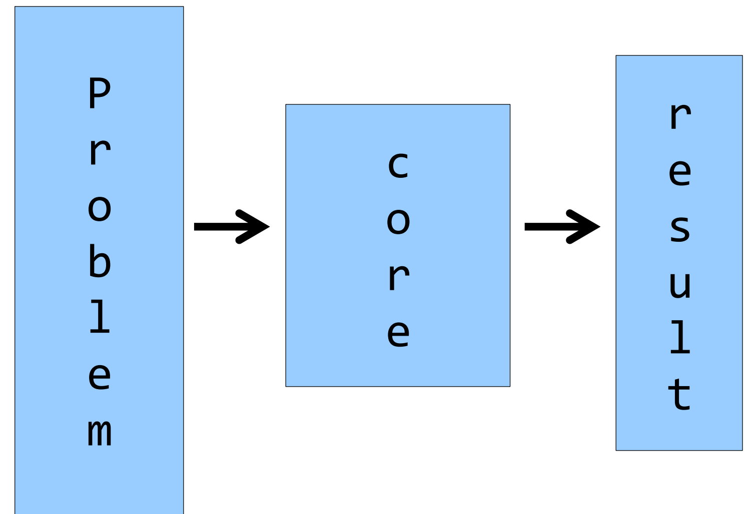
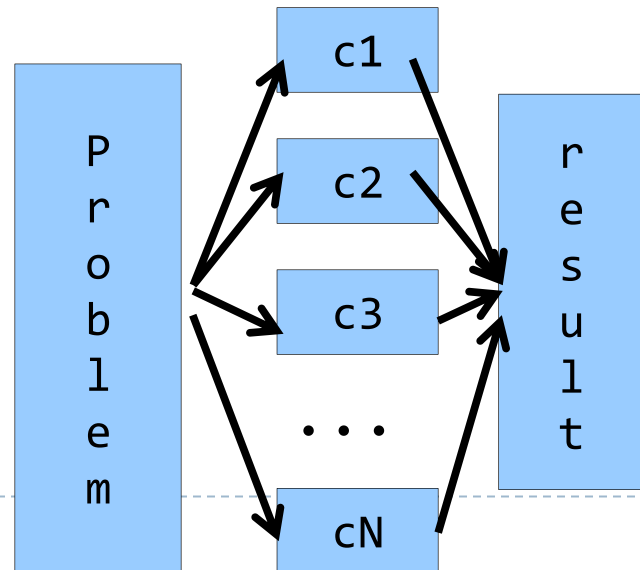
Lecture V: Introduction to parallel programming with Fortran coarrays

What is parallel computing?

- Serial computing
  - Single processing unit (core) is used for solving a problem
  - One task processed at a time
- Parallel computing
  - Multiple cores are used for solving a problem
  - Problem is split into smaller subtasks
  - Multiple subtasks are processed *simultaneously*



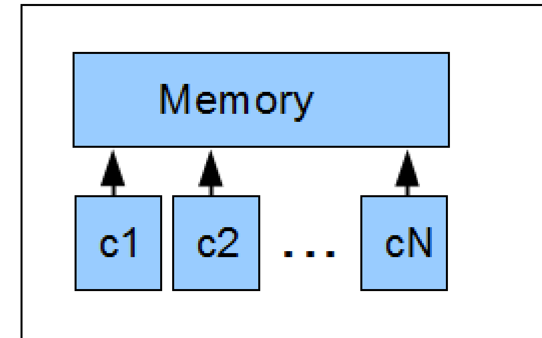


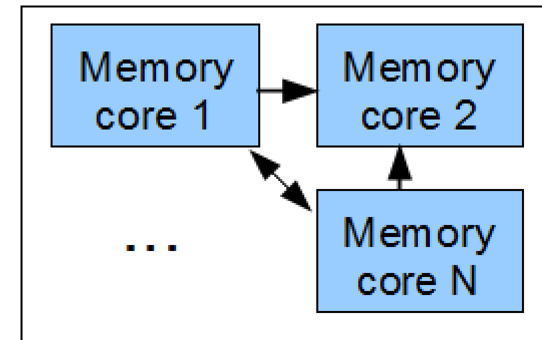
Why parallel computing?

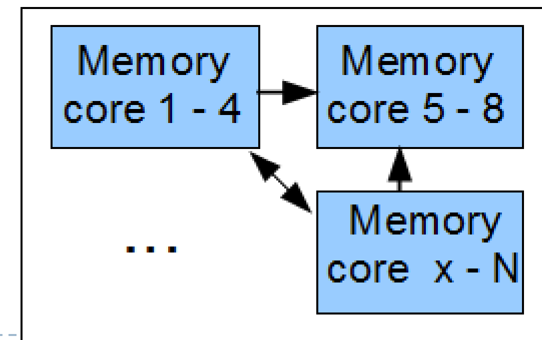
- Solve problems faster
  - CPU clock frequencies are no longer increasing - speed-up is obtained by using multiple cores
  - Parallel programming is required for utilizing multiple cores
- Solve bigger problems
  - Parallel computing may allow application to use more memory
  - Apply old models to new length and time scales
  - Grand challenges
- Solve problems better
  - More precise models

Types of parallel computers

- Shared memory
  - all the cores can access the whole memory
- Distributed memory
  - all the cores have their own memory
  - communication is needed in order to access the memory of other cores
- Current supercomputers combine the distributed memory and shared memory approaches







What are Fortran coarrays about ?

- Adds parallel processing as part of Fortran language
  - Only small changes required to convert existing Fortran code to support a robust and potentially efficient parallelism
- A Partitioned Global Address Space (PGAS) language
  - Coarrays implement parallelism over "distributed shared memory" → potentially massively parallel
- Has been integrated into Fortran 2008 standard
  - Only few compilers so far support the syntax
    - CCE for real
    - Intel tops on MPI
    - GNU supports only one image

Coarrays…

- Add only a few new rules to the Fortran language
- Provide mechanisms to allow
  - SPMD (Single Program, Multiple Data) style of explicitly parallel programming
  - Data distribution over partitioned memory (you can think about "distributed shared memory" here)
  - Guard against race conditions (in e.g. variable value assignments) by using synchronization
  - Memory management for dynamic shared entities

Execution model

- Upon startup a coarrays program gets replicated into a number of copies called *images* (i.e. processes)
  - The number of images is usually decided at the execution time
- Each "replica" (image) runs asynchronously in a loosely/non-coupled way until program controlled synchronization
- Image's (local) data are visible within the image only – except for data declared as special arrays i.e. *coarrays*
  - One-sided data communication enables movement of coarray data across different images of a program

Time for "Hello World"!

- `num_images()` returns the number of images in use for this run (usually set outside the program, by the environment)
- `this_image()` returns the image number in concern – numbered from 1 to `num_images()`
- This program is a trivially parallel i.e. each image does not explicitly share any data and runs seemingly independently

```
program hello_world
   implicit none
   write(*,*) 'Hello world from ', &
    this_image() , 'of', num_images()
end program hello_world
```

Declaring coarrays

- An entirely new data structure, coarrays, become meaningful in parallel programming context, when their data are remotely accessible by its images
- Accomplished through additional Fortran syntax for coarrays for Fortran arrays or scalars, for example :

```
integer, codimension[*] :: scalar
integer :: scalar[*]
real, dimension(64), codimension[*] :: vector
real :: vector(64)[*]
```

- Declares a `scalar` with a local instance *on every image*
- Declares a `vector` with 64 elements *on every image*

Declaring coarrays

- The square brackets [*] denote allocation of special coarrays over allocated images (decided upon program startup)
  - The round brackets "( )" mean local array accesses, and the "[ ]" are meant for remote data array accesses only

```
integer :: global(3)[*], local(3)
global(:) = this_image() * (/ 1, 2, 3 /) ! local initialization
local(:) = global(:)[1]  ! copy from image number 1 to every
                         ! image
```

Synchronization

- We need to be careful when updating coarrays
  - Is the remote data we are copying valid i.e. up to date?
  - Could another image overwrite our data without notice?
  - How do we know if the remote update (fetch) is complete?
- Fortran provides synchronization statements, e.g. adds a barrier for synchronization of all images

```
SYNC ALL
```

- To be absolutely sure we are getting correct result, we need to modify our previous copying example a little …

Synchronization: corrected remote copy

- We need to add barrier synchronization of all images before the copying takes place to be absolutely sure we are getting the most up to date copy of `global(:)[1]`

```
global(:) = this_image() * (/ 1, 2, 3 /)
sync all
local(:) = global(:)[1]
```

- In this particular case – since only the image #1 is in a critical position, we could use an alternative, pairwise form of synchronization:

```
global(:) = this_image() * (/ 1, 2, 3 /)
sync images(1)
local(:) = global(:)[1]
```

Interim summary: basic concepts

- About parallel processing in general
- Concept of *images* and some related functions
- How to declare codimensional arrays (*coarrays*) and access their data
- Image synchronization

Multiple codimensions

- Multidimensional coarrays are possible also in terms of codimension
  - The last codimension must always be declared with asterisk "*"
  - Sum of rank plus corank must be ≤ 15
  - The bounds of codimensions start from 1 by default but can be adjusted

```
integer, codimension[2,*] :: scalar
real, dimension(64,64), codimension[4,*] :: matrix
real, dimension(128,128,128), codimension[0:1,0:1,0:*] :: grid
```

image_index( ) and this_image( )

- So far we have seen **`this_image()`** function been used without arguments
- In its another calling form it takes a co-array as an argument, e.g.

```
real :: a(7,7) [0:3,3:*]
print *, this_image(a)
```

- When running with 10 images it returns (say) for image#7 a vector (2, 4)
- image_index() performs the opposite conversion, i.e. returns a linear number of codimensional coordinates

	3	4	5
0	1	5	9
1	2	6	10
2	3	7	0
3	4	8	0

`A(:,:)[2,4]`

Coarrays in procedures

- When declared in a subroutine or function, a co-array must be one the following
  - Declared as a dummy argument to the procedure
  - Have ALLOCATABLE and/or SAVE attribute
- Re-mapping of corank is also allowed
- A coarray in procedure cannot be an automatic array

Co-arrays in procedures

```
subroutine some_routine (n, array_1, co_array_1, array_2, co_array_2, co_array_3)
implicit none
!-- procedure arguments
integer, intent(in) :: n
integer, intent(inout) :: array_1(n)         ! explicit shape
integer, intent(inout) :: co_array_1(n)[*]   ! explicit shape
integer, intent(inout) :: array_2(:)         ! assumed shape
integer, intent(inout) :: co_array_2(:)[*]   ! assumed shape
integer, intent(inout) :: co_array_3[:]      ! illegal : assumed co-shape
!-- procedure variable declarations (not all ok – see below)
integer :: local_array_1(n)                   ! ok, an automatic (regular) array
integer :: local_array_2(1000)                ! ok, local (regular) array
integer :: local_co_array_1(n)[*]             ! invalid : co-array can’t be automatic
integer :: local_co_array_2(1000)[*]          ! invalid : save-attribute missing
integer, save :: local_co_array_3(1000)[*]    ! ok, co-array with save-attr
integer, allocatable :: local_co_array_4(:)[:] ! ok, co-array with allocatable
integer, pointer :: local_co_array_5(:)[:]    ! invalid : co-array can’t have pointer
end subroutine some_routine
```

I/O conventions

- Each image has its own, independent set of Fortran input and output units
- The default input ("stdin", i.e. READ(*,…) etc) is pre-connected to the master image (image#1) only
  - Do stdin with the image #1 only and broadcast the data
  - Similarly with command-line input

```
program safe_and_correct_stdin
integer :: flag[*] = 0, i
if ( this_image() == 1 ) then
   read *,flag
   do i = 2, num_images()
      flag[i] = flag
   enddo
endif
sync all
end program safe_and_correct_stdin
```

I/O conventions

- The default output (“stdout”) is connected to all images
  - Output is merged (in any order) into one output stream
  - PRINT *, WRITE(*,…), WRITE(6,…)
- The standard error (“stderr”) is redirected to the “stdout”
  - WRITE(0,…)

ALLOCATABLE coarrays

- It is possible to define dynamic coarrays, where both shape (i.e. locally owned part) and co-shape are dynamic, e.g. an allocatable with deferred shapes:

```
integer , allocatable :: a(:) [:]
...
allocate (a(1000)[*])
deallocate (a)
```

- ALLOCATE and DEALLOCATE imply implicit synchronization – *all images* must participate i.e. an implicit `sync all` occurs
- The local size (here: 1000) must be the same on each image
- The last co-dimension must have an asterisk "*"

About POINTERs with coarrays

- A coarray declaration cannot have a POINTER attribute
- Thus the following would be illegal:

```
real, pointer :: ptr[:] ! this is invalid Fortran
```

- However, we *can* define a new TYPE, where type component(s) have POINTER (or ALLOCATABLE) attributes
  - And then define a coarray being of that TYPE
- Used in dynamic dimensioning of coarrays
  - This way *local sizes* on every image can be different

Variable length coarrays via ALLOCATABLE

- Create a new Fortran data type with ALLOCATABLE component in it – and place it in a MODULE

```
type mytype
  real, allocatable :: data(:)
end type mytype
```

- Then declare a coarray of that type, e.g.

```
type (mytype) :: co[*]
```

- ALLOCATE on each image, but *different size*

```
allocate (co % data (10 * this_image()))
```

- Refer to the data of another image such as

```
element = co[1] % data(1)
```

Variable length coarrays via POINTER

- Or, after defining

```
TYPE (mytype) :: co[*]
REAL, TARGET :: chunk(1000)
```

- Make `co%data` to point different chunks of local data

```
CALL get_my_range(size(chunk), istart, iend)
co%data => chunk(istart:iend)
```

Summary of the latter part

- Multiple codimensions & bounds
- Using coarrays in procedures
- I/O in Fortran coarrays
- How to define coarrays with varying-sized local parts

Future Fortran coarrays

- Coarrays continue to evolve
- Technical specification (TS) outlines the future developments
- In the pipeline are the following extensions to the CAF
  - Image teams
  - Fault tolerant features
  - Collective intrinsic functions for coarrays
    - `co_broadcast`
    - `co_sum, co_min, co_max`
    - `co_reduce`
  - Etc

Programming assignment

Parallelize the Game of Life program (assignment #3) with coarrays. by dividing the board in columns and assigning one column to one image. A domain decomposition, that is.

The images are able to update their part of the board independently everywhere else than on the column boundaries - there the communication of a single column with the nearest image is needed (the board is periodic, so the first column of the board is 'connected' to the last column). This is realized by having additional ghost layers on each of the local columns, that contain the boundary data of the neighboring tasks. The periodicity in the other direction is accounted as earlier. Make all the images print their own parts of the board on different files, e.g. life_nnnn_mm.pbm, where nnnn is the iteration and mm is the image number.

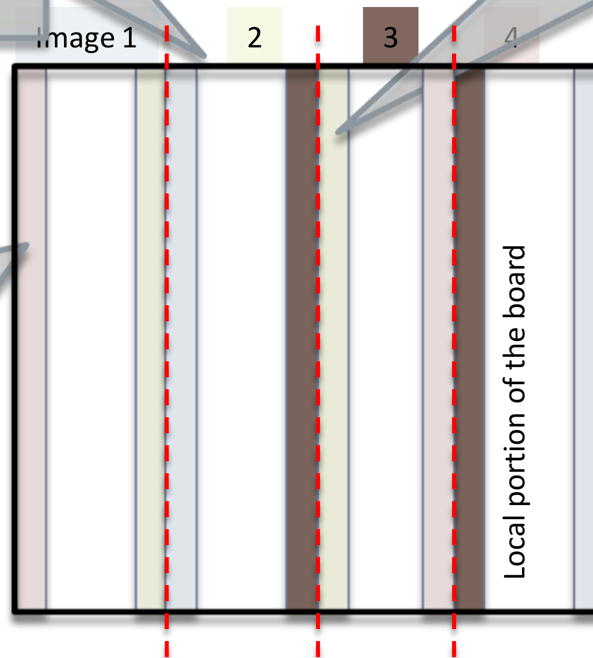
You can ease the problem by requiring the board width to be dividable by the number of images.

You will need to use either CSC's or FMI's Cray XC (Sisu or Voima).

Programming assignment

Each image will have additional columns in both ends of the local board. Before each update the last column of each image's board has to be copied to the ghost layer of the next one

Also the first column in the local board of each image has to be availed in the ghost layer of the previous rank



Due to problem periodicity, the image #1 has to avail its first column for the image #4. The image #4 copies its last column to image #1.