 1
  endif

  !$ Allocate local workspace (and notice that the array is distributed by columns)
  ALLOCATE( a(Nrow, Ncol) )
  !$ Logical column (Local row) of the first "one" entry
  iglob = (Ncol * me) + 1
  if (me >= rem) then
     iglob = iglob + rem
  endif
  !$ Initilize local matrix
  do j=1,Ncol
     do i=1,Nrow
        if (i == iglob) then
           A(i,j) = 1.0
        else
           A(i,j) = 0.0
        endif
     enddo
     iglob = iglob + 1;
  enddo

  write(stringa, *) Nrow
  !$ Print matrix
  if (me == 0) then
     !$ Rank 0: print local buffer
     do j=1,Ncol
        print '('//trim(stringa)//'(I2))', A(:,j)
     enddo
     !$ Receive new data from other processes
     !$ in an ordered fashion and print the buffer
     do k=1, nprocs-1
        if (k==rem) then
           Ncol = Ncol - 1
        endif
        call MPI_RECV(A, Ncol*Nrow, MPI_INTEGER, k, 0, MPI_COMM_WORLD, status, ierr)
        do j=1,Ncol
```

```
                print '('//trim(stringa)//'(I2))', A(:,j)
             enddo
          enddo
       else
          !$ Send local data to Rank 0
          call MPI_SEND(A, Nrow*Ncol, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, ierr)
       endif
       DEALLOCATE(a)
       call MPI_FINALIZE(ierr)

     end program distribuite
```

A.4 Exercises in section 7.2

A.4.1 Solution to exercise 7.2.1

ex1.f90

```
PROGRAM main
  USE MPI
  IMPLICIT NONE

  INTEGER :: ierr, my_id, num_procs,inserted_num,modified_num,buffer

  CALL MPI_INIT( ierr )
  CALL MPI_COMM_RANK( MPI_COMM_WORLD, my_id, ierr )
  CALL MPI_Comm_size ( MPI_COMM_WORLD, num_procs, ierr )

  IF( my_id == 0) THEN
     ! WRITE(*,*) "Insert an integer value : " ! In case of interactive run
     ! READ(*,*) inserted_num
     inserted_num = 57
     modified_num = inserted_num*inserted_num
  ELSE
     inserted_num = 0
     modified_num = 0
  ENDIF

  CALL MPI_BCAST(modified_num, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
  WRITE(*,*) "my_id", my_id, "inserted_num", inserted_num, "modified_num", modified_num
  CALL MPI_FINALIZE(ierr)
END PROGRAM main
```

A.4.2 Solution to exercise 7.2.2

ex2.f90

```
PROGRAM trapezoidal
  USE MPI
  IMPLICIT NONE

  INTEGER :: n, dest=0, tag=0
  REAL :: a, b
  INTEGER :: my_rank, p, local_n, source, status(MPI_STATUS_SIZE), ierr
  REAL :: h, local_a, local_b, integral, total

  CALL MPI_INIT(ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, p, ierr)

  CALL Get_data (a, b, n, my_rank, p)

  h = (b-a)/n
  local_n = n/p

  local_a = a + my_rank*local_n*h
  local_b = local_a + local_n*h
  integral = Trap(local_a, local_b, local_n, h)

  CALL MPI_REDUCE(integral,total,1,MPI_REAL,MPI_SUM,0,MPI_COMM_WORLD,ierr)

  IF (my_rank .EQ. 0) THEN
     PRINT*, 'With n=', n, 'trapezoids, our estimate'
     PRINT*, 'of the integral from', a, 'to', b, '=', total
  ENDIF
  CALL MPI_FINALIZE(ierr)

CONTAINS

  SUBROUTINE Get_data(a, b, n, my_rank, p)
    REAL :: a, b
    INTEGER :: n, my_rank, p, source=0, dest, tag, status(MPI_STATUS_SIZE), ierr

    IF (my_rank .EQ. 0) THEN
       PRINT*, "Enter a, b and n"
       READ*, a, b, n
    END IF

    CALL MPI_BCAST(a,1,MPI_REAL, 0,MPI_COMM_WORLD, ierr )
    CALL MPI_BCAST(b,1,MPI_REAL, 0,MPI_COMM_WORLD, ierr )
    CALL MPI_BCAST(n,1,MPI_REAL, 0,MPI_COMM_WORLD, ierr )
  END SUBROUTINE Get_data


  REAL FUNCTION f(x)
    REAL :: x

    f = x*x
  END FUNCTION f

  REAL FUNCTION Trap(local_a, local_b, local_n, h)
    REAL    :: local_a, local_b, h, integral, x
    INTEGER :: local_n,i

    integral = (f(local_a) + f(local_b))/2.0
    x = local_a
    DO i = 1, local_n-1
       x = x + h
       integral = integral + f(x)
    END DO
    Trap = integral*h
  END FUNCTION Trap

END PROGRAM trapezoidal
```

A.4.3 Solution to exercise 7.2.3

ex3.f90

```
PROGRAM main
  USE MPI
  IMPLICIT NONE

  INTEGER, parameter::N=50
  INTEGER ::my_id, num_procs, ierr
  INTEGER ::i, num_elem
  INTEGER, dimension(N)::array, array_final
  INTEGER, allocatable, dimension(:)::array_recv
  INTEGER, allocatable:: sendcount(:), displs(:)

  CALL MPI_INIT( ierr )
  CALL MPI_COMM_RANK( MPI_COMM_WORLD, my_id, ierr )
  CALL MPI_Comm_size ( MPI_COMM_WORLD, num_procs, ierr )

  ! proc 0 initializes the principal array
  IF( my_id .eq. 0) THEN
     DO i=1,N
```

```
      array(i)= i
   END DO
END IF

num_elem= N/num_procs
IF (my_id < MOD(N,num_procs)) THEN
   num_elem = num_elem +1
ENDIF

ALLOCATE(array_recv(num_elem))

! in case that N is a multiple of the number of MPI tasks, the same number of
! elements is send to (and received from) by root process to others, so
! mpi_scatter and mpi_gather are called
IF ( MOD(N,num_procs) .eq. 0 ) THEN
   CALL MPI_SCATTER(array, num_elem, MPI_INTEGER, array_recv, num_elem, &
        MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

   WRITE(*,*) "my_id", my_id, "elementi ricevuti:", array_recv(1:num_elem)

   DO i=1,num_elem
      array_recv(i)= array_recv(i)+my_id
   END DO

   CALL MPI_GATHER(array_recv, num_elem, MPI_INTEGER, array_final, &
        num_elem, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

   IF( my_id .eq. 0) THEN
      WRITE(*,*) "N is multiple of num_procs, mod(N,num_procs)= ", &
           mod(N,num_procs)
      WRITE(*,*) " array finale: ", array_final(:)
   END IF

ELSE

   !in case that N is not a multiple of the number of MPI tasks,
   !mpi_scatterv and mpi_gatherv have to be used

   ALLOCATE(sendcount(num_procs),displs(num_procs))

   displs(1) = 0
   sendcount=N/num_procs
   if(0<mod(N,num_procs)) sendcount(1)=N/num_procs+1

   DO i=2,num_procs
      if( (i-1) < mod(N,num_procs) ) sendcount(i) = N/num_procs +1
      displs(i) = SUM(sendcount(1:i-1))
   END DO

   IF (my_id .eq. 0 ) THEN
      WRITE(*,*) "sendcount: ", sendcount
      WRITE(*,*) "displs: ", displs
   END IF

   CALL MPI_SCATTERV(array, sendcount, displs, MPI_INTEGER, &
        array_recv, num_elem, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

   WRITE(*,*) "my_id", my_id, "elementi ricevuti:", array_recv(1:num_elem)

   DO i=1,num_elem
      array_recv(i)= array_recv(i)+my_id
   END DO

   CALL MPI_GATHERV(array_recv, num_elem, MPI_INTEGER, array_final, &
        sendcount, displs, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
      IF( my_id .eq. 0) THEN
         WRITE(*,*) "N is not multiple of num_procs, mod(N,num_procs)= ", &
              mod(N,num_procs)
         WRITE(*,*) " array finale: ", array_final(:)
      END IF
   ENDIF

   DEALLOCATE(array_recv)
   CALL MPI_FINALIZE(ierr)
 END PROGRAM main
```

A.4.4 Solution to exercise 7.2.4

ex4.f90

```
 PROGRAM main
   USE MPI
   IMPLICIT NONE

   INTEGER, parameter :: N=20, SEEDSIZE=50
   INTEGER  :: ierr, i, my_id, num_procs, seed, clock,id
   INTEGER, dimension(N)::array
   INTEGER, dimension(N)::array_final_sum
   INTEGER, dimension(N)::array_final_mult
   REAL, dimension(:), ALLOCATABLE :: array_rands
   REAL  :: r_num, max_value
   INTEGER :: sizer = 1
   INTEGER, dimension(SEEDSIZE) :: seedr

   DOUBLE PRECISION :: t0,t1,time

   CALL MPI_INIT( ierr )
   CALL MPI_COMM_RANK( MPI_COMM_WORLD, my_id, ierr )
   CALL MPI_Comm_size ( MPI_COMM_WORLD, num_procs, ierr )

   t0 = MPI_WTIME()

   DO i=1,N
      array(i)= my_id+1
   END DO

   ! Sum
   CALL MPI_REDUCE(array, array_final_sum, N, MPI_INTEGER, MPI_SUM, 0 ,MPI_COMM_WORLD, ierr)
   IF( my_id .eq. 0) THEN
      WRITE(*,*) " Final array after sum ", array_final_sum(:)
   END IF

   ! Product
   CALL MPI_REDUCE(array, array_final_mult, N, MPI_INTEGER, MPI_PROD,0 ,MPI_COMM_WORLD, ierr)
   IF( my_id .eq. 0) THEN
      WRITE(*,*) " Final array after product: ", array_final_mult(:)
   END IF

   ! Random number generation
   CALL RANDOM_SEED(sizer)

 #ifdef DEBUG
   seedr = my_id+1
 #else
   CALL SYSTEM_CLOCK(COUNT=clock)
   seedr = clock + 37 * (my_id+1) * (/ (i - 1, i = 1, SEEDSIZE) /)
 #endif
```

```
  CALL RANDOM_SEED(put=seedr)
  CALL RANDOM_NUMBER(r_num)

  IF (my_id .eq. 0) ALLOCATE(array_rands(num_procs))
  CALL MPI_GATHER(r_num, 1, MPI_REAL, array_rands, 1, MPI_REAL, 0, MPI_COMM_WORLD, ierr)
  IF (my_id .eq. 0) THEN
     DO id=1,num_procs
        WRITE(*,*) "my_id", id-1, "  Random number :", array_rands(id)
     END DO
  END IF

  ! Search for the maximum value among generated random numbers...
  CALL MPI_REDUCE(r_num, max_value, 1, MPI_REAL, MPI_MAX, 0  ,MPI_COMM_WORLD, ierr)

  t1 = MPI_WTIME()
  time = t1 - t0

  IF( my_id .eq. 0) THEN
     WRITE(*,*) " Maximum generated random number :", max_value
     WRITE(*,*) " Total elapsed time [sec] : ", time
  END IF

  CALL MPI_FINALIZE(ierr)

END PROGRAM main
```

A.4.5 Solution to exercise 7.2.5

ex5.f90

```
PROGRAM heat2D
  USE mpi
  IMPLICIT none

  INTEGER :: procs, rank, error
  INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status

  INTEGER :: lado, i, j, t=0
  INTEGER :: my_x, my_y, my_si, my_sj, my_ei, my_ej, my_lado, contact
  REAL :: my_min, my_max, my_diff

  REAL :: min,max,diff
  REAL, DIMENSION(:,:), ALLOCATABLE :: data
  REAL, DIMENSION(:), ALLOCATABLE :: data_temp

  CALL MPI_Init ( error )
  CALL MPI_Comm_size ( MPI_COMM_WORLD, procs, error )
  CALL MPI_Comm_rank ( MPI_COMM_WORLD, rank, error )

  IF (rank .EQ. 0) THEN
     READ*, lado
     ALLOCATE(data(0:lado+1,0:lado+1))

     data = 0.0

     DO i=1,lado
        READ*, data(i,1:lado)
     END DO

     CALL MPI_BCAST(lado,1,MPI_INTEGER,0,MPI_COMM_WORLD,error)
     CALL MPI_BCAST(data,(lado+2)*(lado+2),MPI_REAL,0,MPI_COMM_WORLD,error)
```

```
ELSE
   CALL MPI_BCAST(lado,1,MPI_INTEGER,0,MPI_COMM_WORLD,error)
   ALLOCATE(data(0:lado+1,0:lado+1))
   CALL MPI_BCAST(data,(lado+2)*(lado+2),MPI_REAL,0,MPI_COMM_WORLD,error)
END IF

my_lado = lado / 2
ALLOCATE(data_temp(my_lado))

my_x = (rank / 2)
my_y = MOD(rank,2)

my_si = my_x*my_lado + 1
my_sj = my_y*my_lado + 1

my_ei = my_si + my_lado - 1
my_ej = my_sj + my_lado - 1

my_min=MINVAL(data(my_si:my_ei,my_sj:my_ej))
my_max=MAXVAL(data(my_si:my_ei,my_sj:my_ej))

CALL MPI_ALLREDUCE(my_min,min,1,MPI_REAL,MPI_MIN,MPI_COMM_WORLD,error)
CALL MPI_ALLREDUCE(my_max,max,1,MPI_REAL,MPI_MAX,MPI_COMM_WORLD,error)

diff = max - min
IF (rank .EQ. 0)   PRINT*, "t = ",t, "diff = ",diff

DO WHILE (diff .GE. 1)

   t=t+1
   data(my_si:my_ei,my_sj:my_ej) = 0.99*data(my_si:my_ei,my_sj:my_ej) + 0.01*((data(my_si-1:my_ei-1,
        data(my_si+1:my_ei+1,my_sj:my_ej) + data(my_si:my_ei,my_sj-1:my_ej-1) + data(my_si:my_ei,my_


   !! Envio de datos

   !! Envio de filas (TAG = 10)
   !! Si my_x = 0, tengo que enviar mi ultima fila a my_y + 2
   !! Si my_x <> 0, tengo que enviar mi primera fila a my_y

   IF (my_x .EQ. 0) THEN
      contact = my_y + 2
      data_temp = data(my_ei,my_sj:my_ej)
      CALL MPI_Send(data_temp,my_lado,MPI_REAL,contact,10,MPI_COMM_WORLD,error)
      CALL MPI_Recv(data_temp,my_lado,MPI_REAL,contact,10,MPI_COMM_WORLD,status,error)
      data(my_ei+1,my_sj:my_ej) = data_temp
   ELSE
      contact = my_y
      CALL MPI_Recv(data_temp,my_lado,MPI_REAL,contact,10,MPI_COMM_WORLD,status,error)
      data(my_si-1,my_sj:my_ej) = data_temp
      data_temp = data(my_si,my_sj:my_ej)
      CALL MPI_Send(data_temp,my_lado,MPI_REAL,contact,10,MPI_COMM_WORLD,error)
   END IF

   !! Envio de columnas (TAG = 20)
   !! Si my_y = 0, tengo que enviar mi ultima columna a my_x*2 + 1
   !! Si my_y <> 0, tengo que enviar mi primera columna a my_x*2

   IF (my_y .EQ. 0) THEN
      contact = (my_x * 2) + 1
      data_temp = data(my_si:my_ei,my_ej)
      CALL MPI_Send(data_temp,my_lado,MPI_REAL,contact,20,MPI_COMM_WORLD,error)
      CALL MPI_Recv(data_temp,my_lado,MPI_REAL,contact,20,MPI_COMM_WORLD,status,error)
      data(my_si:my_ei,my_ej+1) = data_temp
```

```
     ELSE
        contact = my_x * 2
        CALL MPI_Recv(data_temp,my_lado,MPI_REAL,contact,20,MPI_COMM_WORLD,status,error)
        data(my_si:my_ei,my_sj-1) = data_temp
        data_temp = data(my_si:my_ei,my_sj)
        CALL MPI_Send(data_temp,my_lado,MPI_REAL,contact,20,MPI_COMM_WORLD,error)
     END IF

     my_min=MINVAL(data(my_si:my_ei,my_sj:my_ej))
     my_max=MAXVAL(data(my_si:my_ei,my_sj:my_ej))

     CALL MPI_ALLREDUCE(my_min,min,1,MPI_REAL,MPI_MIN,MPI_COMM_WORLD,error)
     CALL MPI_ALLREDUCE(my_max,max,1,MPI_REAL,MPI_MAX,MPI_COMM_WORLD,error)

     diff = max - min
     IF (rank .EQ. 0)   THEN
        IF (MOD(t,1000) .EQ. 0) PRINT*, "t = ",t, "diff = ",diff
     END IF

  END DO

  IF (rank .EQ. 0) THEN
     PRINT*, "Final t is: ", t
  END IF

  CALL MPI_Finalize ( error )
END PROGRAM heat2D
```

Appendix B

Barnes-Hut Algorithm

B.1 Barnes-Hut original paper in Nature

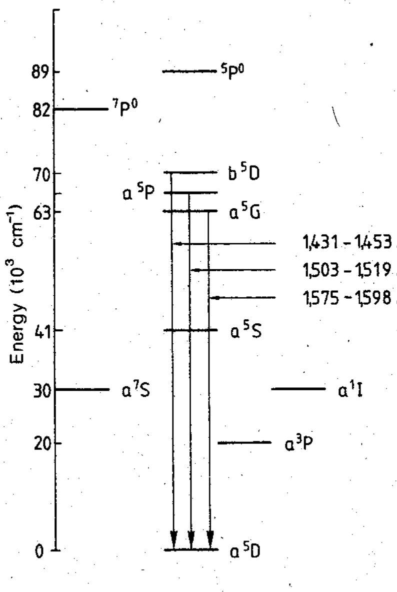


Fig. 2 A partial term scheme for Fe III, showing the observed transitions, some lower-lying terms of the same even parity and the first two terms of odd parity. Term energies are given in cm^{-1}.

abilities found by Garstang[8] for lower-lying levels are in each case ~1. Thus it is likely that the upper levels have populations close to their Boltzmann values with collisional de-excitation rates exceeding radiative decay rates. Then the ratio of the [Fe III] line fluxes to those of permitted transitions would be sensitive to the electron density. Detailed calculations of the atomic data are required to establish the collisional-radiative regime for the individual lines. The upper levels may attain only a pseudo-Boltzmann population if collisional excitation to higher states of odd parity exceeds the rate for collisional de-excitation to lower levels. In any case the new [Fe III] identifications will provide more information on the structure of the solar chromosphere-corona transition region.

If the a^5G, a^5P and b^5D levels are collisionally de-excited in the solar atmosphere, they could become stronger, relative to permitted transitions of species of similar excitation, in astrophysical sources of lower electron density. For this reason their presence is being investigated in such sources, including the Seyfert galaxy NGC 4151, which has unidentified emission features around 1,575, 1,581 and 1,518 Å (refs 13 and 14). [Fe III] emission is observed in the optical spectrum of NGC 4151[15,16]. The relative intensities of the quintet transitions in NGC 4151 and other sources cannot be predicted until collision cross-sections and transition probabilities are known, and these are urgently required to establish whether or not the [Fe III] lines are of wider astrophysical significance.

Received 3 September; accepted 23 October 1986.

1. Bartoe, J.-D. F., Brueckner, G. E., Purcell, J. D. & Tousey, R. *Appl. Opt.* **16**, 879-885 (1977).
2. Sandlin, G. D., Bartoe, J.-D. F., Brueckner, G. E., Tousey, R. & VanHoosier, M. E. *Astrophys. J. suppl. Ser.* **61**, 801-898 (1986).
3. Jordan, C., Brueckner, G. E., Bartoe, J.-D. F., Sandlin, G. D. & VanHoosier, M. E. *Astrophys. J.* **226**, 687-697 (1978).
4. Bartoe, J.-D. F. *et al.* in *AIAA 24th Aeropsace Sciences Meeting, Reno*, AIAA-86-0225 (in the press).
5. Moore, C. E. in *Selected Tables of Atomic Spectra: Atomic Energy Levels and Multiplet Tables, Si L* NSRDS-NBS-3, Sect. 2 (1967).
6. Edlén, B. & Swings, P. *Astrophys. J.* **95**, 532-554 (1942).
7. Reader, J. & Sugar, J. *J. phys. chem. Ref. Data* **4**, 353-440 (1975).
8. Garstang, R. H. *Mon. Not. R. astr. Soc.* **117**, 393-405 (1957).
9. Garstang, R. H., Robb, W. D. & Rountree, S. P. *Astrophys. J.* **222**, 384-397 (1978).
10. Abbott, D. C. *J. Phys.* **B11**, 3479-3497 (1978).
11. House, L. L. *Astrophys. J. suppl. Ser.* **8**, 307-328 (1964).
12. Munro, R. H. & Withbroe, G. L. *Astrophys. J.* **176**, 511-520 (1972).
13. Ulrich, M. H. *et al. Nature* **313**, 747-751 (1985).
14. Cassatella, A. & Jordan, C. (in preparation).
15. Boksenberg, A. *et al. Mon. Not. R. astr. Soc.* **173**, 381-386 (1975).
16. Boksenberg, A. & Penston, M. V. *Mon. Not. R. astr. Soc.* **177**, 127P-131P (1976).

A hierarchical $O(N \log N)$ force-calculation algorithm

Josh Barnes & Piet Hut

The Institute for Advanced Study, School of Natural Sciences, Princeton, New Jersey 08540, USA

Until recently the gravitational N-body problem has been modelled numerically either by direct integration, in which the computation needed increases as N^2, or by an iterative potential method in which the number of operations grows as $N \log N$. Here we describe a novel method of directly calculating the force on N bodies that grows only as $N \log N$. The technique uses a tree-structured hierarchical subdivision of space into cubic cells, each of which is recursively divided into eight subcells whenever more than one particle is found to occupy the same cell. This tree is constructed anew at every time step, avoiding ambiguity and tangling. Advantages over potential-solving codes are: accurate local interactions; freedom from geometrical assumptions and restrictions; and applicability to a wide class of systems, including (proto-)planetary, stellar, galactic and cosmological ones. Advantages over previous hierarchical tree-codes include simplicity and the possibility of rigorous analysis of error. Although we concentrate here on stellar dynamical applications, our techniques of efficiently handling a large number of long-range interactions and concentrating computational effort where most needed have potential applications in other areas of astrophysics as well.

Until recently, the dynamics of a system of self-gravitating bodies (the gravitational N-body problem) has been modelled numerically in two fundamentally different ways. The first one, direct N-body integration, involves the computation of all $\frac{1}{2}N(N-1)$ forces between all pairs of particles. This allows an accurate description of the dynamical evolution but at a price that grows rapidly for increasing N[1]. The second way involves a two-step approach: after fitting the global potential field to a special model with a number of free parameters, each particle is propagated in this background field for a short time before the same procedure is reiterated. The potential method involves a number of operations that grow only as $N \log N$. Thus calculations can be performed more quickly, but with a loss of accuracy and generality. The special nature of each potential-solving code is caused by the need to use some technique that is tuned to the geometry of the problem being considered (such as Fourier transforms or spherical or bispherical harmonics[2]).

Recently, some of the advantages of both approaches have been combined by using direct integrations of force while grouping together increasingly large groups of particles at increasingly large distances. This corresponds to the way humans interact with neighbouring individuals, further villages and increasingly further and larger states and countries—driven by increasing cost and decreasing need to deal with more removed groups on an individual basis. The first implementation of such a hierarchical grouping of interactions was given by Appel[3], who used a tree structure to represent an N-body system, with the particles stored in the leaves of the tree. An independent implementation by Jernigan[4] and Porter[5] incorporated regularization of close encounters. However, in both codes the logarithmic-growth gain in efficiency comes at the price of introducing additional errors that are hard to analyse because of the arbitrary structure of the tree. Nearby particles may be grouped as leaves of nearby branches, but the phase-space flow of realistic self-gravitating systems demands a continuous updating of the tree structure to avoid tangling and unphysical grouping, requiring complicated book-keeping. It is not at all clear how to understand and estimate the errors caused by the process of approximating lumps of particles together as single pseudo-particles, because individual lumps can take more or less arbitrary shapes and sizes.

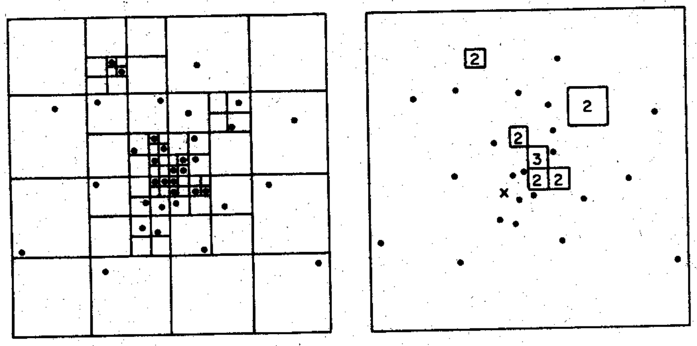


Fig. 1 Hierarchical boxing and force calculation, presented for simplicity in two dimensions. On the left, a system of particles and the recursive subdivision of system space induced by these particles. Our algorithm makes the minimum number of subdivisions necessary to isolate each particle. On the right, how the force on particle x is calculated. Fitted cells contain particles that have been lumped together by our 'opening angle' criterion; each such cell represents a single term in the force summation.

We present here a new way of realizing a tree-based force calculation with logarithmic growth of force terms per particle that avoids the tree-tangling complications mentioned above, allows rigorous upper bounds for errors that arise from neglecting internal lump structure, and also offers a well-defined procedure for estimating more typical, average errors. The essential ingredients are (1) a virtual cubical division of empty space in (sub)cells with daughter cells having exactly half the length, breadth and width of their parent; (2) the construction of the actual tree of cells from the virtual one by (i) discarding empty subcells, (ii) accepting subcells with one occupant, and (iii) recursively dividing shared occupancies in sub-subcells; and (3) performing this reconstruction *ab initio* at every time step.

Given this book-keeping structure, the dynamics are implemented by assigning to every non-empty cell, as well as to higher-order cells containing more than one particle, a (pseudo-)particle that contains the total mass in the cell located at the centre-of-mass of all the particles it contains. Any single real particle feels the force of all (pseudo-)particles in the system that represent a cell small enough and far enough to forego the need of further division, thereby screening all its component (pseudo-)particles.

A computer program that implements the hierarchical force calculation is available from us upon request. It contains less than a thousand lines of C code: 150 lines of definitions, 150 lines for tree construction, 100 lines for force calculation and 100 lines for a simple integrator; the remaining lines handle input-output book-keeping.

In what follows we summarize some of the more technical details. The method we use to compute a force in time of $O(\log N)$ is based on a representation of the mass distribution as a hierarchical tree structure, constructed as follows. Begin with an empty cubical cell big enough to contain the system. One by one, load particles into this 'root' cell. If any two particles fall into the same cell, divide that cell into eight cubical subcells (thus the first such division occurs as soon as the second particle has been loaded in, splitting the system into at least eight pieces). Each divided cell is represented by a data structure that holds information about the subcells it contains: a summary of global physical quantities (mass and centre-of-mass position) as well as pointers to the daughter cells, which may be referenced to obtain more detailed information. Continue this process of subdividing to as high a level as required. When all N particles have been loaded, the system space will have been partitioned up into a number of cubical cells of different sizes, with at most one particle per cell. These particle-bearing cells are grouped together into larger cubical cells, which are grouped together into still larger parent cells, and so on down to the root cell, which contains the entire system. The average size of a particle-bearing cell is of the order of the interparticle spacing, so the 'height of the tree' (that is, the number of subdivisions required to reach a typical cell, starting at the root), is of $O(\log_2 N^{1/3}) = O(\log N)$, and the time required to construct the tree is of $O(N \log N)$. The final step in constructing the tree is to tag the subdivided cells with the total mass and centre-of-mass position of the particles they contain; by propagating information down the tree from the particles towards the root, this step may also be accomplished in a time of $O(N \log N)$.

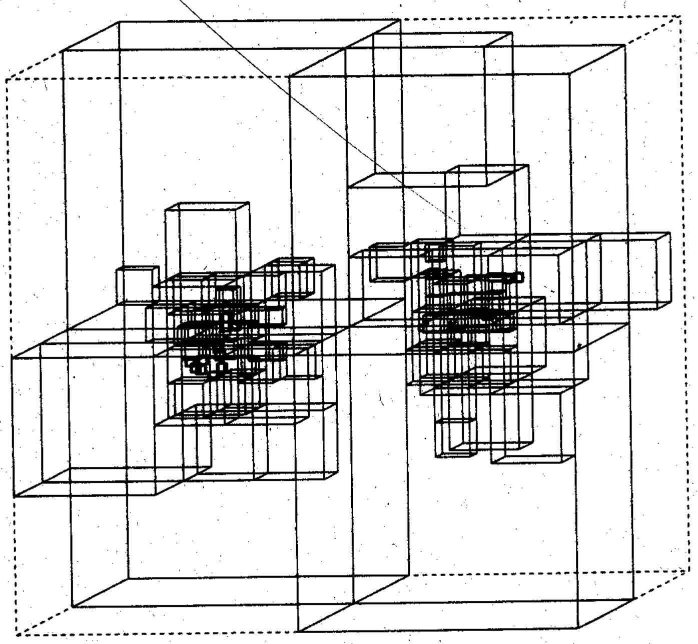


Fig. 2 Box structure induced by a three-dimensional particle distribution. This example was taken from the early stages of an encounter of two $N = 64$ systems, and shows how the boxing algorithm can accommodate systems with arbitrarily complicated geometry. The particle distribution corresponding to a system with 32 times as many members is shown in Fig. 3.

Having constructed such a tree, the force on any particle p may be approximated by a simple recursive calculation. Start at the root cell of the tree, which contains the entire system. Let l be the length of the cell currently being processed and D the distance from the cell's centre-of-mass to p. If $l/D < \theta$, where θ is a fixed accuracy parameter ~ 1, then include the interaction between this cell and p in the total being accumulated. Otherwise, resolve the current cell into its eight subcells, and recursively examine each one in turn. The core of the force calculation routine may be compactly expressed in SCHEME, a dialect of LISP:

```
(define (acceleration particle ensemble)
  (cond ((singleton? ensemble)
         (newton-acceleration particle (the-element ensemble)))
        ((< (/ (diameter ensemble)
               (distance particle (centroid ensemble)))
            theta)
         (newton-acceleration particle (centroid ensemble)))
        (else
         (reduce sum-vector
                 (map (lambda (e) (acceleration particle e))
                      (subdivisions ensemble))))))
```

Note that in LISP, a function with arguments $f(x, y, \ldots)$ is written as $(f\ x\ y \ldots)$. For example, (newton-acceleration p_1 p_2) calls a function to compute the acceleration of particle p_1 due to p_2. The (cond . . .) form is a three-way conditional, computing the acceleration directly in the first two cases, and by recursion in the final (else . . .) clause. Elements of the SCHEME programming language are presented in Abelson *et al.*[6]. Figure 1

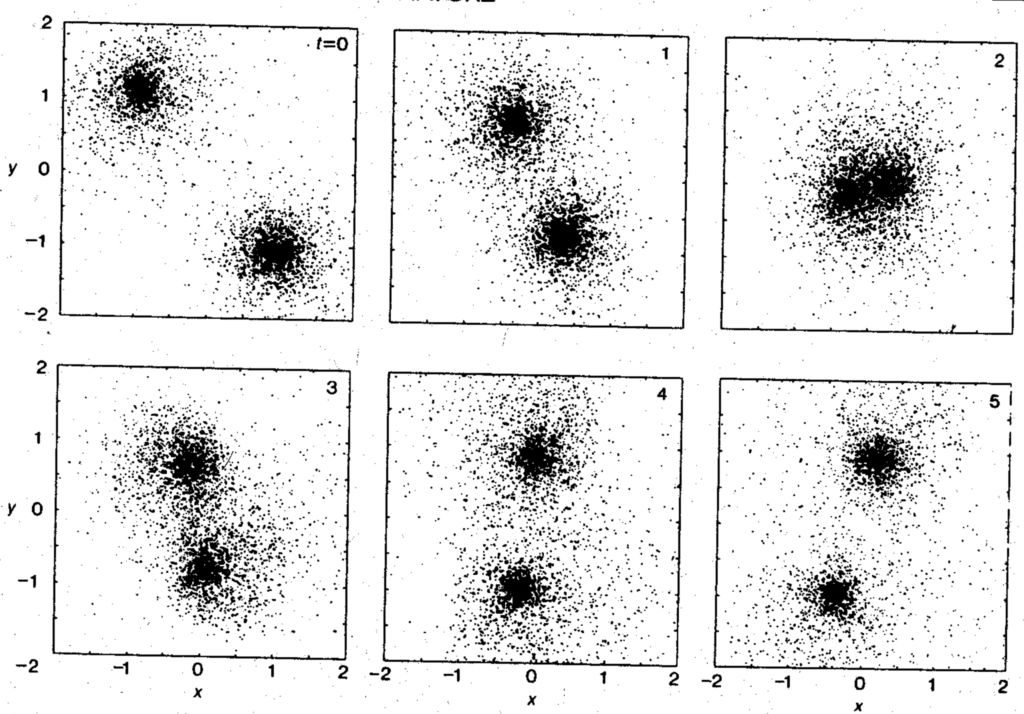


Fig. 3 Encounter of two spherical systems, simulated by using our hierarchical acceleration technique in combination with a simple leap-frog integrator. The incoming systems were launched on parabolic orbits, but become bound because of dynamical friction. Note the striking wakes lagging behind the density centres of the two systems at $t = 3$, 4 (in our units the gravitational constant, the mass of each galaxy and the total binding energy of the whole system all equal unity). With a total of $N = 4{,}096$ particles in the system and an opening angle criterion of $\theta = 1$, the number of two-body interactions computed by our technique is less than 0.1 of the $\frac{1}{2}N/(N-1)$ required by a direct-summation force calculation. The calculation took 10 h on a VAX 11/780 (in double precision because of compiler limitations; a single-precision calculation would take half as long). With a time step $\Delta t = 0.05$ and softening parameter $\varepsilon = 0.025$, energy was conserved to $\sim 1\%$.

illustrates this process for a small number of particles in two dimensions; increasing the number to $10^4 \sim 10^5$ in three dimensions typically increases the number of interactions per particle to only of order 10^2.

The number of interactions considered by this procedure in computing the force on p is of order log N for large N. Suppose the mass distribution is homogeneous within the root cell. Increasing the total number of particles eightfold is roughly equivalent to adjoining eight similar root cells together. The seven new cells not containing p will contribute some 'relatively small' number ΔN_t of additional terms to the force approximation. Now the expectation value $\langle \Delta N_t \rangle$ depends on θ, but not on the total number of particles or the size of the system. Thus the time required to calculate the force on a particle increases by a constant increment (of $\langle \Delta N_t \rangle$), whereas N increases by a constant factor (of eight). In other words, the time required by the CPU (central processing unit) to compute the force on a single particle is on the scale of $O(\log N)$.

A rigorous error analysis of the force-calculation algorithm is possible because our prescription yields a unique, well-characterized tree structure based on up-to-date particle positions. Each compound cell that we choose not to subdivide introduces a small error due to quadrupole and higher-order moments of the mass distribution within the cell (the dipole term vanishes when expanding around the centroid). The magnitude of this error may be bounded by a 'worst-ease' analysis for which the quadrupole moment is maximized (for example, two lumps placed in opposite corners of the cell), and estimated from an analysis of root-mean-square fluctuations within each cell together with estimates of the coherence time scales for these fluctuations. We shall present this analysis in a more detailed paper. In practice, forces computed even with an opening angle parameter as large as $\theta = 1$ are still accurate to $\sim 1\%$ with little dependence on N. Empirically, we find the force error scales approximately as the -1.5 power of the computing time. These errors are only weakly correlated from one time step to the next, resulting in a build-up close to a random walk rather than a steady drift.

As a test of our new method, we have written a simple N-body code using our force-calculation scheme with a time-centred leap-frog integrator, in which positions and velocities are alternately advanced. A parabolic encounter of two galaxies is initiated at a distance of several galactic radii, leading to a box structure as shown in Fig. 2. The results of a 4096-body calculation of such an encounter are shown in Fig. 3. This calculation took 10 h of CPU time on a VAX 11/780 with a floating point accelerator.

There are several ways in which the code can be made more efficient. We are now investigating these, and we shall discuss our results in detail elsewhere. We just mention three possible improvements: (1) using a higher-order integration scheme such as Aarseth's fourth-order polynomial method rather than our second-order leap-frog method, which will require careful adjustments to avoid glitches caused by discrete differences between the grouping of particles in cells from one time step to the next (for example by multiply covering space in partly overlapping virtual grids); (2) including quadrupole moments in the description of cells as pseudoparticles characterized by the total mass in the cell as located in the centre of mass; (3) introducing individual time steps for particles which undergo strongly changing interactions, which could be accomplished by subsequently halving the time step when needed—thus extending the three-dimensional spatial halving of cells to a four-dimensional space-time division in rectangular subcells.

An interesting aspect of our new code is the different emphasis it places both on software and hardware, in comparison with other codes. On the hardware side, the hierarchical structure of our code does not lend itself easily to vectorization (although this may well be worth exploring). In contrast, we expect our code to be most useful on computers with highly parallel architectures (with one processor per particle, computer time is reduced approximately by a factor of N). On the software side, the hierarchical decomposition of the problem is best realized by using recursive descriptions. Recursive function calls and other general control and data structures are not well supported or clearly represented in FORTRAN. This has led us to consider other programming languages such as C, PASCAL and LISP. Another advantage offered by these languages is that they permit a clarity of presentation of our ideas, which makes the underlying techniques available to other researchers. Of course, if a particular computer has a FORTRAN compiler which is an order of magnitude faster than other compilers, it makes sense to translate a version of our program into FORTRAN, trading clarity and modularity for efficiency.

Our application to *N*-body calculations is only one in a range of possibilities including the calculation of radiation fields (replacing particles with sources) and self-gravitating fluid flow (cell division being governed by the complexity of the local flow pattern). Thus our technique forms a general tool for simultaneously handling a large number of long-range interactions and for concentrating computing resources locally where most needed.

We thank John Bahcall, Jeremiah Ostriker and especially Gerald Sussman for interesting discussions. Part of this work was supported by the NSF through grant PHY-8217352; P.H. is an Alfred P. Sloan Foundation fellow.

Received 16 July; accepted 1 October 1986.

1. Aarseth, S. J. in *Multiple Time Scales* (ed. Brackbill, J. U. & Cohen, B. I.) 377-418 (Academic Press, New York, 1985).
2. Hockney, R. W. & Eastwood, J. W. *Computer Simulation using Particles* (McGraw-Hill, New York, 1981).
3. Appel, A. *SIAM J. Sci. statist. Comput.* **6**, 85-103 (1985).
4. Jernigan, J. G. *I.A.U. Symp.* **113**, 275-284.
5. Porter, D. thesis, Physics Dept, Univ. California, Berkeley (1985).
6. Abelson, H., Sussman, G. J. & Sussman, J. *Structure and Interpretation of Computer Programs* (MIT Press, Cambridge, Massachusetts, 1985).

The 400-km seismic discontinuity and the proportion of olivine in the Earth's upper mantle

Craig R. Bina & Bernard J. Wood

Department of Geological Sciences, Northwestern University, Evanston, Illinois 60201, USA

The 400-km seismic discontinuity has traditionally been ascribed to the isochemical transformation of α-olivine to the β-modified-spinel structure in a mantle of peridotitic bulk composition[1-6]. It has recently been proposed[7,8] that the observed seismic velocity increase at 400 km depth is too abrupt and too small to result from a phase change in olivine but instead requires that the transition zone be chemically distinct in bulk composition from the uppermost mantle. By requiring phase relations in the Mg_2SiO_4-Fe_2SiO_4 system to be internally consistent thermodynamically, we find that the α–β transition in olivine of mantle $(Mg_{0.9}Fe_{0.1})_2SiO_4$ composition is extremely sharp, occurring over a depth interval (isothermal) of ~6 km. The magnitude of the predicted velocity increase is in agreement with that observed seismically[9,10] if the transition zone is composed of ~60–70% olivine. Thus, our results indicate that seismic velocities across the 400-km discontinuity are consistent with a transition zone of homogeneous peridotitic composition and do not require chemical stratification.

The 400-km seismic discontinuity reflects a change in elastic properties of the mantle and has been attributed to a phase transformation of olivine to a spinel-like structure at high pressures[1]. Subsequent work has given rise to a generally accepted model in which the discontinuity is attributed to such an isochemical phase change in a mantle of homogeneous olivine-rich, or peridotitic, composition[2-6]. This model has the advantage of simplicity and can be tested experimentally.

Recently, it has been suggested[7,8] that a phase transition in olivine would produce a gradual velocity increase over an appreciable depth interval—rather than the abrupt increase observed seismically—and that the magnitude of the increase would be more than twice that actually observed. It was proposed that the seismic data require the transition zone to be chemically distinct in bulk composition from the uppermost mantle, with the transition zone consisting of a pyroxene–garnet rich 'piclogite' composition containing either 16%[7] or 30%[8] olivine. The 400-km discontinuity is ascribed to either a change in chemical composition from peridotite to underlying piclogite or—if this periodotite/piclogite boundary is referred to shallower depths—to the transformation of pyroxene to a garnet-like structure.

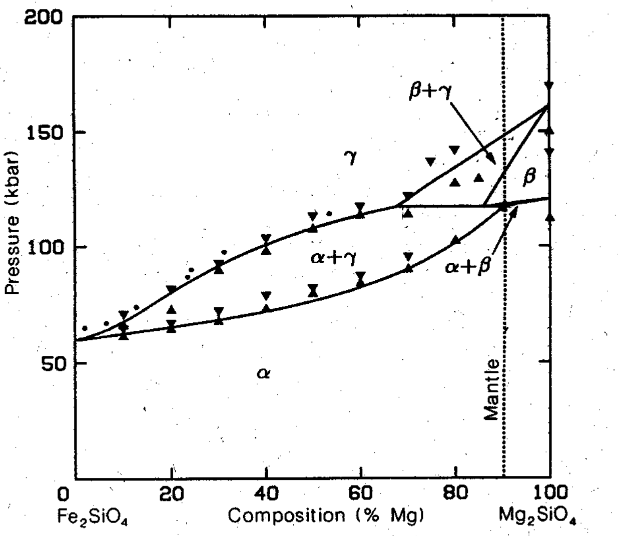


Fig. 1 Isothermal pressure–composition diagram showing calculated boundaries for olivine polymorph stability fields at 1,273 K. Also shown are experimental data points[12-14,18-20] delimiting the high-pressure stability limits of the low-pressure assemblages (▲), the low-pressure stability limits of the high-pressure assemblages (▼), and the compositions of γ phase which coexist with α phase at the indicated pressures (●). Dashed line shows $(Mg_{0.9}Fe_{0.1})_2SiO_4$ composition.

In a previous study[11], we showed that the transformation of pyroxene to a garnet structure would produce a smooth and gradual increase in seismic velocity, rather than the discontinuity observed at 400 km. In the present study, we have examined the olivine–spinel phase transitions to determine whether the observed seismic velocity variations may be attributable to such a phase change. The α-olivine to β-modified-spinel transition has been commonly represented by a broad '$\alpha+\beta$ divariant loop', a region in which both phases coexist in stable equilibrium. If this representation were accurate, the α phase would transform to the β phase in a continuous and gradual manner, and this phase change would not produce a sharp discontinuity in seismic velocity. However, the available experimental data (Fig. 1) do not constrain the width of this $\alpha+\beta$ loop, since no high-pressure experiments have yet produced both phases together in equilibrium for olivine of mantle $(Mg_{0.9}Fe_{0.1})_2SiO_4$ composition. We have used available thermoelastic and calorimetric data on the olivine polymorphs (α-olivine, β-modified-spinel, and γ-spinel) to constrain the width of the $\alpha+\beta$ divariant loop. By requiring the phase diagram for the Mg_2SiO_4-Fe_2SiO_4 system to be internally consistent thermodynamically, we have attempted to determine the sharpness and magnitude of a seismic discontinuity resulting from a phase change in olivine.

If the partial molar free energies of Mg_2SiO_4 and Fe_2SiO_4 components are known as functions of pressure, temperature, and composition, then the boundaries of the stability fields for the various phase assemblages (α, $\alpha+\beta$, β, $\beta+\gamma$ and so on) can be calculated explicitly. To compute the free-energy functions, we require knowledge of the enthalpies, entropies, volumes, and solution activities of the components in the various phases at the pressures, temperatures and compositions of interest. We used the available experimentally-measured values of the enthalpies and entropies[5,12-14], heat capacities[15], molar volumes and coefficients of thermal expansion[4], elastic moduli[16], and activity coefficients[17] for the phases and components in question. Where measured values were extremely uncertain or

PART V

PROJECTS

Appendix C

Class projects

C.1 Project A - Genealogy project (pointers and recursion)

The goal of this project is to create an efficient genealogy tree. As input your program will read a list like the following:

```
1 4 5
1 4 5
0 1 6
0 1 7
2 1 3
0 1 2
0 1 3
0 1 4
0 5 3
0 5 6
1 2 1
0 4 4
-1 0 0
```

Each line represents a relation parent/child between to people, with three items:

```
relation id_p1 id_p2
```

relation can be: either 0 [child] or 1 [parent]. *id_p1* and *id_p2* are IDs of the two people involved. Thus for example: *1 4 5* means that 5 is a parent of 4, while *0 1 6* means that 6 is a child of 1.

The code should ignore: duplicated relations and relations with wrong number (i.e. not 0 or 1). At the same time, your code should have for each node a maximum of 4 parents and 4 children, so we should also skip if we try to add more than 4 parents or 4 children for a node.

A *relation* of -1 is used to identify the end of the input.

Thus, the relations given above represent a genealogy tree as can be seen in figure C.1. Note that 4 is considered to be a child of itself! and that the relation "0 1 7" has been dropped because node 1 already has 4 children.

Your goal is to create a Binary Search Tree where the nodes will be sorted based on their ID, but at the same time, each node will include links to its parents and its children (not the IDs of its children and parents, but pointers to the appropriate nodes in the tree). Thus, the BST would look like figure C.2 (assuming you add nodes to the tree as they appear in the input file above), with the arrow representing the usual left/right nodes, but on top of that, you should make sure that each node maintains links to its children and parents.

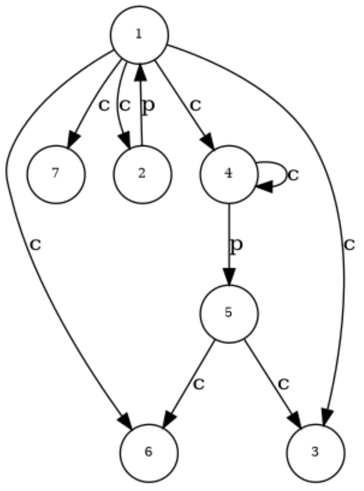


Figure C.1: Relations graph - ’c’ (child); ’p’ (parent)

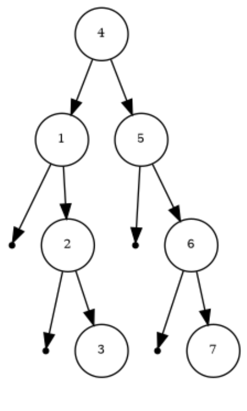


Figure C.2: BST

The graph with all the arrow pointing left/right, with all the children and parents become messy quite quickly, but we can see an example of how it would look like in figure C.3, which is only a restricted view, with the relations for nodes 4,1 and 5. We see that the left pointer of 4 points to 1 and the right pointer to 5, thus keeping the structure of the BST as shown above in figure C.2. At the same time, the first child (c1) of node 4 points to itself (node 4), while the first parent (p1) of node 4 points to node 5, etc. So you can see that what you are supposed to do is just build the BST tree according to the ID of each person, but then make sure that you link the children and parents pointers according to the relations in the input file.

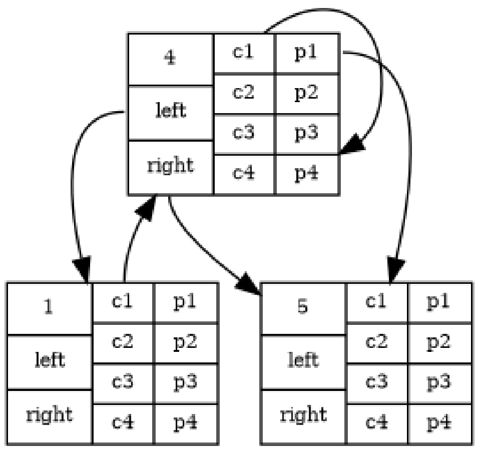


Figure C.3: BST with full connections, restricted to nodes 4,1 and 5

Remember that in Fortran you cannot have an array of pointers, but this is easily done by having a derived type which only component is a pointer and then having an array of that new type. To make this easier for you, the type definitions needed for each of the nodes could be something like:

```
TYPE CPtr
   TYPE(PERSON), POINTER :: ptr
END type CPtr

TYPE PERSON
   INTEGER :: id
   TYPE(PERSON),POINTER :: left,right
   TYPE(CPtr), DIMENSION(4) :: parents
   TYPE(CPtr), DIMENSION(4) :: children
END type PERSON
```

When you write the code, make sure that it does two things:

1. Create the tree, printing when a new node is created, and giving warnings when: there is no space to hold more relations in a node; a relation has already been created; the relation type is wrong
2. Print the tree, following the way we printed a BST tree in the lectures, from the smallest ID to the biggest ID, printing for each node its parents and children.

Remember that the project is not all or nothing, and your mark will depend on how far you get, but you can submit partial code, non-working code (if possible with comments) and even a text explanation of things that you tried to do but don't work, etc.

Try to start simple and build up the code when you are sure the previous step works. For example, try first to generate the BST tree for the people ID and verify that it is well formed, but ignoring the relations. Once that is properly done you can start adding relations but ignoring the warnings (thus, assuming that the input file will never have, for example, repeated relations), and at the end adding the code to check the warning situations. Keep copies of the code as you get partial solutions, so that you can at least submit something that works in the case that the next steps become difficult and you cannot solve them. Also, remember that this is an INDIVIDUAL project. You can discuss generic ideas, but the final code has to be yours.

Lastly, as an example of what your code should produce when giving the above input file, it could look like this:

```
[angelv]$ ./gen < gen.in

 -----------------------
 CREATING GENEALOGY TREE
 -----------------------
 creating new node           4
 creating new node           5
 ... WARNING: relation           1           4           5 already entered, skipping
 creating new node           1
 creating new node           6
 creating new node           7
 creating new node           3
 ... WARNING: wrong relation value           2 , skipping
 creating new node           2
 ... WARNING: no empty places for           0           1           4 , skipping

 -----------------------
 PRINTING GENEALOGY TREE
 -----------------------

 ------------ node            1 --------------
 PARENTS
 CHILDREN
           6
           7
           2
           3

 ------------ node            2 --------------
 PARENTS
           1
 CHILDREN

 ------------ node            3 --------------
 PARENTS
 CHILDREN

 ------------ node            4 --------------
 PARENTS
           5
 CHILDREN
           4

 ------------ node            5 --------------
 PARENTS
 CHILDREN
           3
           6

 ------------ node            6 --------------
 PARENTS
 CHILDREN
 ------------ node            7 --------------
 PARENTS
 CHILDREN

[angelv]$
```

C.2 Project B - RT-like (MPI)

The goal of this project is to parallelize a serial code, inspired in a radiative transfer code.

The serial code is given below. As you can see, the only input to the program are the number of points in the X and Y dimensions (*nx, ny*) used to allocate array with the given sizes. This is initialized (*init_atmos()*) with some arbitrary data, and then the code calls *rt_l*), which performs what you can graphically see in figure C.4: we simulate 45^o rays in a non-periodic domain, so we *follow* the rays starting in the first column and in the last row throughout the domain, applying a simple expression (see procedure *propagate()*) to recalculate the values in the array. Lastly, with the proocedure *print_atmos()* we can compare the initial with the final values.

```
PROGRAM RT_LIKE
  IMPLICIT NONE

  DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE :: atmos
  INTEGER :: nx,ny

  READ*, nx,ny
  ALLOCATE(atmos(nx,ny))

  !! Initialize array with some sample data
  CALL init_atmos()

  PRINT*, " ------------ Initial values -----------------"
  CALL print_atmos()

  PRINT*, ""
  CALL rt_l()
  PRINT*, ""

  PRINT*, " ------------ Final values -----------------"
  CALL print_atmos()

CONTAINS

  ! ---------------------------------------------
  ! rt_l
  !
  ! This routine is fixed for 45° angles
  ! ---------------------------------------------
  SUBROUTINE rt_l()
    INTEGER :: st_row,st_col,uw_row,uw_col,dw_row,dw_col

    ! Rays starting in column 1
    DO st_row=1,nx
       uw_row = st_row
       uw_col = 1
!       PRINT*, "Doing ray starting at:", uw_row,uw_col
       CALL propagate(uw_row,uw_col)
    END DO

    ! Rays starting in row nx
    DO st_col=2,ny
       uw_row = nx
       uw_col = st_col
!       PRINT*, "Doing ray starting at;", uw_row,uw_col
       CALL propagate(uw_row,uw_col)
    END DO
  END SUBROUTINE rt_l

  ! ---------------------------------------------
  ! propagate
```

```
    !
    ! --------------------------------------------
    SUBROUTINE propagate(uw_row,uw_col)
      INTEGER :: uw_row,uw_col,dw_row,dw_col

      DO
         dw_row = uw_row - 1
         dw_col = uw_col + 1
         IF (dw_row < 1 .OR. dw_col > ny) EXIT
         atmos(dw_row,dw_col) = (atmos(uw_row,uw_col) + atmos(dw_row,dw_col)) * 0.5
!         PRINT*, "pos",dw_row,dw_col,atmos(uw_row,uw_col),atmos(dw_row,dw_col)
         uw_row = dw_row
         uw_col = dw_col
      END DO
    END SUBROUTINE propagate

    ! --------------------------------------------
    ! init_atmos
    !
    ! --------------------------------------------
    SUBROUTINE init_atmos()
      INTEGER :: x,y

      DO x=1,nx
         DO y=1,ny
            atmos(x,y) = 1.234 * x + 2.345 * y
         END DO
      END DO
    END SUBROUTINE init_atmos

    ! --------------------------------------------
    ! print_atmos
    !
    ! --------------------------------------------
    SUBROUTINE print_atmos()
      INTEGER :: x
      CHARACTER(LEN=50) :: fmt

      write(fmt,*) "(" , ny , "E10.3)"
      DO x=1,nx
         WRITE(*,fmt) atmos(x,:)
      END DO
    END SUBROUTINE print_atmos

  END PROGRAM RT_LIKE
```

We can see that the serial code can work for any array size. For example, below we show runs for array sizes 4x4 and 7x9

```
[angelv]$ ./rt-serial
4 4
  ------------ Initial values -----------------
 0.358E+01 0.592E+01 0.827E+01 0.106E+02
 0.481E+01 0.716E+01 0.950E+01 0.118E+02
 0.605E+01 0.839E+01 0.107E+02 0.131E+02
 0.728E+01 0.963E+01 0.120E+02 0.143E+02


  ------------ Final values -----------------
 0.358E+01 0.537E+01 0.744E+01 0.964E+01
 0.481E+01 0.660E+01 0.867E+01 0.110E+02
 0.605E+01 0.784E+01 0.102E+02 0.125E+02
 0.728E+01 0.963E+01 0.120E+02 0.143E+02
```

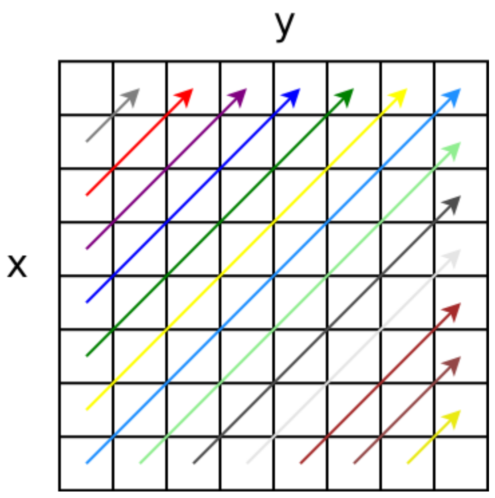


Figure C.4: RT serial

```
[angelv]$ ./rt-serial
7 9
  ------------ Initial values -----------------
 0.358E+01 0.592E+01 0.827E+01 0.106E+02 0.130E+02 0.153E+02 0.176E+02 0.200E+02 0.223E+02
 0.481E+01 0.716E+01 0.950E+01 0.118E+02 0.142E+02 0.165E+02 0.189E+02 0.212E+02 0.236E+02
 0.605E+01 0.839E+01 0.107E+02 0.131E+02 0.154E+02 0.178E+02 0.201E+02 0.225E+02 0.248E+02
 0.728E+01 0.963E+01 0.120E+02 0.143E+02 0.167E+02 0.190E+02 0.214E+02 0.237E+02 0.260E+02
 0.852E+01 0.109E+02 0.132E+02 0.156E+02 0.179E+02 0.202E+02 0.226E+02 0.249E+02 0.273E+02
 0.975E+01 0.121E+02 0.144E+02 0.168E+02 0.191E+02 0.215E+02 0.238E+02 0.262E+02 0.285E+02
 0.110E+02 0.133E+02 0.157E+02 0.180E+02 0.204E+02 0.227E+02 0.251E+02 0.274E+02 0.297E+02


  ------------ Final values -----------------
 0.358E+01 0.537E+01 0.744E+01 0.964E+01 0.119E+02 0.142E+02 0.166E+02 0.189E+02 0.212E+02
 0.481E+01 0.660E+01 0.867E+01 0.109E+02 0.132E+02 0.155E+02 0.178E+02 0.202E+02 0.225E+02
 0.605E+01 0.784E+01 0.990E+01 0.121E+02 0.144E+02 0.167E+02 0.191E+02 0.214E+02 0.238E+02
 0.728E+01 0.907E+01 0.111E+02 0.133E+02 0.157E+02 0.180E+02 0.204E+02 0.227E+02 0.251E+02
 0.852E+01 0.103E+02 0.124E+02 0.147E+02 0.171E+02 0.194E+02 0.218E+02 0.241E+02 0.264E+02
 0.975E+01 0.115E+02 0.139E+02 0.162E+02 0.186E+02 0.209E+02 0.233E+02 0.256E+02 0.280E+02
 0.110E+02 0.133E+02 0.157E+02 0.180E+02 0.204E+02 0.227E+02 0.251E+02 0.274E+02 0.297E+02
```

Your goal then is to parallelize this code. You can assume that rank 0 initializes the whole array (which is useful for printing), and then it is scattered amongst all the processes, which will run the procedure rt_l() on their local chunk of the array, then gathered again in rank 0 to print the final value. In the parallel code, besides getting the size of the array (*nx, ny*), you should also get as input the number of blocks you will do in every dimension *blocks_x, blockx_y*, so that the number of total processes in your parallel run is *blocks_x* * *blocks_y*. To clarify with one example, see figure C.5. There, nx=8 and ny=8, and we want to use 4 processes (i.e., we will run the code as *mpirun -np 4 [...]*). The number of blocks in X is 2, and the number of blocks in Y is also 2, so each process will end up working with a chunk of size=4x4 (but will need to make the local array bigger with ghost cells in order to store the corresponding values from the neighbour processes).

The idea here is quite similar to the heat2D exercise in 7.2.5, so you can reuse some of the code from the

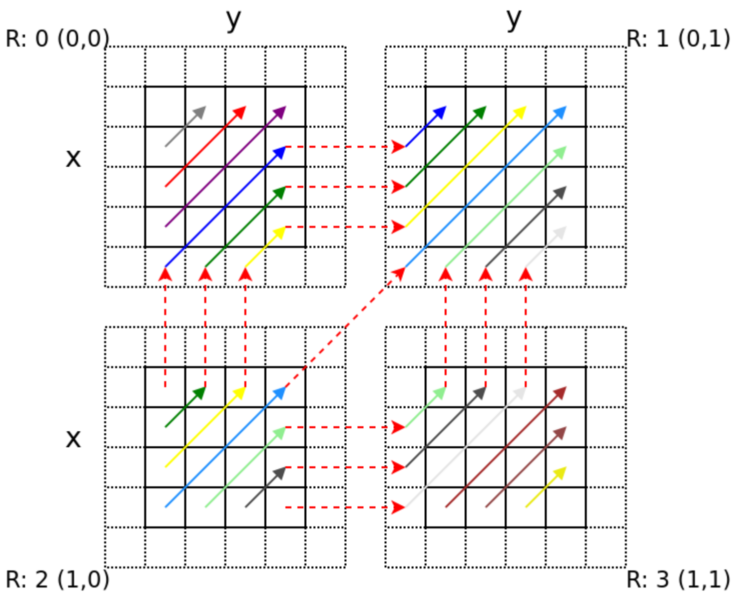


Figure C.5: RT parallel

solution to that exercise. When making the code generic, you will probably find useful to create a cartesian distribution of processes, so for example the process 0 in figure C.5 (with label “R: 0 (0,0)”) could have process coordinates (0,0) while process with rank 3 (“R: 3 (1,1)”) would have process coordinates (1,1). Knowing which process row and/or column a process is in, will be crucial to understand in which order it has to perform the receive/send operations, where to start propagating the rays, etc.

The idea is not too difficult, but it can get confusing with indexes, etc. quickly, so start assuming that you will always run it with 4 processes in a distribution as per figure C.5 and with a number of grid cells that is divisible by 2. This will give you an idea of the difficulties with indexes, etc. but it should be relatively easy. **If your code works up to here, getting *EXACTLY* the same final values as in the serial version, you would get 7 out of 10 points for this project**.

The next step would be to make the work code with other processes numbers and distributions, but still assuming that the number of cells in each dimension is divisible by the number of blocks in that dimension, so all processes end up with chunks of the same size. **If you are able to get it to work up to here, then you will have full marks for this project.**

EXTRA POINTS: If you are able to make it work for any distribution but also for a number of cells not divisible by the number of blocks, then this will give you full marks in this project, but also 1 point more in the final mark of the course (10 being the maximum possible mark).

Below you can see executions of my solution for the array sizes as per the serial code above. The first one (4x4 and 2 blocks in each dimension should be your first goal, where each process will have a chunk of 2x2).

```
[angelv]$ mpirun -np 4 rt-par
 Global array dimensions?
4 4
 Number of blocks (x,y)?
```

```
2 2
  ------------ Initial values -----------------
 0.358E+01 0.592E+01 0.827E+01 0.106E+02
 0.481E+01 0.716E+01 0.950E+01 0.118E+02
 0.605E+01 0.839E+01 0.107E+02 0.131E+02
 0.728E+01 0.963E+01 0.120E+02 0.143E+02


  ------------ Final values -----------------
 0.358E+01 0.537E+01 0.744E+01 0.964E+01
 0.481E+01 0.660E+01 0.867E+01 0.110E+02
 0.605E+01 0.784E+01 0.102E+02 0.125E+02
 0.728E+01 0.963E+01 0.120E+02 0.143E+02



[angelv]$ mpirun --oversubscribe -np 12 rt-par
 Global array dimensions?
7 9
 Number of blocks (x,y)?
3 4
  ------------ Initial values -----------------
 0.358E+01 0.592E+01 0.827E+01 0.106E+02 0.130E+02 0.153E+02 0.176E+02 0.200E+02 0.223E+02
 0.481E+01 0.716E+01 0.950E+01 0.118E+02 0.142E+02 0.165E+02 0.189E+02 0.212E+02 0.236E+02
 0.605E+01 0.839E+01 0.107E+02 0.131E+02 0.154E+02 0.178E+02 0.201E+02 0.225E+02 0.248E+02
 0.728E+01 0.963E+01 0.120E+02 0.143E+02 0.167E+02 0.190E+02 0.214E+02 0.237E+02 0.260E+02
 0.852E+01 0.109E+02 0.132E+02 0.156E+02 0.179E+02 0.202E+02 0.226E+02 0.249E+02 0.273E+02
 0.975E+01 0.121E+02 0.144E+02 0.168E+02 0.191E+02 0.215E+02 0.238E+02 0.262E+02 0.285E+02
 0.110E+02 0.133E+02 0.157E+02 0.180E+02 0.204E+02 0.227E+02 0.251E+02 0.274E+02 0.297E+02


  ------------ Final values -----------------
 0.358E+01 0.537E+01 0.744E+01 0.964E+01 0.119E+02 0.142E+02 0.166E+02 0.189E+02 0.212E+02
 0.481E+01 0.660E+01 0.867E+01 0.109E+02 0.132E+02 0.155E+02 0.178E+02 0.202E+02 0.225E+02
 0.605E+01 0.784E+01 0.990E+01 0.121E+02 0.144E+02 0.167E+02 0.191E+02 0.214E+02 0.238E+02
 0.728E+01 0.907E+01 0.111E+02 0.133E+02 0.157E+02 0.180E+02 0.204E+02 0.227E+02 0.251E+02
 0.852E+01 0.103E+02 0.124E+02 0.147E+02 0.171E+02 0.194E+02 0.218E+02 0.241E+02 0.264E+02
 0.975E+01 0.115E+02 0.139E+02 0.162E+02 0.186E+02 0.209E+02 0.233E+02 0.256E+02 0.280E+02
 0.110E+02 0.133E+02 0.157E+02 0.180E+02 0.204E+02 0.227E+02 0.251E+02 0.274E+02 0.297E+02
```

The second one is the most generic one (and probably quite difficult to get right). The number of cells in x is 7 and we want to divide them in 3 blocks, so two blocks should have two cells in x while another one should have 3 cells, etc. In case it is not clear, the way my code distributes the 12 processes in chunks is as follows:

```
| 3x3 | 3x2 | 3x2 | 3x2 |
| 2x3 | 2x2 | 2x2 | 2x2 |
| 2x3 | 2x2 | 2x2 | 2x2 |
```

This is not easy, but it is not too difficult either. Start with the 4 process division, which will help you to understand which things have to be generalized in order to be able to get it to work with an arbitrary distribution.

PART VI

BIBLIOGRAPHY, INDEX, LIST OF ACRONYMS, APPENDICES, ETC.

Appendix D

Bibliography

[1] Schlitt, Wayne. *The XStar N-body Solver. Theory of Operation.* `http://www.schlitt.net/xstar/n-body.pdf`.

[2] L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. In *Parallel Programming using C++, G. V. Wilson and P. Lu, editors*, pages 175–213. MIT Press, 1996.