# Parallel Programming Examples

## Examples in CFD Using FORTRAN

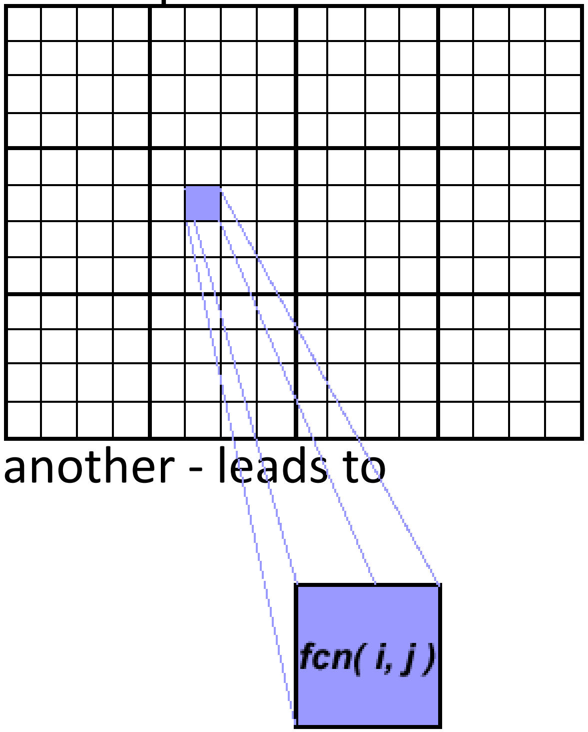
# Agenda

- **Array Processing**
- **PI Calculation**
- **Simple Heat Equation**
- **1-D Wave Equation**

# Array Processing

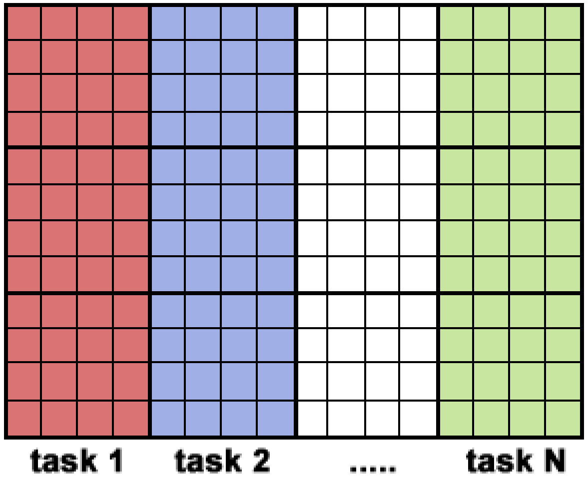
- This example demonstrates calculations on 2-dimensional array elements, with the computation on each array element being independent from other array elements.
- The serial program calculates one element at a time in sequential order.
- Serial code could be of the form:

```
do j = 1,n
    do i = 1,n
        a(i,j) = fcn(i,j)
    end do
end do
```




- The calculation of elements is independent of one another - leads to an embarrassingly parallel situation.
- The problem should be computationally intensive.

# Array Processing Solu




- Arrays elements are distributed so that each processor owns a portion of an array (subarray).
- Independent calculation of array elements insures there is no need for communication between tasks.
- Distribution scheme is chosen by other criteria, e.g. unit stride (stride of 1) through the subarrays. Unit stride maximizes cache/memory usage.
- Since it is desirable to have unit stride through the subarrays, the choice of a distribution scheme depends on the programming language. See the Block - Cyclic Distributions Diagram for the options.
- After the array is distributed, each task executes the portion of the loop corresponding to the data it owns. For example, with Fortran block distribution:

```
do j = mystart, myend
     do i = 1,n
          a(i,j) = fcn(i,j)
     end do
end do
```

- Notice that only the outer loop variables are different from the serial solution.

# Array Processing Solution 1
# One possible implementation

- Implement as SPMD model.
- Master process initializes array, sends info to worker processes and receives results.
- Worker process receives info, performs its share of computation and sends results to master.
- Using the Fortran storage scheme, perform block distribution of the array.
- Pseudo code solution: **red** highlights changes for parallelism.

# Array Processing Solution 1

## One Possible Implementation

```
find out if I am MASTER or WORKER

if I am MASTER

  initialize the array
  send each WORKER info on part of array it owns
  send each WORKER its portion of initial array

  receive from each WORKER results

else if I am WORKER
  receive from MASTER info on part of array I own
  receive from MASTER my portion of initial array

  # calculate my portion of array
  do j = my first column,my last column
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
  end do

  send MASTER results

endif
```

# Array Processing Solution 2: Pool of Tasks

- The previous array solution demonstrated static load balancing:
  - Each task has a fixed amount of work to do
  - May be significant idle time for faster or more lightly loaded processors - slowest tasks determines overall performance.
- Static load balancing is not usually a major concern if all tasks are performing the same amount of work on identical machines.
- If you have a load balance problem (some tasks work faster than others), you may benefit by using a "pool of tasks" scheme.

# Array Processing Solution 2
# Pool of Tasks Scheme

- Two processes are employed
- Master Process:
  - Holds pool of tasks for worker processes to do
  - Sends worker a task when requested
  - Collects results from workers
- Worker Process: repeatedly does the following
  - Gets task from master process
  - Performs computation
  - Sends results to master
- Worker processes do not know before runtime which portion of array they will handle or how many tasks they will perform.
- Dynamic load balancing occurs at run time: the faster tasks will get more work to do.
- Pseudo code solution: **red** highlights changes for parallelism.

# Array Processing Solution 2 Pool of Tasks Scheme

```
find out if I am MASTER or WORKER

if I am MASTER

  do until no more jobs
    send to WORKER next job
    receive results from WORKER
  end do

  tell WORKER no more jobs

else if I am WORKER

  do until no more jobs
    receive from MASTER next job

    calculate array element: a(i,j) = fcn(i,j)

    send results to MASTER
  end do

endif
```

# Pi Calculation

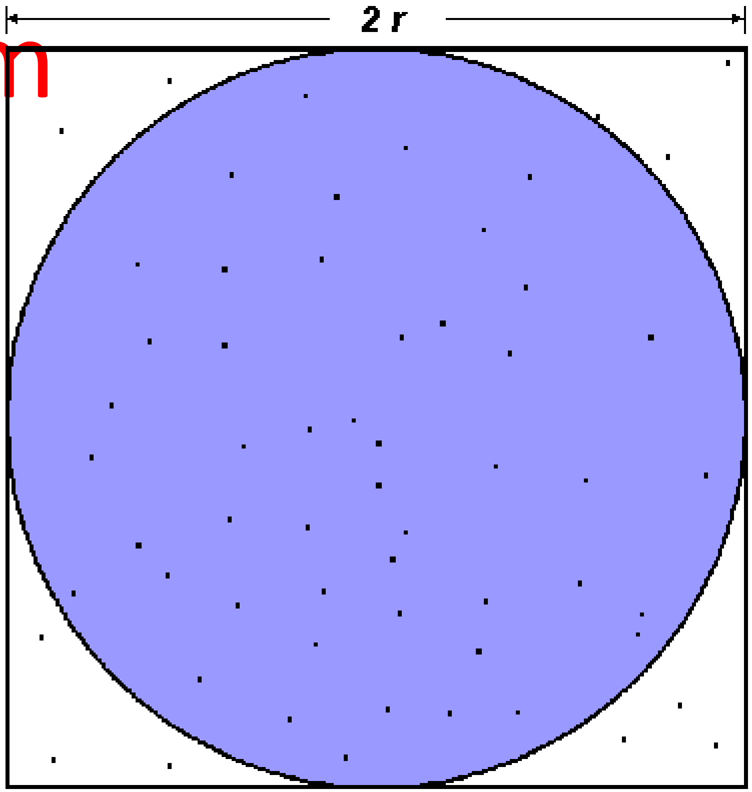
- The value of PI can be calculated in a number of ways. Consider the following method of approximating PI
  - Inscribe a circle in a square
  - Randomly generate points in the square
  - Determine the number of points in the square that are also in the circle
  - Let r be the number of points in the circle divided by the number of points in the square
  - PI ~ 4 r
- Note that the more points generated, the better the approximation

# Discussion

- In the above pool of tasks example, each task calculated an individual array element as a job. The computation to communication ratio is finely granular.
- Finely granular solutions incur more communication overhead in order to reduce task idle time.
- A more optimal solution might be to distribute more work with each job. The "right" amount of work is problem dependent.

# Algorithm

```
npoints = 10000
circle_count = 0
do j = 1,npoints
   generate 2 random numbers between
   0 and 1
   xcoordinate = random1 ;
   ycoordinate = random2
   if (xcoordinate, ycoordinate)
   inside circle  then circle_count =
   circle_count + 1
end do
PI = 4.0*circle_count/npoints
```




$$A_S = (2r)^2 = 4r^2$$

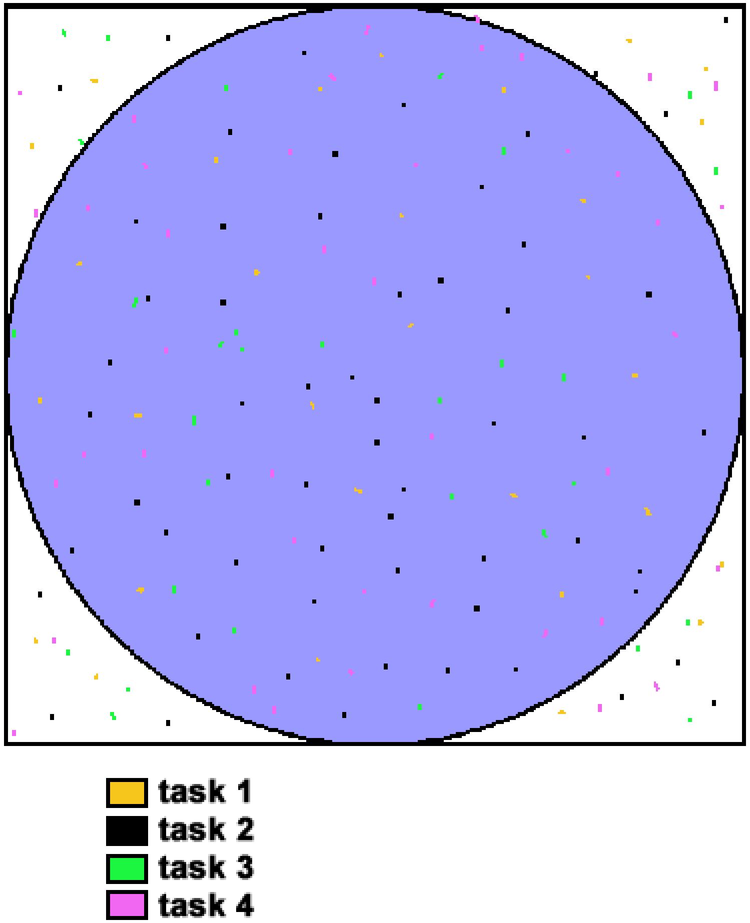
$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

- Note that most of the time in running this program would be spent executing the loop
- Leads to an embarrassingly parallel solution
  - Computationally intensive
  - Minimal communication
  - Minimal I/O

# PI Calculation
# Parallel Solution

- Parallel strategy: break the loop into portions that can be executed by the tasks.
- For the task of approximating PI:
  - Each task executes its portion of the loop a number of times.
  - Each task can do its work without requiring any information from the other tasks (there are no data dependencies).
  - Uses the SPMD model. One task acts as master and collects the results.
- Pseudo code solution: **red** highlights changes for parallelism.

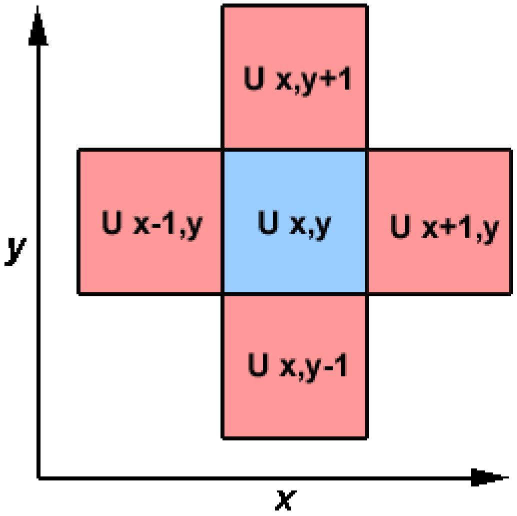


# PI Calculation
# Parallel Solution

```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1 ; ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER

  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)

else if I am WORKER

  send to MASTER circle_count

endif
```

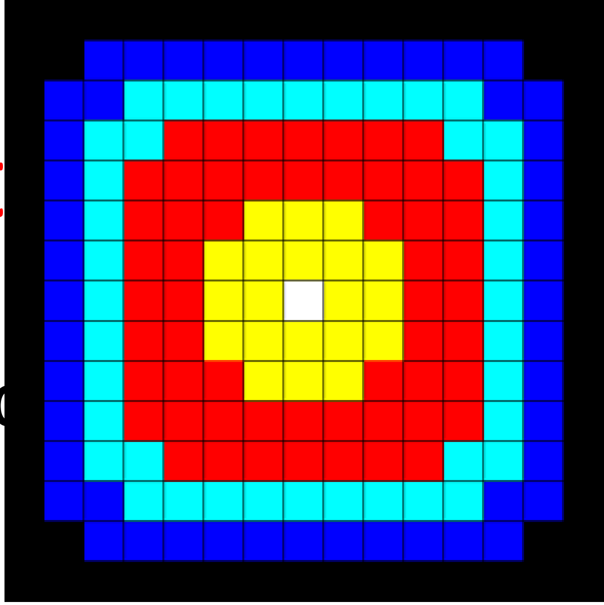
# Simple Heat Equation

- Most problems in parallel computing require communication among the tasks. A number of common problems require communication with "neighbor" tasks.
- The heat equation describes the temperature change over time, given initial temperature distribution and boundary conditions.
- A finite differencing scheme is employed to solve the heat equation numerically on a square region.
- The initial temperature is zero on the boundaries and high in the middle.
- The boundary temperature is held at zero.




- For the fully explicit problem, a time stepping algorithm is used. The elements of a 2-dimensional array represent the temperature at points on the square.

# Simple Heat Equat



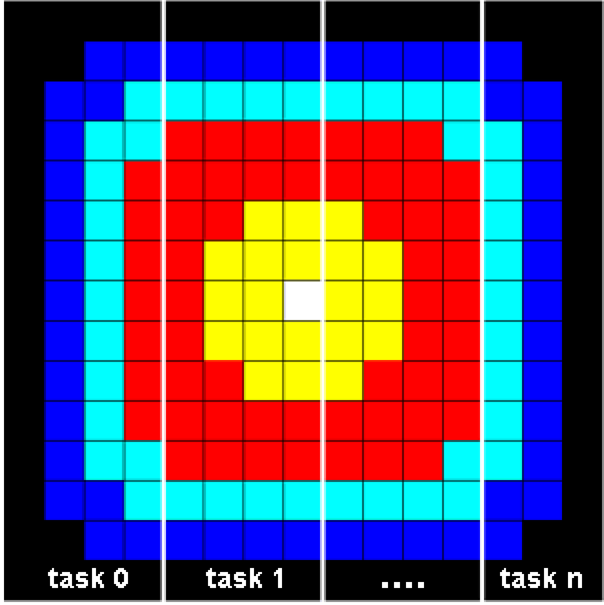
- The calculation of an element is d
upon ne values.

$$U_{x,y} = U_{x,y}$$
$$+ C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy})$$
$$+ C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})$$

- A e:

```
do iy = 2, ny - 1
do ix = 2, nx - 1
  u2(ix, iy) =
    u1(ix, iy)  +
      cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) +
      cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy))
end do
end do
```

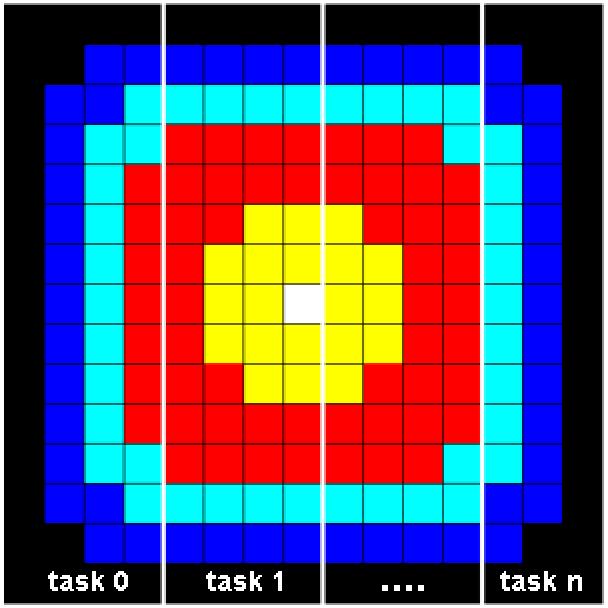
# Parallel Solution




- Implement as an SPMD model
- The entire array is partitioned and distributed as subarrays to all tasks. Each task owns a portion of the total array.
- Determine data dependencies
  - interior elements belonging to a task are independent of other tasks
  - border elements are dependent upon a neighbor task's data, necessitating communication.
- Master process sends initial info to workers, checks for convergence and collects results
- Worker process calculates solution, communicating as necessary with neighbor processes
- Pseudo code solution: **red** highlights changes for parallelism.

# Parallel Solution 1

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray

  do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do

  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged
    update time
    send neighbors my border info
    receive from neighbors their border info

    update my portion of solution array

    determine if my solution has converged
      send MASTER convergence data
      receive from MASTER convergence signal
  end do

  send MASTER results

endif
```



# Parallel Solution 2
# Overlapping Communication and Computation

- In the previous solution, it was assumed that blocking communications were used by the worker tasks. Blocking communications wait for the communication process to complete before continuing to the next program instruction.
- In the previous solution, neighbor tasks communicated border data, then each process updated its portion of the array.
- Computing times can often be reduced by using non-blocking communication. Non-blocking communications allow work to be performed while communication is in progress.
- Each task could update the interior of its part of the solution array while the communication of border data is occurring, and update its border after communication has completed.
- Pseudo code for the second solution: **red** highlights changes for non-blocking communications.

# Parallel Solution 2
# Overlapping Communication and Computation

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray

  do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do

  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged
    update time

    non-blocking send neighbors my border info
    non-blocking receive neighbors border info

    update interior of my portion of solution array
    wait for non-blocking communication complete
    update border of my portion of solution array

    determine if my solution has converged
      send MASTER convergence data
      receive from MASTER convergence signal
  end do

  send MASTER results

endif
```

# 1-D Wave Equation

- In this example, the amplitude along a uniform, vibrating string is calculated after a specified amount of time has elapsed.
- The calculation involves:
  - the amplitude on the y axis
  - i as the position index along the x axis
  - node points imposed along the string
  - update of the amplitude at discrete time steps.

# 1-D Wave Equation

- The equation to be solved is the one-dimensional wave equation:

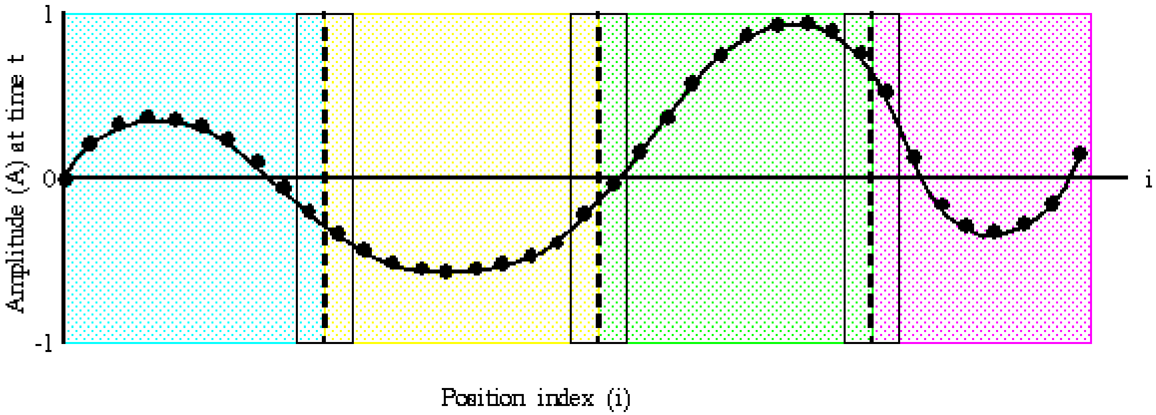
```
A(i,t+1) = (2.0 * A(i,t)) - A(i,t-1)
    + (c * (A(i-1,t) - (2.0 * A(i,t)) + A(i+1,t)))
```

  where c is a constant

- Note that amplitude will depend on previous timesteps (t, t-1) and neighboring points (i-1, i+1). Data dependence will mean that a parallel solution will involve communications.

# 1-D Wave Equation
# Parallel Solution

- Implement as an SPMD model
- The entire amplitude array is partitioned and distributed as subarrays to all tasks. Each task owns a portion of the total array.
- Load balancing: all points require equal work, so the points should be divided equally
- A block decomposition would have the work partitioned into the number of tasks as chunks, allowing each task to own mostly contiguous data points.
- Communication need only occur on data borders. The larger the block size the less the communication.



# 1-D Wave Equation Parallel Solution

```
find out number of tasks and task identities

#Identify left and right neighbors
left_neighbor = mytaskid - 1
right_neighbor = mytaskid +1
if mytaskid = first then left_neigbor = last
if mytaskid = last then right_neighbor = first

find out if I am MASTER or WORKER
if I am MASTER
  initialize array
  send each WORKER starting info and subarray
else if I am WORKER
  receive starting info and subarray from MASTER
endif

#Update values for each point along string
#In this example the master participates in calculations
do t = 1, nsteps
  send left endpoint to left neighbor
  receive left endpoint from right neighbor
  send right endpoint to right neighbor
  receive right endpoint from left neighbor

#Update points along line
  do i = 1, npoints
    newval(i) = (2.0 * values(i)) - oldval(i)
    + (sqtau * (values(i-1) - (2.0 * values(i)) + values(i+1)))
  end do

end do

#Collect results and write to file
if I am MASTER
  receive results from each WORKER
  write results to file
else if I am WORKER
  send results to MASTER
endif
```

**Thank You**

**www.itlectures.aa.am**

# 1-D Wave Equation Parallel Solution

```
find out number of tasks and task identities

#Identify left and right neighbors
left_neighbor = mytaskid - 1
right_neighbor = mytaskid +1
if mytaskid = first then left_neigbor = last
if mytaskid = last then right_neighbor = first

find out if I am MASTER or WORKER
if I am MASTER
  initialize array
  send each WORKER starting info and subarray
else if I am WORKER
  receive starting info and subarray from MASTER
endif

#Update values for each point along string
#In this example the master participates in calculations
do t = 1, nsteps
  send left endpoint to left neighbor
  receive left endpoint from right neighbor
  send right endpoint to right neighbor
  receive right endpoint from left neighbor

#Update points along line
  do i = 1, npoints
    newval(i) = (2.0 * values(i)) - oldval(i)
    + (sqtau * (values(i-1) - (2.0 * values(i)) + values(i+1)))
  end do

end do

#Collect results and write to file
if I am MASTER
  receive results from each WORKER
  write results to file
else if I am WORKER
  send results to MASTER
endif
```