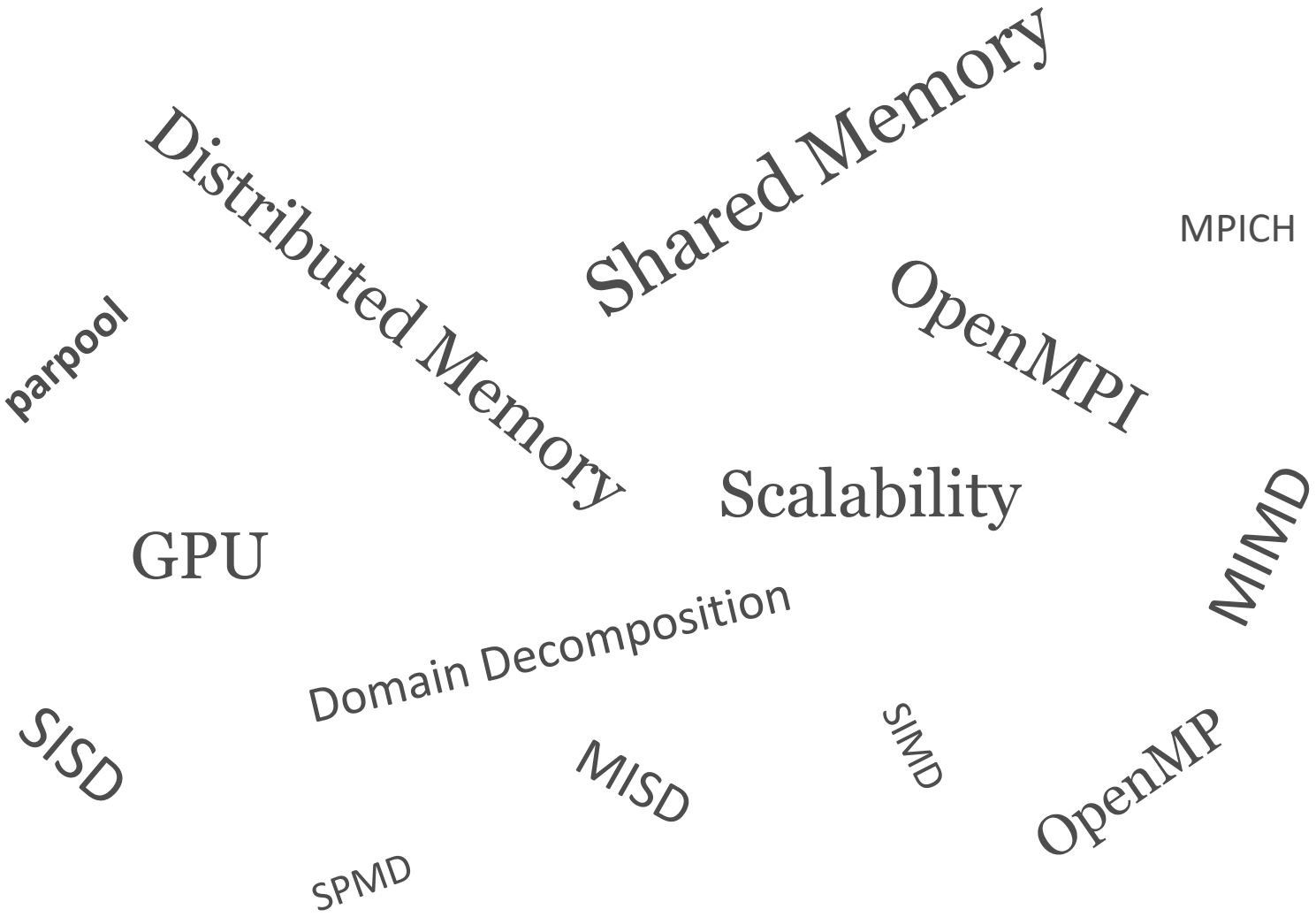
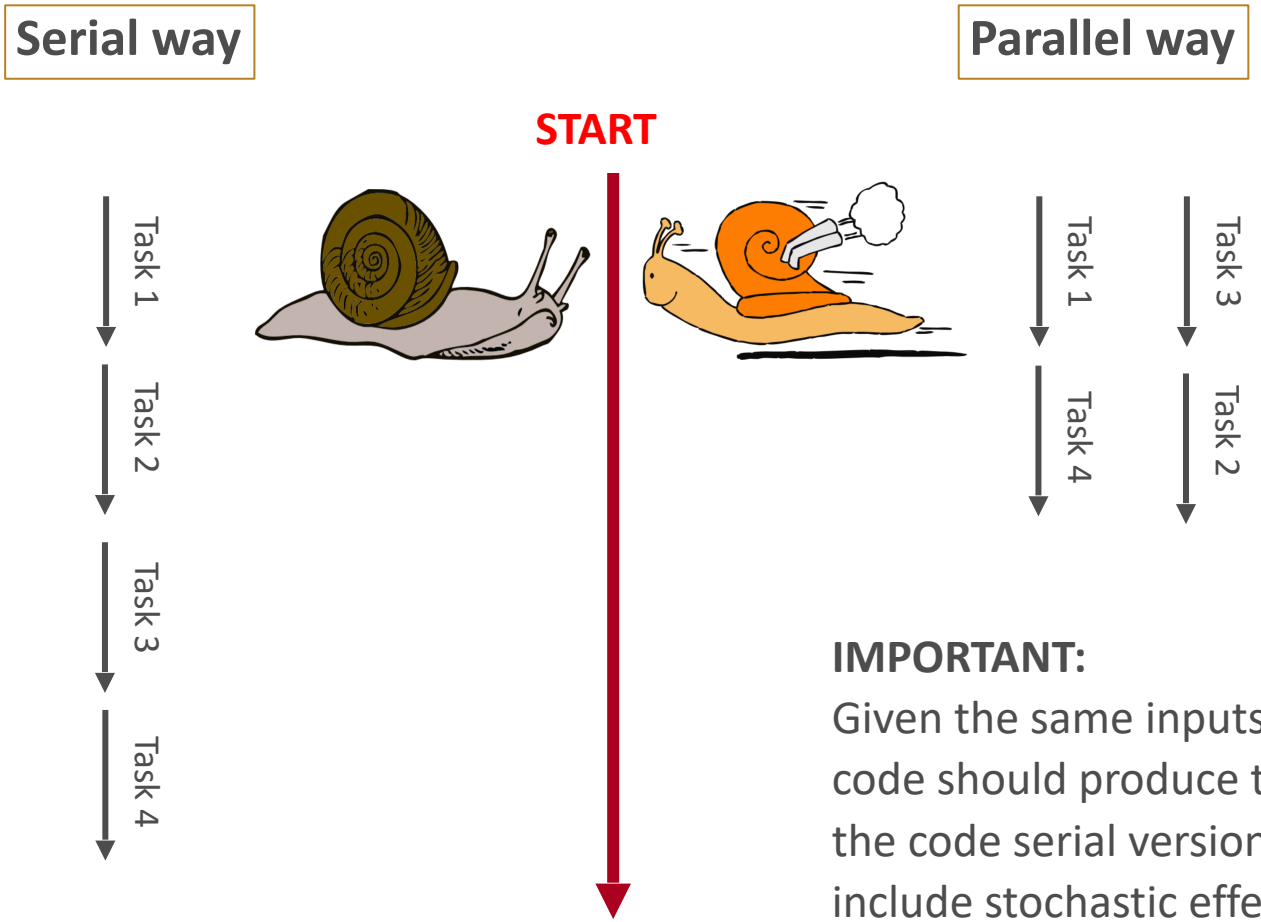
Introduction to Parallel Computing

Raffaele Potami

Parallelization is ...



Solving a problem in a …



IMPORTANT:
Given the same inputs, a parallelized code should produce the same output of the code serial version (exceptions include stochastic effects, numerical truncation errors, etc.)

Why should you use Parallel Computing?

- Increase Performance
- Save Time
- Solve Complex Problems
- Solve Large(r) Problems
- Optimize Memory Usage

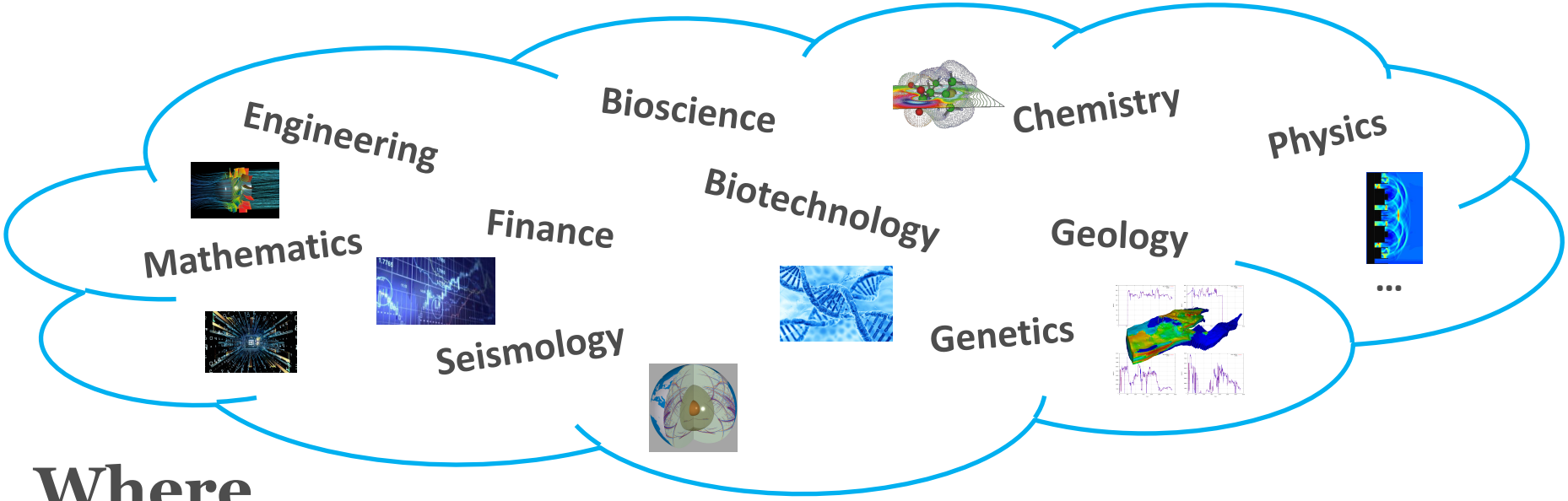
Example:
Process 1000 large images

- 5 min run-time x image ~ 3.5 days of computing on single core
- Use 20 cores and get it done in ~ 4 hours

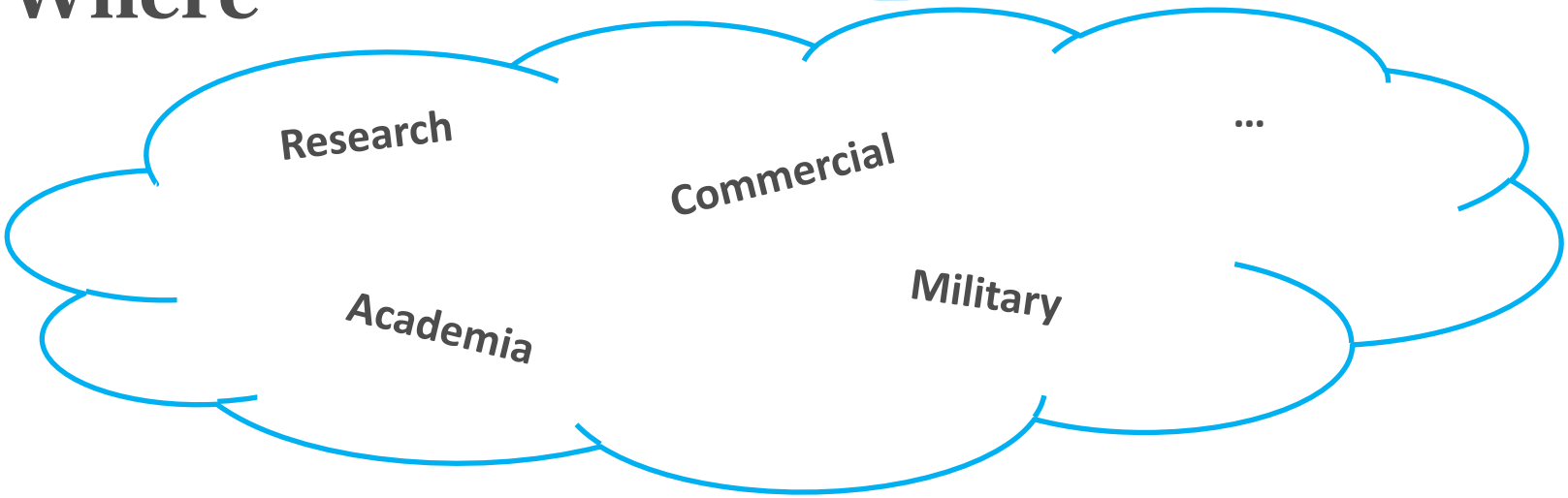
Example:
Process a very large dataset

- Serial approach requires 1 server with 200GB of Memory
- Parallel approach, divide the dataset in 10 segments of ~20GB each, easier to get

Who is using Parallel Computing?



Where



Why not always use Parallel Computing?

It cannot be done: Not all serial workflows can be parallelized.

A general (simplified) rule: a set of tasks can be executed in parallel only if they can be executed in any arbitrary order.

Time evolution of a given system is a typical example of something that cannot be parallelized.

It is not worth doing: Not all serial workflows can be improved by parallelization.

There are several aspects that could make parallelization unfeasible:

- additional code complexity
- additional computational cost $$$ not worth
- parallelizable part of the workflow too small
- parallelization limits
- …

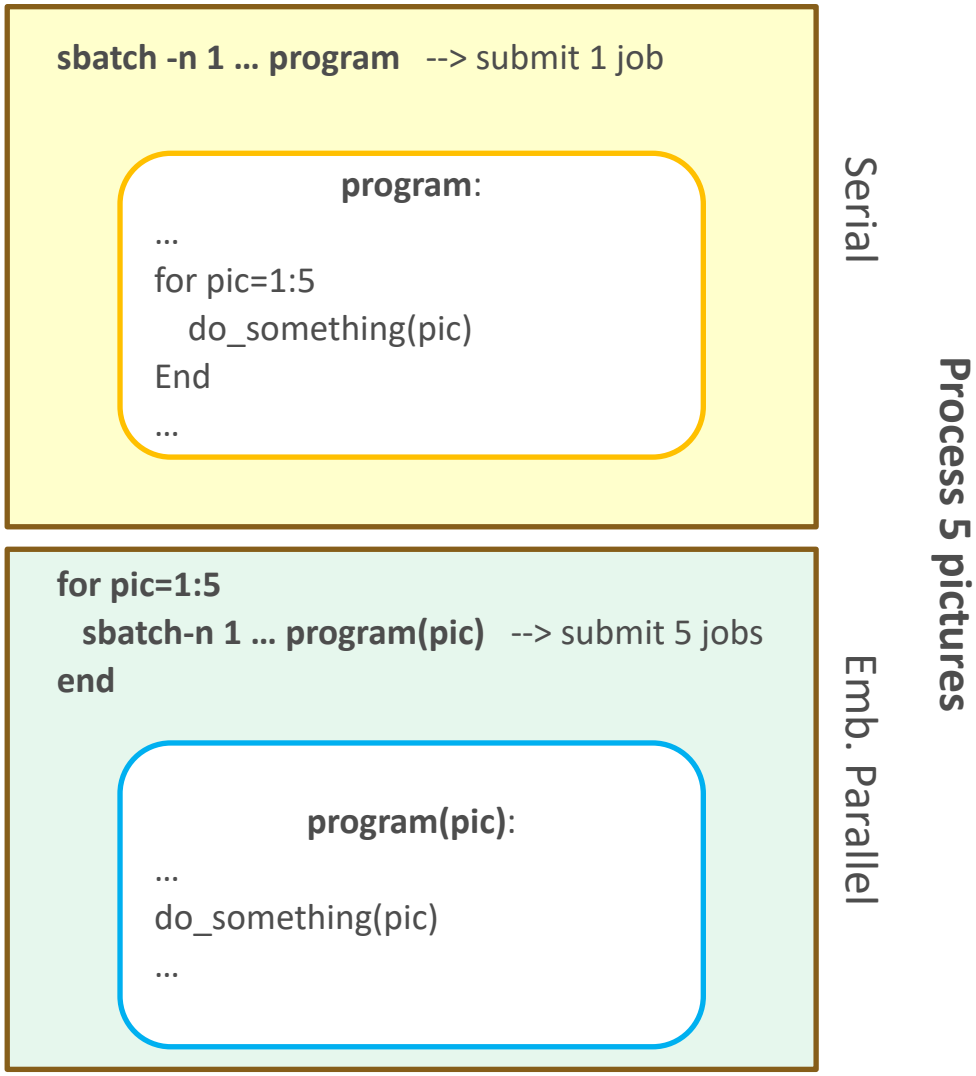
Different types of parallelization

Embarrassingly Parallel:

In this case the entire set of tasks in a given workflow can be executed in any arbitrary order and in a completely independent way.

The tasks can be executed asynchronously and the computational resources are not necessarily required during the same time interval.

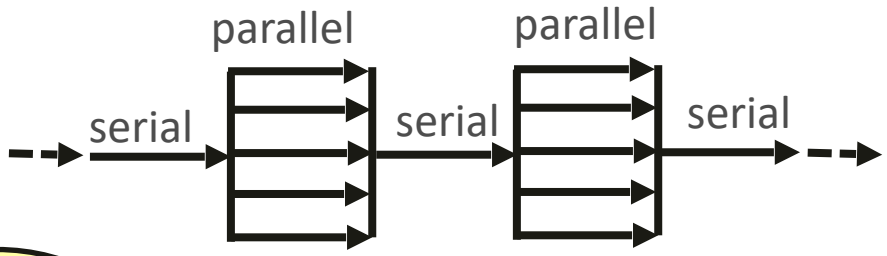
This is approach does not require any real parallelization.



Different types of parallelization

Multithreading and Multitasking parallelization:

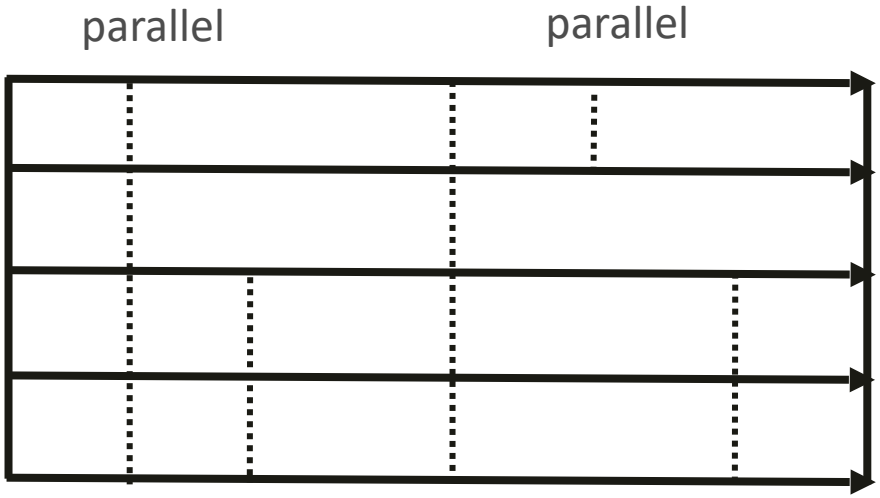
The operations cannot be executed in a fully asynchronous way. Some part of the workflow must be executed serially and/or communication between the different tasks is required



1 Job
5 Cores

These are the most common scenarios in parallel computing.

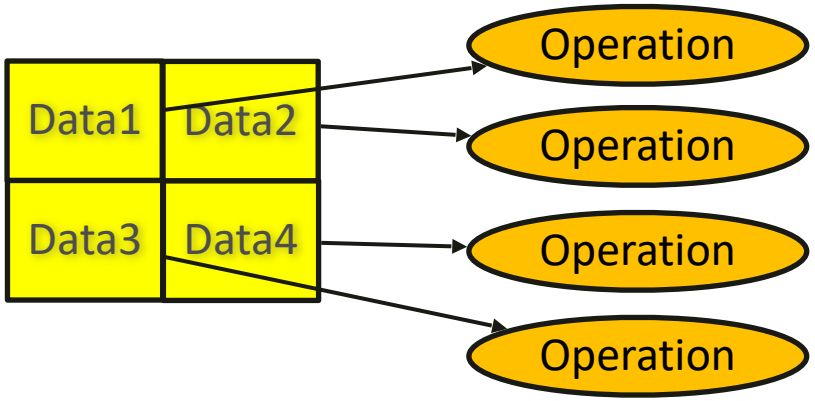
Performance improvement and complexity associated with these types of parallelization are different case by case.



Data Parallelism vs Task Parallelism

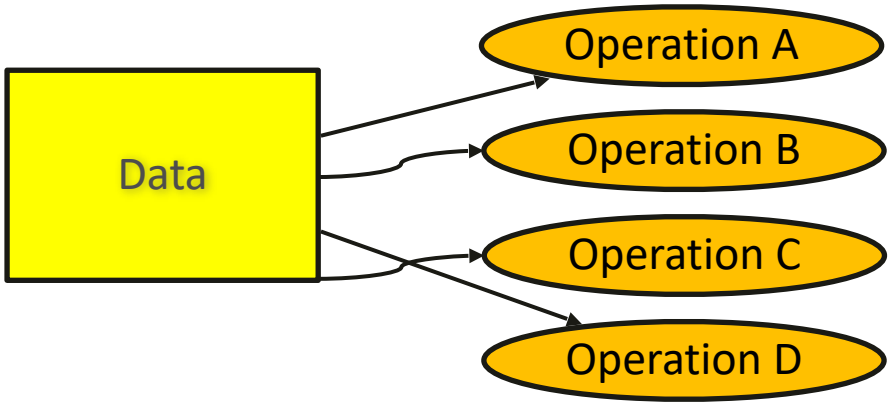
Data Parallelism:

Perform the same type of operation on different parts of the data at the same time



Task Parallelism:

Perform different types of operation on the same data at the same time



Combination of both

Parallelism and Performance

Amdahl's Law

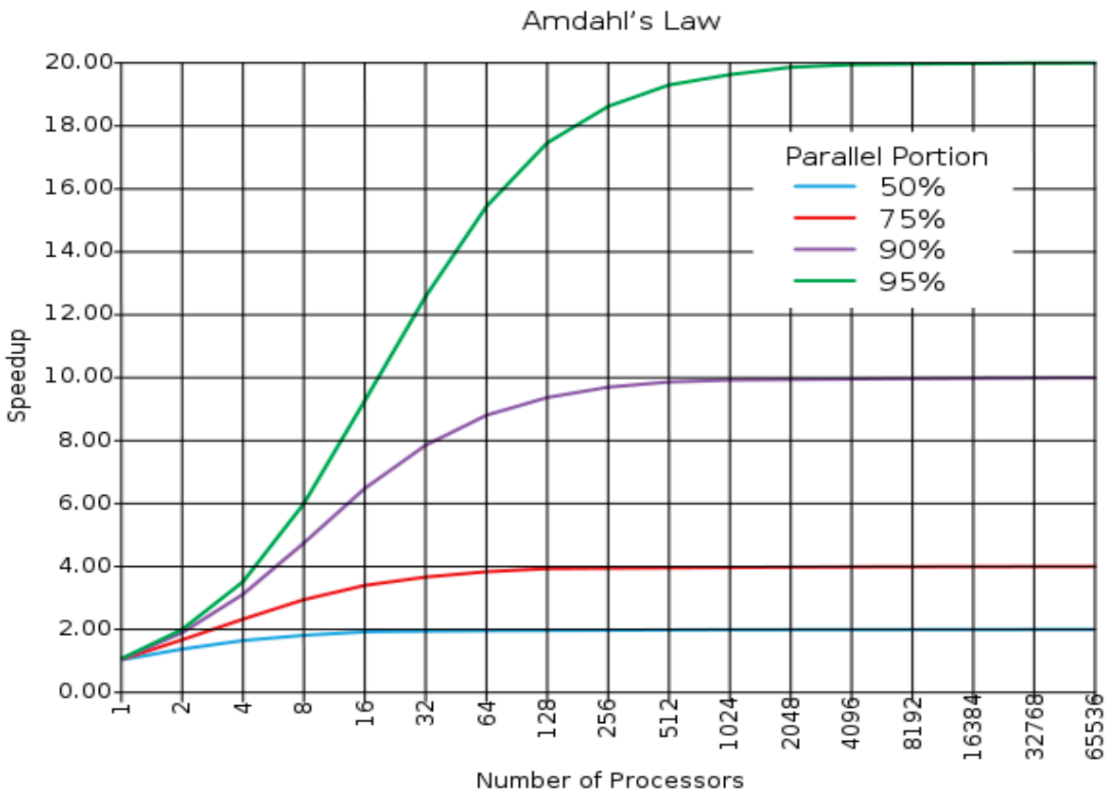
$$SpeedUp = \frac{1}{\frac{P}{N} + S}$$

Where N is the number of processors, P is the parallel time fraction of the code and S is the serial time fraction (P+S=1)

if $N \rightarrow inf$

$$SpeedUp \rightarrow \frac{1}{1 - P}$$

Very ideal case assuming ***0*** costs associated with the parallelization.
This is never the case in real life



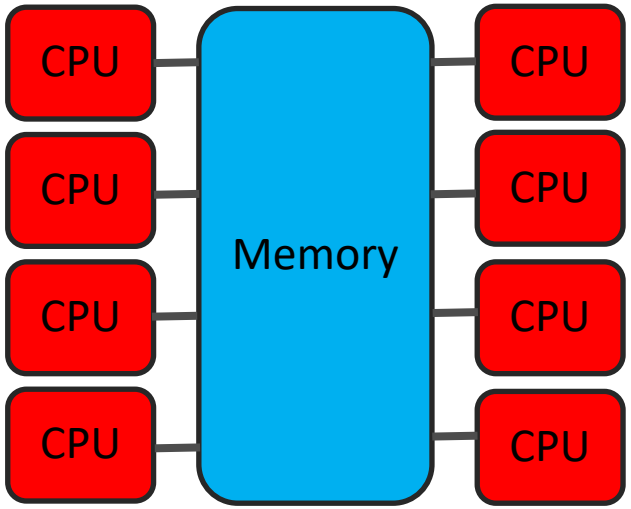
Shared Memory Parallelization

This method of parallelization is used on HW architectures with multiple cores having access to the same memory

- Access to shared memory allows access to shared variables across the different CPUs
- Easy to implement and user friendly

Disadvantages:

- Lack of scalability between memory and CPUs (traffic on memory-cpu communication)
- Race condition / memory synchronization
- Expensive HW
- Limited to one node



Most Common Method:
OpenMP (Open Multi-Processing)

Distributed Memory Parallelization

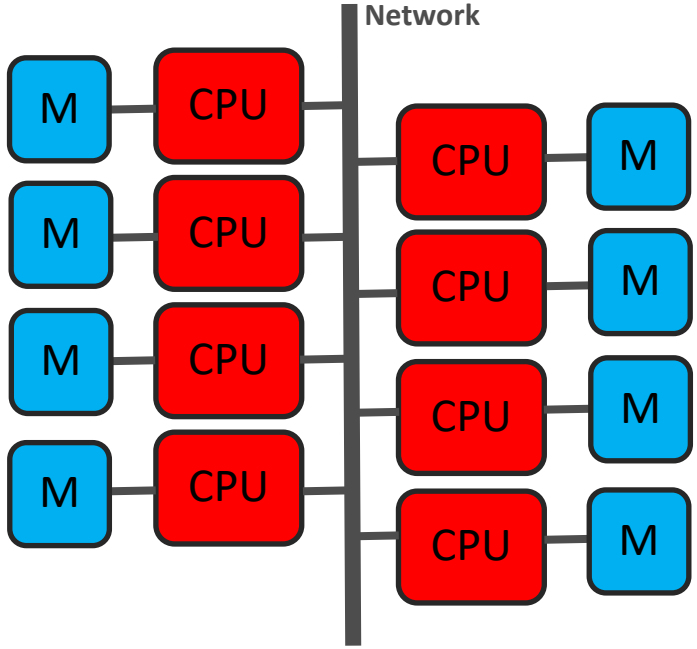
This method of parallelization is used on distributed memory HW architectures using interconnecting communication network. Processors have their own memory.

Advantages :

- More scalable both in term of Memory and CPUs
- Independent cpu-memory access, less overhead
- More versatile

Disadvantages:

- Significantly more complex to implement
- Communication must be handled explicitly
- Additional networking cost for high performance architectures



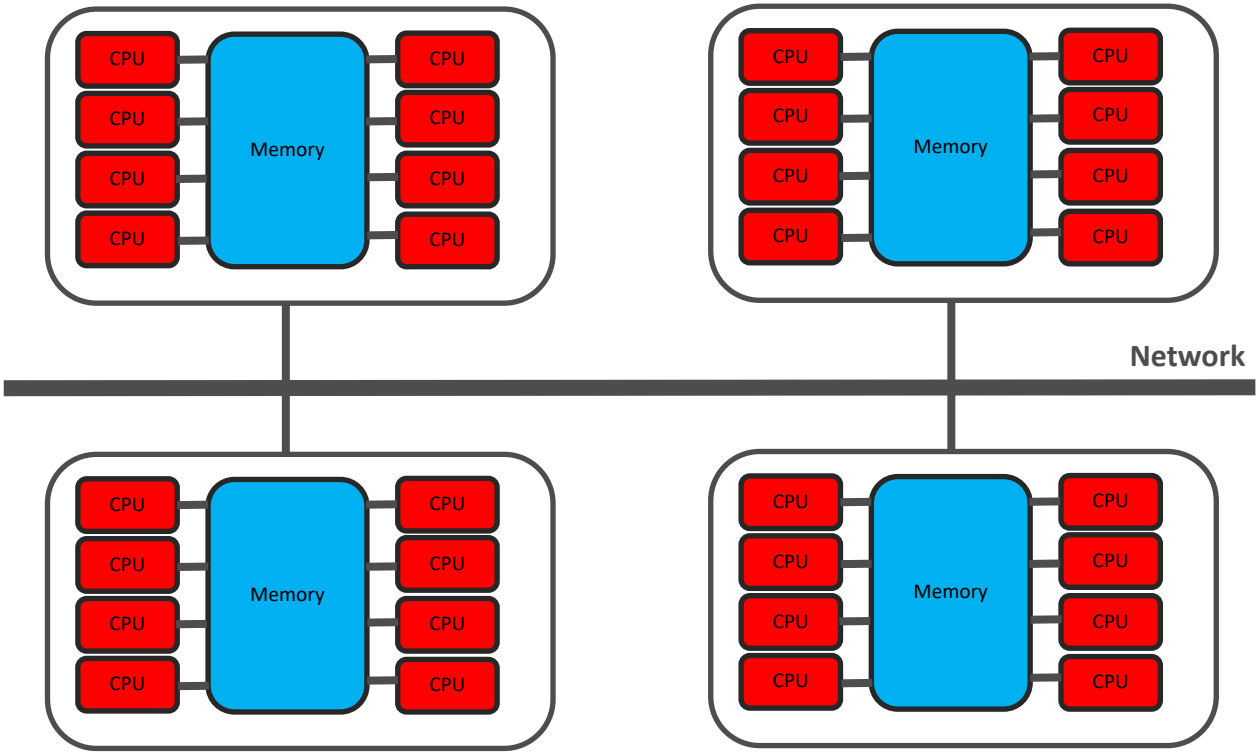
Most Common Method:

MPI (Message Passing Interface)

OpenMPI, MPICH

Mixed Memory Parallelization Model

Combine together the concept of shared and distributed memory parallelization



OpenMP

It is fairly easy to implement, basically "*just*" need to add ***pragma*** lines to the original code in order to create multithreaded segments.
OpenMP pragma starts with the comment symbol so they are compiled only if the appropriate flag is used at compilation time

Programming Languages

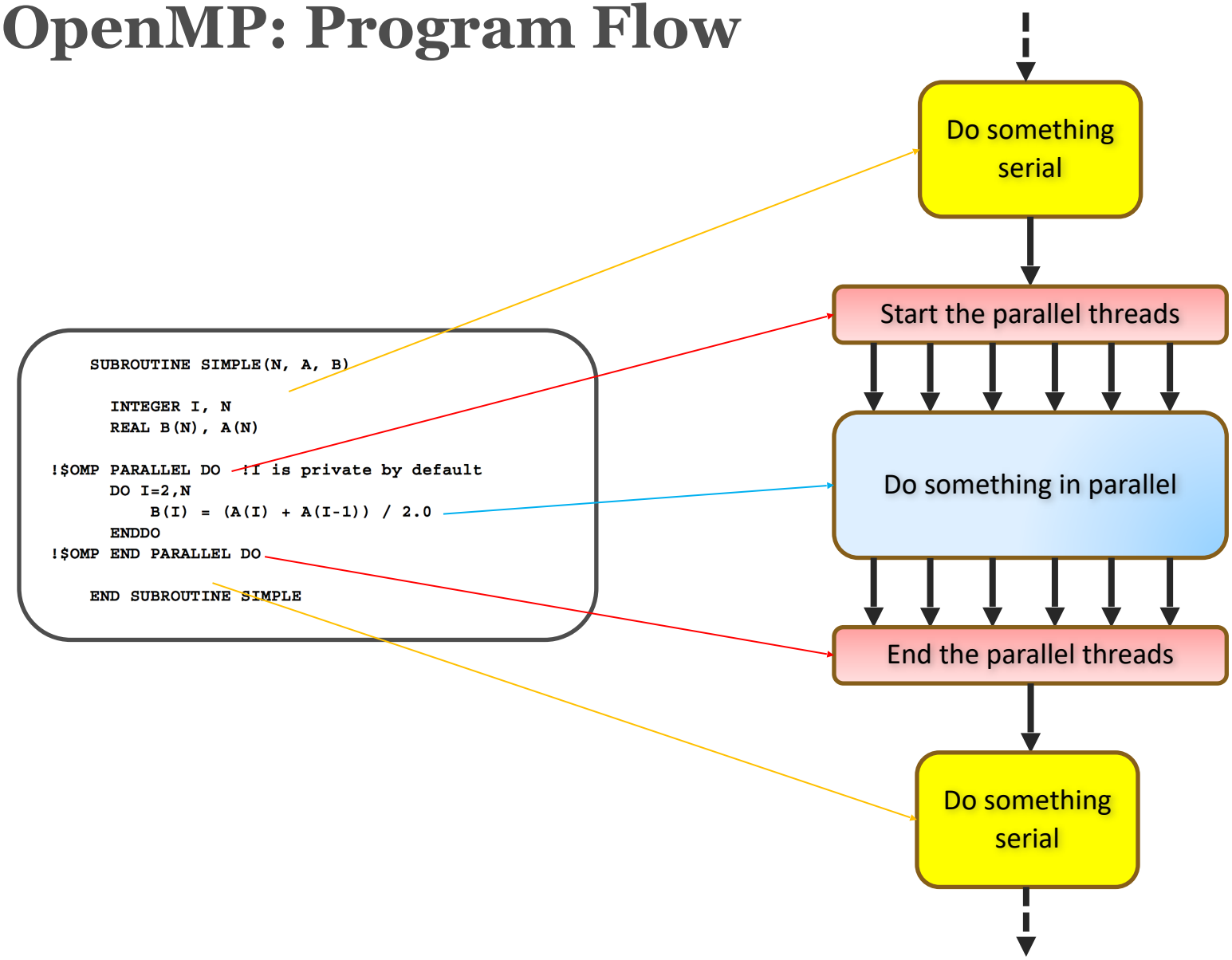
OpenMP is fully available for C/C++ and Fortran for the most used compilers GNU, Intel, Portland.
Multithreading modules are available in Python and R.

OpenMP is often used in "pre-built" modules, libraries or extensions for languages like R and Python.

```
void simple(int n, float *a, float *b)
{
    int i;

#pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

```
    SUBROUTINE SIMPLE(N, A, B)

      INTEGER I, N
      REAL B(N), A(N)

!$OMP PARALLEL DO  !I is private by default
      DO I=2,N
          B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
!$OMP END PARALLEL DO

    END SUBROUTINE SIMPLE
```

OpenMP: Program Flow



OpenMP: Directives

- **Parallel:** Forms a team of threads and starts parallel execution

  !$omp parallel [clause[[,]clause] ...]

  structured-block

  !$omp end parallel

- **Do:** Specifies that the iterations of associated loops will be executed in parallel by threads in the team

  !$omp do [clause[[,]clause] ...]

  do-loops

  !$omp end do

- **Section:** A non iterative work sharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team.

  !$omp sections [clause[[,] clause] ...]

  !$omp section

  Structured-block

  !$omp section

  structured-block

  !$omp end sections

- **Single:** Specifies that the associated structured block is executed by only one of the threads in the team.

- **ordered:** Specifies a structured block in a loop region that will be executed in the order of the loop iterations.

- **Master:** Specifies a structured block that is executed by the master thread of the team.

- **Critical:** Restricts execution of the associated structured block to a single thread at a time.

- **Barrier:** Placed only at a point where a base language statement is allowed, this directive specifies an explicit barrier at the point at which the construct appears.

- **Atomic:** identifies a specific memory location that must be updated atomically, and not be exposed to multiple, writing threads (useful for code that requires fine-grain synchronization)

- **Several more…**

OpenMP: Clauses, Libraries, Variables

Clauses

- **default**(**private** | **firstprivate** | **shared** | **none**)
  Explicitly determines the default data-sharing attributes of variables that are referenced in a **parallel**, **task**, or **teams** construct
- **shared**(*list*)
  Declares one or more list items to be shared by tasks generated by a **parallel**, **task**, or **teams** construct.
- **private**(*list*)
  Declares one or more list items to be private to a task or a SIMD lane.
- **firstprivate**(*list*)
  Declares list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.
- **lastprivate**(*list*)
  Declares one or more list items to be private to an implicit task or to a SIMD lane, and causes the corresponding original list item to be updated after the end of the region.
- …

Libraries

- **omp_set_num_threads:** Affects the number of threads used for subsequent parallel regions
- **omp_get_num_threads:** Returns the number of threads in the current team
- **omp_get_thread_num:** Returns the thread number of the calling thread
- …

Env. Variables

- **OMP_NUM_THREADS:** Sets the *nthreads-var* ICV for the number of threads to use for **parallel** regions
- …

OpenMP: Some Important Info

- **Scope of Variables**:

  Programmer must handle carefully private and shared variables scope inside multithreaded sections of the code.

- **Performance/ CPU Scalability:**

  All CPUs are accessing the same memory, using “too many” cores might result in overall performance degradation. This is highly depending on the tasks being parallelized.

- **Race Condition:**

  Programmer must handle carefully memory access synchronization as needed to avoid erroneous results.

```
#pragma omp parallel for
for ( int i = 0; i < 10000000; i++ )
{
    x = x + 1;
}
```

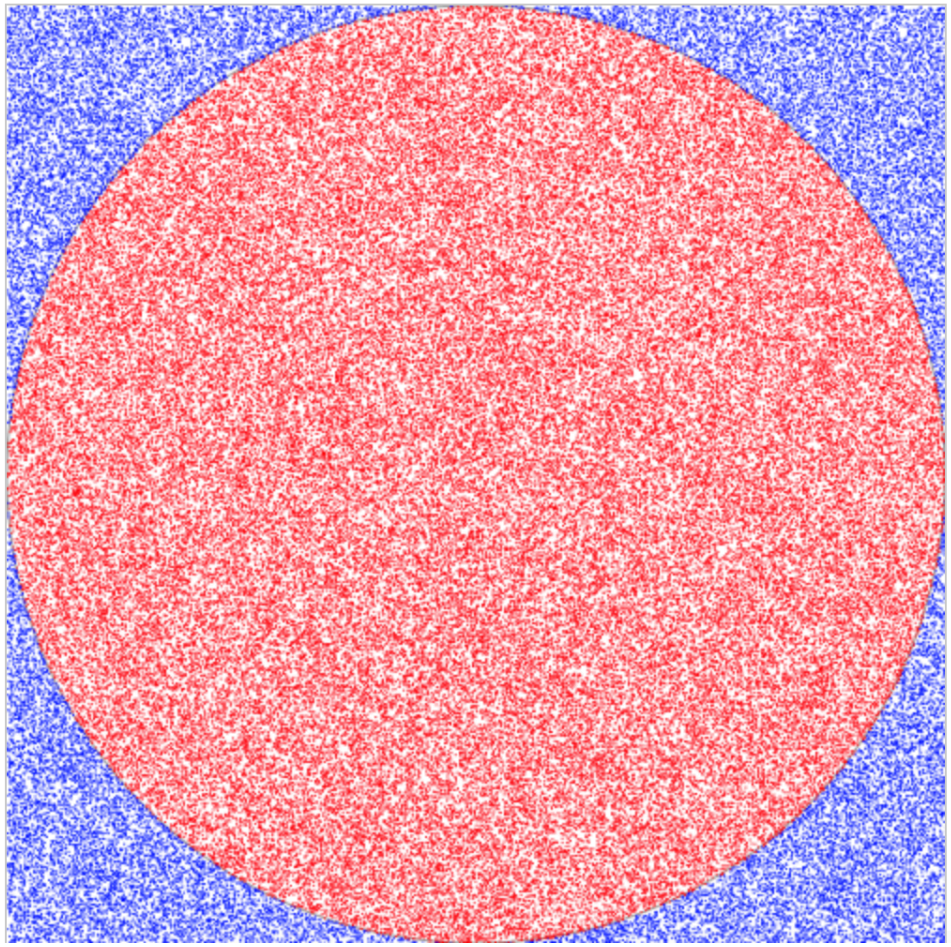
- **Cost of Parallelization:**

  Creation and destruction of parallel threads has a computational cost. Almost never parallelize the 2nd+ loop in a concatenated loops sequence

Example: Calculate PI

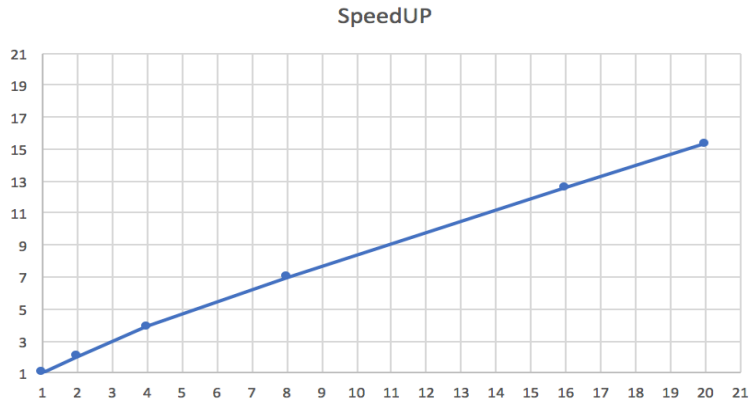
Serial Code

```
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
```



OpenMP Example: Calculate PI

```
rp189@compute-a-16-69:C gcc -fopenmp pi_openmpi.c -o PI
rp189@compute-a-16-69:C ./PI 500000000 1
Estimate of pi: 3.14152
Time:    9.20
rp189@compute-a-16-69:C ./PI 500000000 2
Estimate of pi: 3.14160
Time:    4.59
rp189@compute-a-16-69:C ./PI 500000000 4
Estimate of pi: 3.14159
Time:    2.38
rp189@compute-a-16-69:C ./PI 500000000 8
Estimate of pi: 3.14151
Time:    1.32
rp189@compute-a-16-69:C ./PI 500000000 16
Estimate of pi: 3.14157
Time:    0.73
rp189@compute-a-16-69:C ./PI 500000000 20
Estimate of pi: 3.14163
Time:    0.60
rp189@compute-a-16-69:C
```




```c
/*
 *  * OpenMP implementation of Monte Carlo pi-finding algorithm
 *   * (based on an example in Chap. 17 of Quinn (2004))
 *    *
 *     * usage: pi <samples> <threads>
 *      */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main (int argc, char *argv[])
{
  int i, count, samples, nthreads, seed;
  struct drand48_data drand_buf;
  double x, y;
  double t0, t1;

  samples  = atoi(argv[1]);
  nthreads = atoi(argv[2]);
  omp_set_num_threads (nthreads);

  t0 = omp_get_wtime();
  count = 0;

#pragma omp parallel private(i, x, y, seed, drand_buf) shared(samples)
{
  seed = 1202107158 + omp_get_thread_num() * 1999;
  srand48_r (seed, &drand_buf);

#pragma omp for reduction(+:count)
  for (i=0; i<samples; i++) {
    drand48_r (&drand_buf, &x);
    drand48_r (&drand_buf, &y);
    if (x*x + y*y <= 1.0) count++;
  }
}

  t1 = omp_get_wtime();
  printf("Estimate of pi: %7.5f\n", 4.0*count/samples);
  printf("Time: %7.2f\n", t1-t0);
}
```

MPI

It is fairly complex to implement. Each MPI task executes the same code, it is up to the programmer to partition the computational domain, to handle what each process should do, when communication between the different tasks must happen, what should be communicated and how that should happen.

Programming Languages

MPI is available for several programming languages including C/C++, Fortran, R, Python and Matlab

```
program hello

 implicit none
 include 'mpif.h'
 integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)
 integer INFO
 character*24 hostname,PORT_NAME,jobid


 call MPI_INIT(ierror)
 call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
 call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)


  print *, 'Hello world from rank ',rank, 'of ',size


 call MPI_FINALIZE(ierror)
end
```

```
#!/usr/bin/env python

# IMPORT EVERYTHING HERE

from mpi4py import MPI
import os
import socket
import time
# CREATE FUNCTIONS AND/OR CLASSES HERE


# MAIN CODE
def main():


    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()

    node=socket.gethostname()
    print "hello with rank",rank, "from host ", node
    MPI.Finalize()
if __name__ == '__main__':
    main()
```

MPI

MPI jobs are usually started with the command ***mpirun*** or ***mpiexec***. This command initiates a chain of ***Np*** processes where the MPI code is executed.

For example the python and Fortran code in the previous slide would produce:

```
rp189@login03:PYTHON_CODE mpirun -np 4 ./mpihello.py
hello with rank 3 from host  compute-a-16-77.o2.rc.hms.harvard.edu
hello with rank 0 from host  compute-a-16-74.o2.rc.hms.harvard.edu
hello with rank 1 from host  compute-a-16-75.o2.rc.hms.harvard.edu
hello with rank 2 from host  compute-a-16-76.o2.rc.hms.harvard.edu
rp189@login03:PYTHON_CODE
```

```
rp189@login03:FORTRAN mpirun -np 4 ./hello
 Hello world from rank           3 of           4
 Hello world from rank           2 of           4
 Hello world from rank           0 of           4
 Hello world from rank           1 of           4
rp189@login03:FORTRAN
```

Note that each MPI task executes the same code, however each task is aware of its own identity and of its siblings

Important:

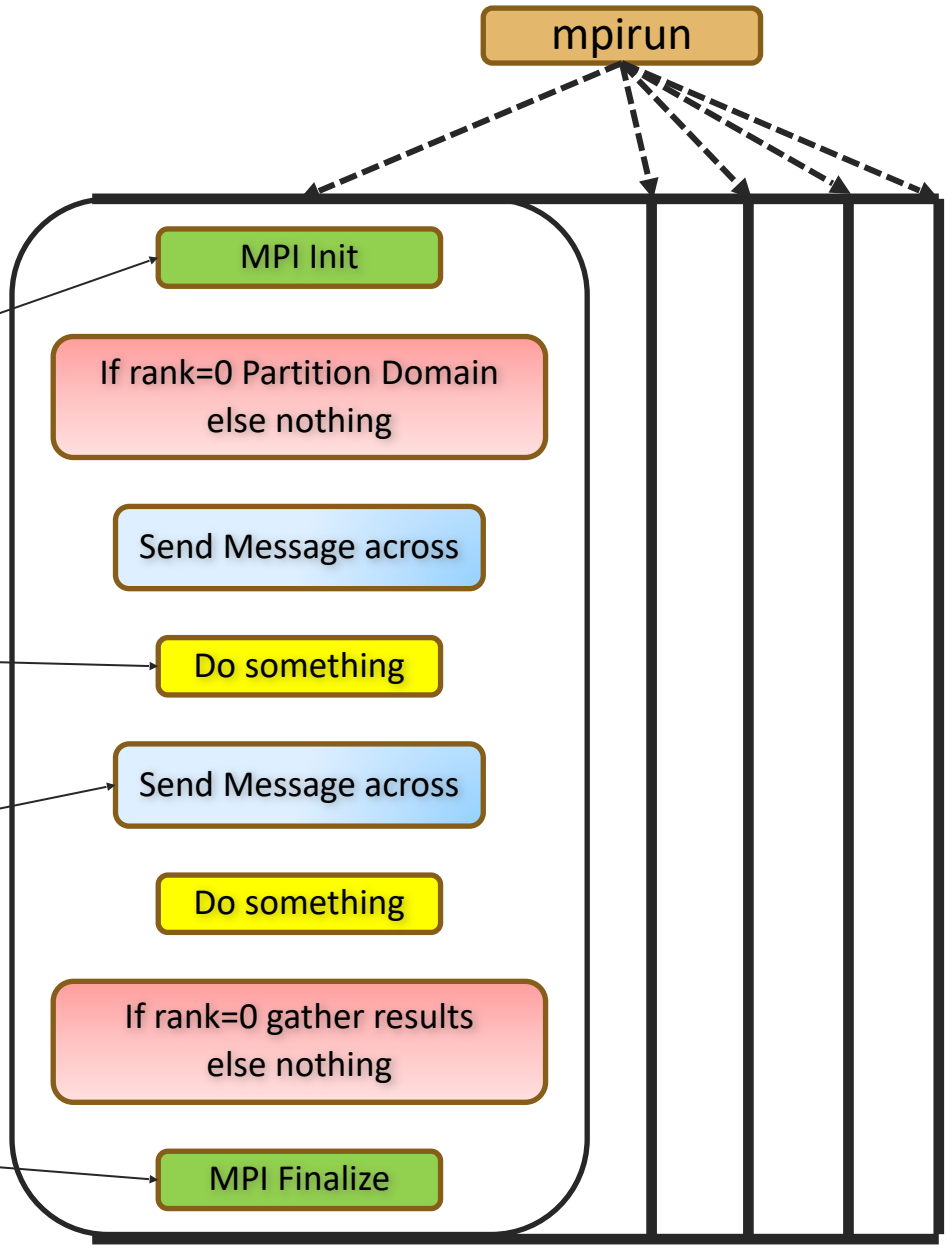
mpirun –np X code will start X "copies" of ***code*** whether ***code*** is an actually MPI-compatible or not. If you use mpirun to start any non-MPI code you will have no performance speedup and your output files will most likely be wrong.

MPI: Most Important Routines

- **MPI_Init():** Initiate an MPI communicator
- **MPI_Finalize():** Terminate the MPI communicator and clean up
- **MPI_Comm_size():** Number of the MPI processes in the communicator
- **MPI_Comm_rank():** Which one am I? Ranks start from 0, called the master process
- **MPI_Send():** Send a message to another process in the MPI communicator
- **MPI_Receive():** Receive a message from another process in the MPI communicator
- **MPI_ISend():** Same as Send but does not wait for message to be "safely sent" (buffer)
- **MPI_IReceive():** Same as Receive but does not wait for message to be received
- **MPI_Abort():** Terminates all MPI processes associated with the communicator
- **MPI_Barrier():** Creates a barrier synchronization for the processes in the communicator
- **MPI_Bcast ():** Broadcasts a message to all other processes in the communicator
- **MPI_Reduce ():** Applies a reduction operation on all tasks in the communicator and places the result in one task.

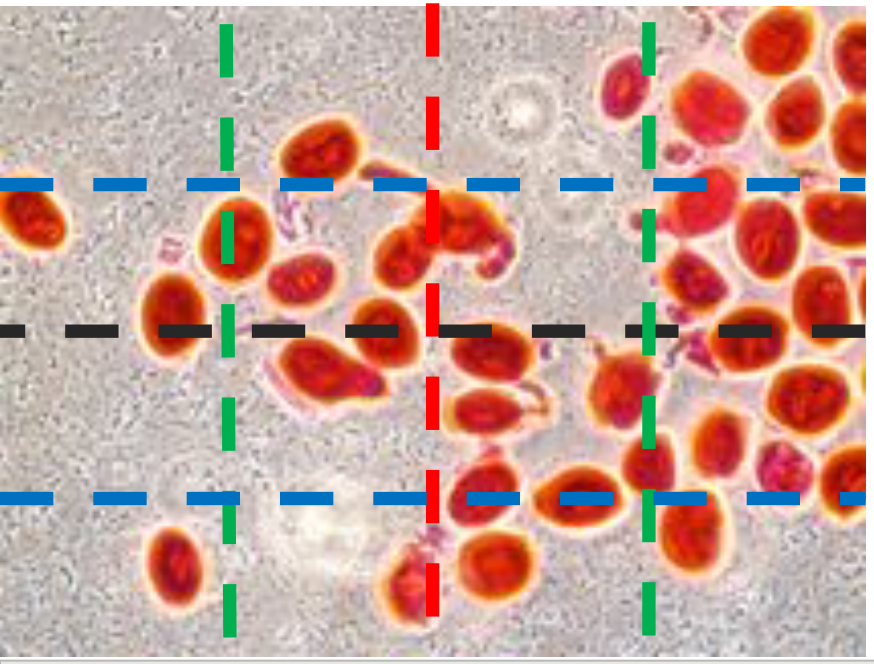
MPI: Program Flow

```
#include "mpi.h"

int main(int argc,char *argv[])
{
    int numprocs,myid,tag,source,destination,count,buffer;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1234;
    source=0;
    destination=1;
    count=1;

    if(myid == source){
        buffer=5678;
        MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
        printf("processor %d sent %d\n",myid,buffer);
    }
    if(myid == destination){
        MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
        printf("processor %d got %d\n",myid,buffer);
    }
    MPI_Finalize();
}
```



MPI: Domain Decomposition and Load Balancing

- The total number of tasks to be executed must be explicitly divided and communicated across the processes in the MPI communicator.
- Basic Approach: Divide the domain uniformly if the operations to be performed by each processors are the same. Balance the load across the processors
- Domain decomposition can get extremely complex and it can strongly affect the overall speedup performances.



Np = 2

Np = 4

Np = 8

Np = 16

Operation
Identify cells and perform a computationally intense task for each one

MPI: Some Important Info

- **Domain Decomposition:**

  When using MPI parallelization it is always up to the Programmer to decompose the domain

- **(Dynamic) Load balancing:**

  Sometime it is necessary to rebalance dynamically the load across the different processors

- **Communication:**

  Communication is always up to the Programmer and if not done carefully it can easily become the performance bottleneck in the parallelization.

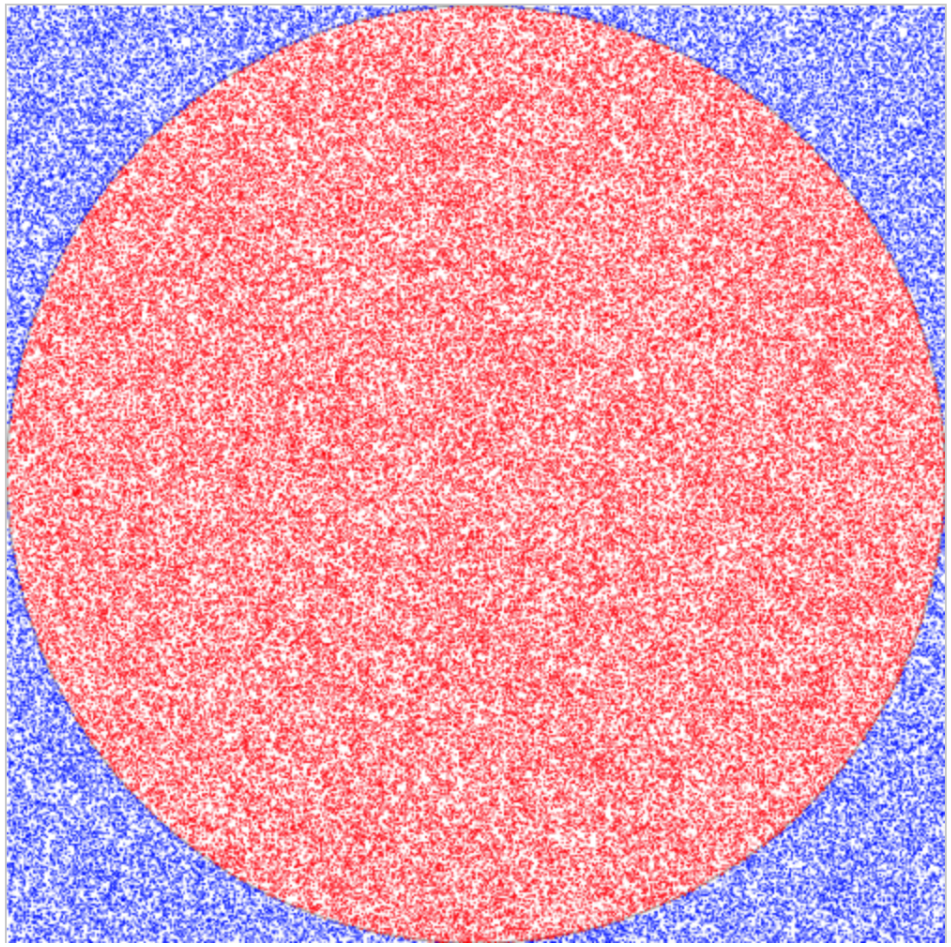
- **Homogenous HW**

  MPI program should always be executed on homogenous architecture to prevent load imbalances due to different HW performances

Example: Calculate PI

Serial Code

```
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
```



MPI Example: Calculate PI

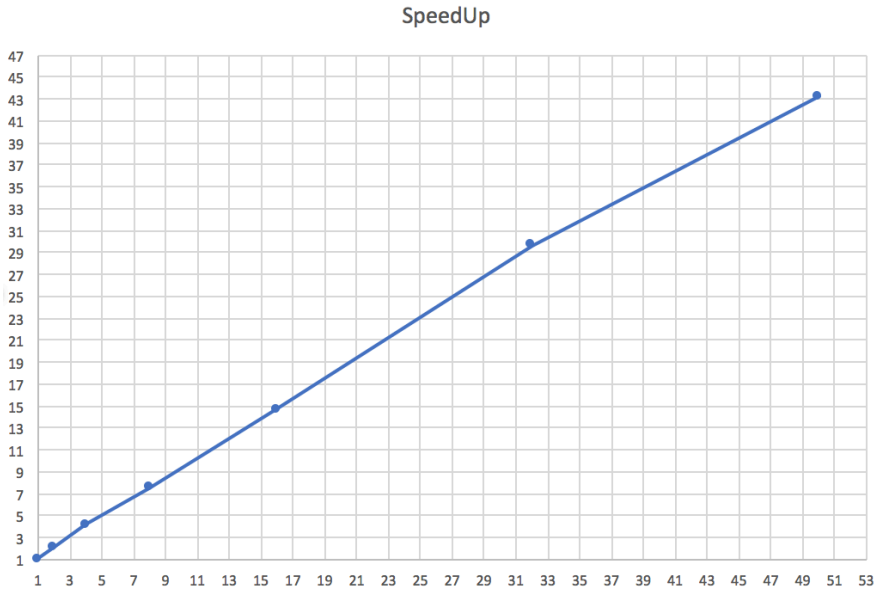
```
      program pi_reduce
      implicit none
      include 'mpif.h'

      integer ROUNDS,TOTROUNDS, MASTER,ierr
      parameter(TOTROUNDS = 500000000)
      parameter(MASTER = 0)

      integer     taskid, numtasks, i, status(MPI_STATUS_SIZE)
      real*4    seednum
      real*8     homepi, pi, avepi,starttime,endtime
      integer   score, n,pisum
      real*4    r
      real*8    x_coord, x_sqr, y_coord, y_sqr
      external  d_vadd

C     Obtain number of tasks and task ID
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, taskid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numtasks, ierr )

        if (taskid .eq. MASTER) then
         starttime=MPI_WTIME()
         ROUNDS=TOTROUNDS/numtasks
         print *, 'rounds for every processors ',ROUNDS
        endif
      call MPI_BCAST(ROUNDS,1,MPI_INTEGER,MASTER,MPI_COMM_WORLD,ierr)
      score = 0
      call init_random_seed()
      do 40 i = 1, ROUNDS

        call random_number(r)
        x_coord = (2.0 * r) - 1.0
        x_sqr = x_coord * x_coord

        call random_number(r)
        y_coord = (2.0 * r) - 1.0
        y_sqr = y_coord * y_coord
```

```
C       if dart lands in circle, increment score
        if ((x_sqr + y_sqr) .le. 1.0) then
         score = score + 1
        endif
40    continue
        call MPI_BARRIER(MPI_COMM_WORLD,ierr)

        call MPI_REDUCE( score, pisum, 1, MPI_INTEGER,
     &                   MPI_SUM, MASTER, MPI_COMM_WORLD, ierr )

        if (taskid .eq. MASTER) then
         pi = 4.0 * pisum / (ROUNDS*numtasks)
         print *, 'calculated pi is ',pi
         endtime=MPI_WTIME()
         print *, 'Overall Time Elapsed ',endtime-starttime
        endif
      call MPI_FINALIZE(ierr)
      end

C-------------------------------------------------------------
          SUBROUTINE init_random_seed()
            INTEGER :: i, n, clock
            INTEGER, DIMENSION(:), ALLOCATABLE :: seed

            CALL RANDOM_SEED(size = n)
            ALLOCATE(seed(n))

            CALL SYSTEM_CLOCK(COUNT=clock)

            seed = clock + 37 * (/ (i - 1, i = 1, n) /)
            CALL RANDOM_SEED(PUT = seed)

            DEALLOCATE(seed)
          END SUBROUTINE
~
```

MPI Example: Calculate PI

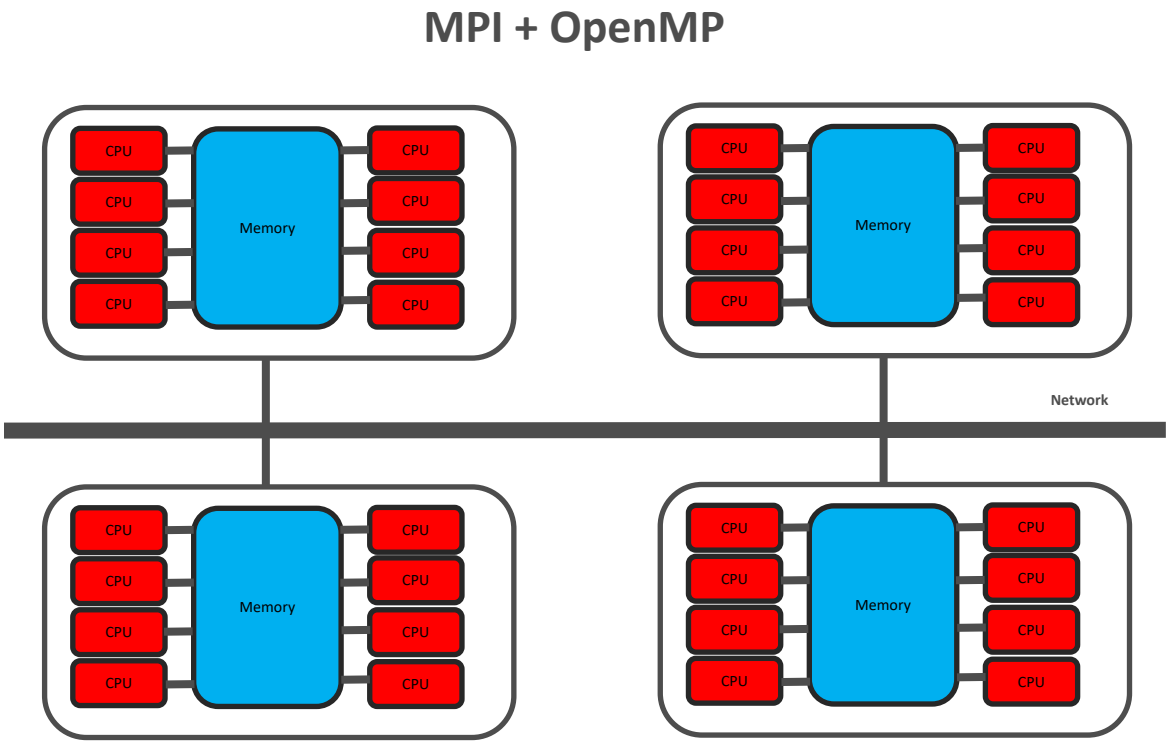
```
rp189@login04:FORTRAN mpif90 mpi_pi_reduce.f -o mpi_pi_reduce
rp189@login04:FORTRAN ./mpi_pi_reduce
 rounds for every processors     500000000
 calculated pi is     3.1415178775787354
 Overall Time Elapsed     25.863259488716722
rp189@login04:FORTRAN mpirun -np 2 ./mpi_pi_reduce
 rounds for every processors     250000000
 calculated pi is     3.1415843963623047
 Overall Time Elapsed     12.149118419969454
rp189@login04:FORTRAN mpirun -np 4 ./mpi_pi_reduce
 rounds for every processors     125000000
 calculated pi is     3.1418721675872803
 Overall Time Elapsed     6.2974237260641530
rp189@login04:FORTRAN mpirun -np 8 ./mpi_pi_reduce
 rounds for every processors      62500000
 calculated pi is     3.1416537761688232
 Overall Time Elapsed     5.5295753739774227
rp189@login04:FORTRAN mpirun -np 16 ./mpi_pi_reduce
 rounds for every processors      31250000
 calculated pi is     3.1412007808685303
 Overall Time Elapsed     1.7667511970503256
rp189@login04:FORTRAN mpirun -np 32 ./mpi_pi_reduce
 rounds for every processors      15625000
 calculated pi is     3.1413545608520508
 Overall Time Elapsed    0.87170137208886445
rp189@login04:FORTRAN mpirun -np 50 ./mpi_pi_reduce
 rounds for every processors      10000000
 calculated pi is     3.1412568092346191
 Overall Time Elapsed    0.59864780399948359
rp189@login04:FORTRAN
```

```
rp189@login04:FORTRAN mpirun -np 8 ./mpi_pi_reduce
 rounds for every processors      62500000
 calculated pi is     3.1414322853088379
 Overall Time Elapsed     3.4332790799671784
rp189@login04:FORTRAN mpirun -np 8 ./mpi_pi_reduce
 rounds for every processors      62500000
 calculated pi is     3.1414628028869629
 Overall Time Elapsed     3.4477858709869906
```



Mixed Memory Parallelization Model

Combine together the concept of shared and distributed memory parallelization



Parallel Computing with Matlab

MathWorks®

parpool()

```
>> help parpool
 parpool Create a parallel pool of workers on a cluster and return a pool object
    pool = parpool creates and returns a pool on the default cluster with its
    NumWorkers in the range [1, preferredNumWorkers] for running parallel
    language features (parfor, parfeval, parfevalOnAll, spmd, and distributed).
    preferredNumWorkers is the value defined in your parallel preferences. Use
    delete(pool) to shut down the parallel pool.

    pool = parpool(numWorkers) creates and returns a pool with the
    specified number of workers. numWorkers can be a positive integer or a
    range specified as a 2-element vector of integers. If numWorkers is a
    range, the resulting pool has size as large as possible in the range
    requested.
```

parfor

```
>> help parfor
 parfor Execute for loop in parallel on workers in parallel pool
    The general form of a parfor statement is:

        parfor loopvar = initval:endval
            <statements>
        END
```

spmd

```
>> help spmd
 spmd Single Program Multiple Data
    The general form of an spmd statement is:

    spmd
        <statements>
    END
```

parfor behavor is very similar to the *omp parallel do* pragma

spmd is very similar to the mpi

In reality Matlab always uses MPI (MPICH) to manage the parallelism and add a very user friendly structure around it.
This makes implementing a parallel code in Matlab much easier than in other languages

Parallel Computing with Matlab: Example

```
TOTROUNDS=500000000;
%%% SERIAL %%%
tic
score=0;
for dart=1:TOTROUNDS
    X2=((2*rand)-1)^2;
    Y2=((2*rand)-1)^2;
    if (X2+Y2 < 1)
        score=score+1;
    end
end
pi=4*score/TOTROUNDS
toc
```

```
%%% PARFOR %%%
tic
score=0;
parfor dart=1:TOTROUNDS
    X2=((2*rand)-1)^2;
    Y2=((2*rand)-1)^2;
    if (X2+Y2 < 1)
        score=score+1;
    end
end
pi=4*score/TOTROUNDS
toc
```

```
%%% SPMD  %%%
tic
score=0;
spmd
    ROUNDS=int64(TOTROUNDS/numlabs);
    for dart=1:ROUNDS
     X2=((2*rand)-1)^2;
     Y2=((2*rand)-1)^2;
     if (X2+Y2 < 1)
        score=score+1;
     end
    end
    if (labindex ~= 1 )
      labSend(score,1)
    end

   if (labindex == 1)
      for ww=2:numlabs

         score=score+labReceive(ww);
      end
      PI=4*score/TOTROUNDS
   end
end
toc
```

Parallel Computing with Matlab: Example

10 workers in the Matlab pool

Overall Speedup ~ 6 X

```
---- SERIAL ----

pi =

    3.1416

Elapsed time is 22.392642 seconds.
---- PARFOR ----

pi =

    3.1415

Elapsed time is 3.879296 seconds.
---- SPMD ----
Lab  1:

  PI =

     3.1416

Elapsed time is 3.739191 seconds.
>> gcp

ans =

 Pool with properties:

           Connected: true
          NumWorkers: 10
             Cluster: local
       AttachedFiles: {}
         IdleTimeout: 30 minutes (24 minutes remaining)
         SpmdEnabled: true

>>
```

Parallel Computing with Matlab: Local Profile vs O2 Profile

It is possible to use different cluster profiles when starting a parallel pool of workers. By default Matlab uses the ***local*** cluster profile. This will create the parallel pool of worker using the cores available on the local machine where it is running.

However Matlab is also capable of interacting directly with the O2 scheduler (SLURM) when using the ***o2 local*** profile. In this case when a parpool request is launched Matlab interacts directly with the Scheduler and manages CPU resources once those are allocated. This is particularly useful when requesting several cores because it allows for the parallel pool of worker to be span over multiple nodes. Matlab uses MPI, so it is compatible with distributed memory systems. Note that in this case the command parpool actually submits a separate cluster job to get the workers cores

```
>> parcluster

ans =

 Local Cluster

    Properties:

                    Profile: local
                   Modified: false
                       Host: compute-a-16-72
                 NumWorkers: 4
                 NumThreads: 1

            JobStorageLocation: /home/rp189/.matlab/local_cluster_jobs/R2017b
 RequiresMathWorksHostedLicensing: false

    Associated Jobs:

                Number Pending: 0
                 Number Queued: 0
                Number Running: 0
               Number Finished: 0

>> parpool(3)
Starting parallel pool (parpool) using the 'local' profile ...
connected to 3 workers.

ans =

 Pool with properties:

          Connected: true
         NumWorkers: 3
            Cluster: local
      AttachedFiles: {}
    AutoAddClientPath: true
        IdleTimeout: 30 minutes (30 minutes remaining)
        SpmdEnabled: true
```

```
>> parcluster

ans =

 Generic Cluster

    Properties:

                    Profile: o2 local R2017b
                   Modified: false
                       Host: compute-a-16-72
                 NumWorkers: 2500
                 NumThreads: 1

            JobStorageLocation: /home/rp189/MdcsDataLocation/o2/compute-a-16-68.o2.rc.hms.harvard.edu/R2017b/local
             ClusterMatlabRoot: /n/app/matlab/2017b
               OperatingSystem: unix

RequiresMathWorksHostedLicensing: false
      IntegrationScriptsLocation: /n/app/rc-testing/apps-test/matlab/2017b/toolbox/local/IntegrationScripts/o2
            AdditionalProperties: List properties

    Associated Jobs:

                Number Pending: 0
                 Number Queued: 0
                Number Running: 0
               Number Finished: 2

>>
```

```
>> parpool(4)
Starting parallel pool (parpool) using the 'o2 local R2017b' profile ...

additionalSubmitArgs =

    '--ntasks=4 -t 01:00:00 -p mpi'

connected to 4 workers.

ans =

 Pool with properties:

          Connected: true
         NumWorkers: 4
            Cluster: o2 local R2017b
      AttachedFiles: {}
    AutoAddClientPath: true
        IdleTimeout: 30 minutes (30 minutes remaining)
        SpmdEnabled: true
 EnvironmentVariables: {}

>>
```

Parallel Computing with GPU

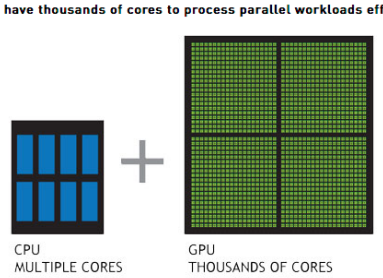
"GPU-accelerated computing is the use of a graphics processing unit (GPU) *together* with a CPU to accelerate deep learning, analytics, and engineering applications"

One of the most commonly used language is **Cuda** (C based) which works only on Nvidia GPU cards.

OpenCL is an alternative open parallel programming language for heterogeneous platform (CPU+GPU) which works with most GPU card available on the market.

OpenACC is a user-driven directive-based performance-portable parallel programming model. Similar to OpenMP but designed to leverage the GPU computing capabilities. Available for C/C++ and Fortran with the PGI compiler.

Matlab also supports GPU parallelization




```cpp
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);

  cudaFree(d_x);
  cudaFree(d_y);
  free(x);
  free(y);
}
```

https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c

Please fill out the survey

- Accessible through the Harvard Training Portal
  - https://trainingportal.harvard.edu/
- Click on “Me” then “Intro to Parallel Computing”
- Scroll to “Evaluations” and click on the survey

- We appreciate any feedback or comments!

Welcome to the Parallel world !

rchelp@hms.harvard.edu