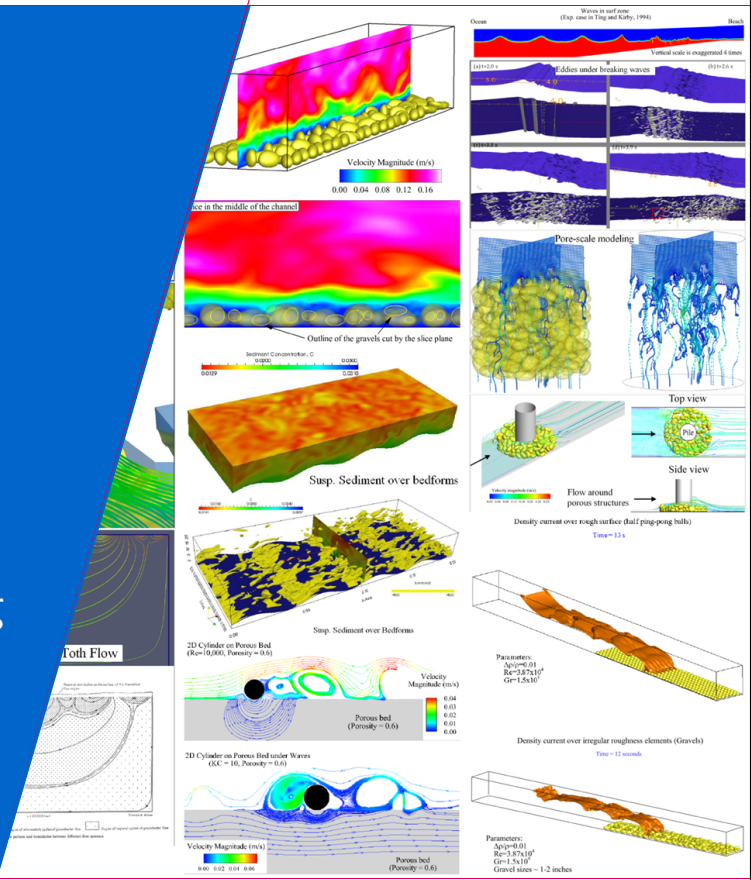
PENNSTATE

1855

CE597D Special Topics, Fall, 2014

# Chapter 8: Parallel Computation

Xiaofeng Liu, Ph.D., P.E.
Assistant Professor
Department of Civil and Environmental Engineering
Pennsylvania State University
xliu@engr.psu.edu



# What will be covered in this chapter?



- General introduction of parallel computation
- Parallel computation in OpenFOAM®

# Outline



General introduction of parallel computation

Parallel computation in OpenFOAM

# General introduction



Why we need parallel computation?

- To run simulations faster
- To handle large data set which might not fit into the memory of single computer
- To run large number of cases, e.g., Monte Carlo simulations

Two major approaches for parallel computation:

- Shared memory process (SMP): OpenMP (Open Multiple Processing)
- Distributed computing: MPI (Message Passing Interface)

New developing trend: GPU

http://www.nvidia.com/object/what-is-gpu-computing.html

# General introduction



What is OpenMP (for shared memory)?

- a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism
- Example: parallel summation of two arrays and assign to a new array

```
!$OMP PARALLEL DO
do i=1,128
   b(i) = a(i) + c(i)
end do
!$OMP END PARALLEL DO
```

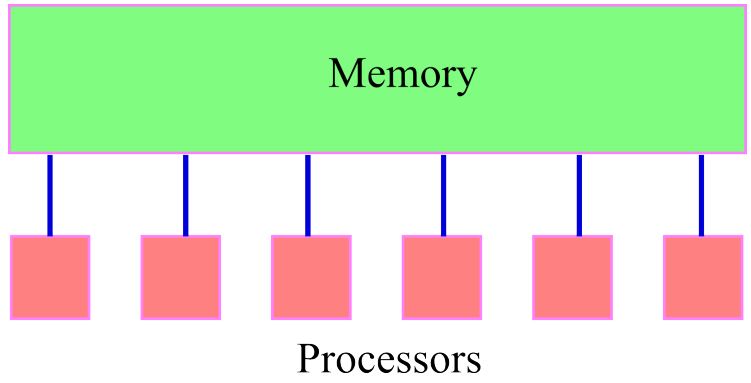



Figure: Shared memory system

# General introduction



What is MPI (for distributed memory)?

- http://www.mcs.anl.gov/research/projects/mpi/
- MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.
- Distributed memory system have separate memory address for each processor
- Data (e.g., domain) must be decomposed: domain decomposition

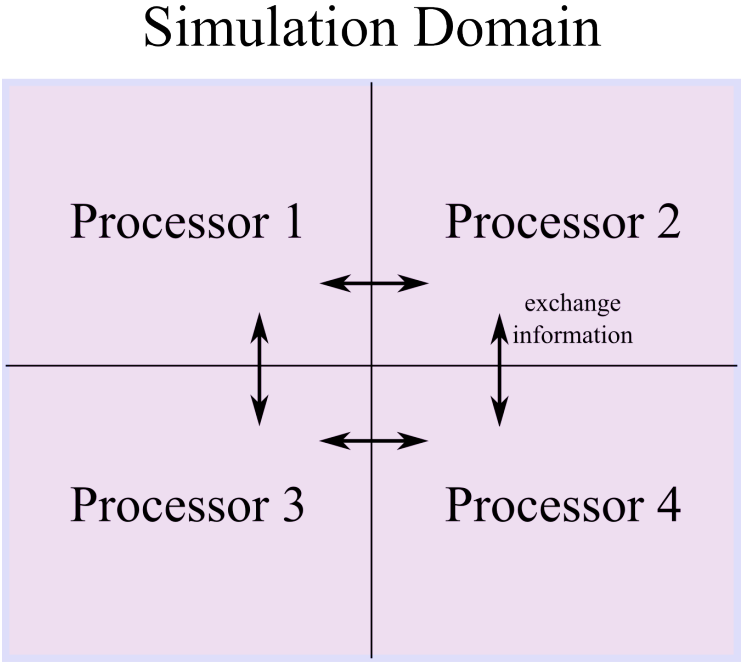



Figure: Domain decomposition

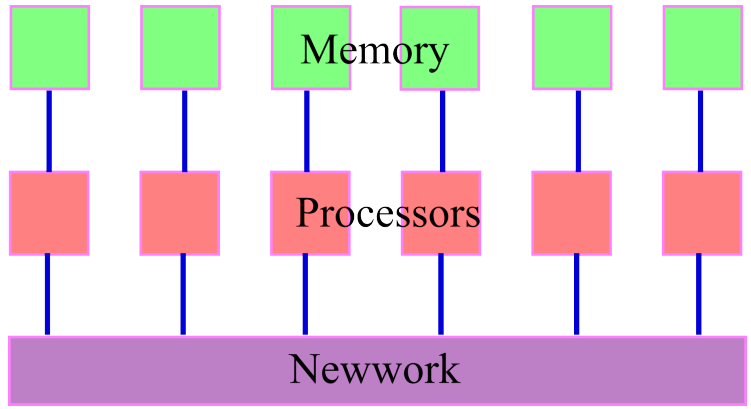



Figure: Distributed memory system

# General introduction



What is MPI (for distributed memory)?

- Communications: MPI programs send and receive data among the processors
  - `MPI_send()` and `MPI_Isend()`
  - `MPI_Recv()` and `MPI_Irecv()`
- Two types of communications:
  - Blocking communication: `MPI_send()` and `MPI_Recv()` function calls do not return (i.e., block) until the communication is finished.
  - Non-blocking communication: `MPI_Isend()` and `MPI_Irecv()` function calls return immediately (i.e., non-block). The program must call `MPI_Wait()` or `MPI_Probe` to synchronize the processors.
  - What is the usage of "non-blocking" communication: The code can issue send/receive command and do some other computations while the communication is going on. To improve parallel performance.
  - In OpenFOAM® , this is controlled through the switch `commsTypes` in `etc/controlDict` file. The default is `nonBlocking`.

# General introduction



Some terminologies for parallel computation:

- HPC or supercomputer: Hard to define nowadays; www.top500.org
- Computing node
- CPU: Central Processing Unit; new CPUs usually have multiple cores

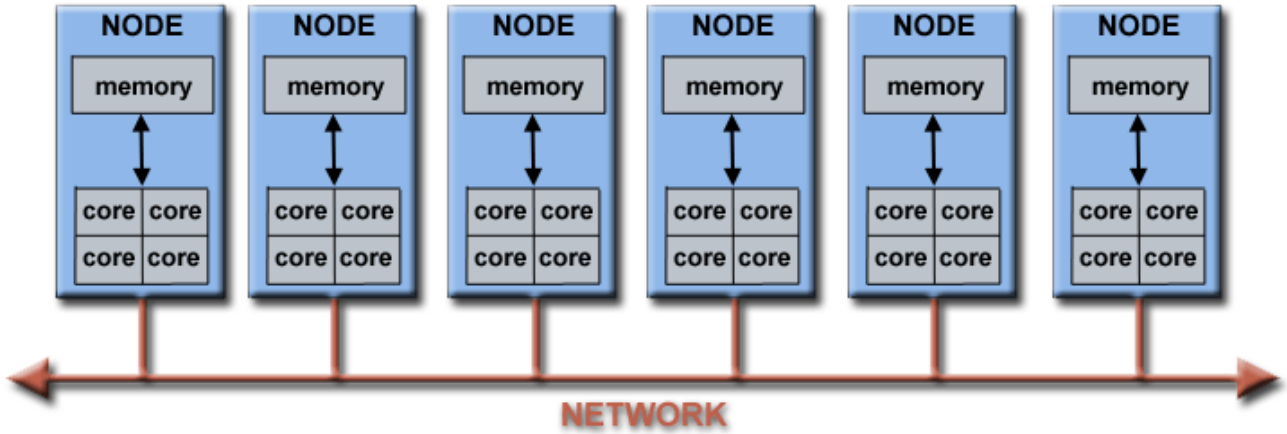



Figure: Simple diagram of a typical parallel computer with multiple computing nodes. Each node has multiple CPUs/Cores and they all share the same pool of memory. However, the memory is not shared with other nodes. Source: llnl.gov

# General introduction



Some terminologies for parallel computation:

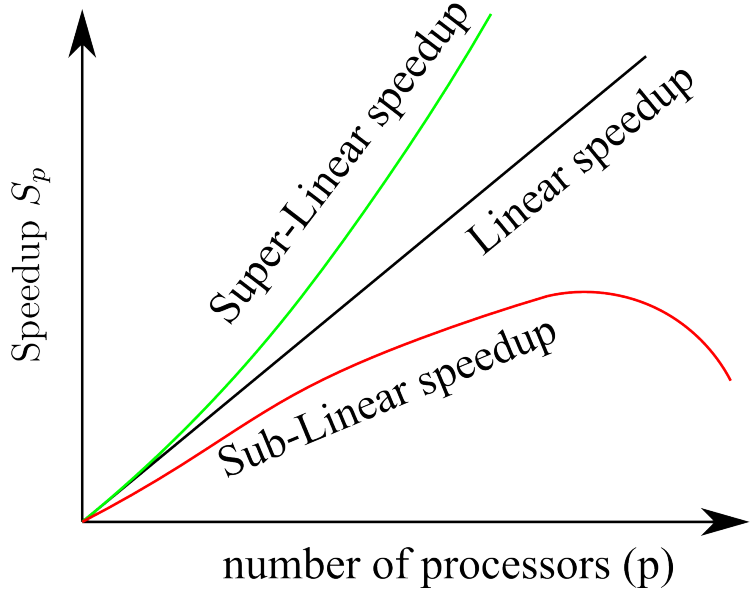
- Speedup:

$$S_p = \frac{T_s}{T_p} \tag{1}$$

  where $p$ is the number of processors, $T_s$ is the computing time using one processor (sequential run), $T_p$ is the computing time using $p$ processors (parallel run).

- Linear speedup (ideal condition):

$$S_p = p \tag{2}$$



# General introduction



Some terminologies for parallel computation:

- Parallel efficiency:

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p} \tag{3}$$

- For linear speedup, $E_p = 1.0$

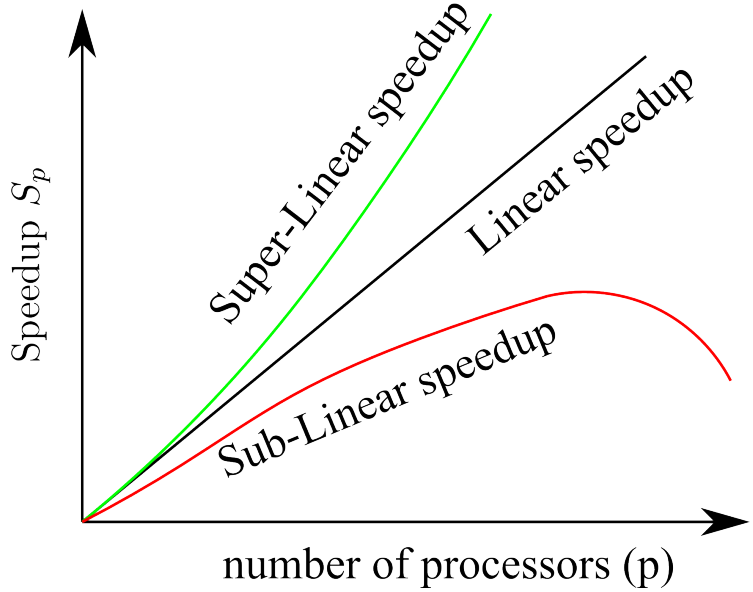



Figure: Speedup of parallel computatition

# General introduction



Factors affecting parallel speedup:

- Startup overhead: prepare parallel computation, reading mesh, build communication topology, prepare information exchange at the boundary, etc.
- Communication cost during the simulation
- Percentage of serial sections in the code: Some operations can only be done by the master processor.
  - Calculation of force on an object: all slave processors send in the partial force in their domain; the master process assemble them together.
- Load balancing

# Parallel Job Scheduler



- A job contains the following information
  - name of the application
  - input and output information
  - computer resources needed: number of CPUs/cores, amount of memory, run time, etc.)
- Job scheduler is a software to manage jobs:
  - allocate the computer resources requested for different jobs; setup priorities for the jobs in the queues (might have many queues)
  - run the job and optionally charge the time to user's account
  - feedback to the user the status and outcome of the job
- Different computer systems might use different job scheduler:
  - Cyberstar: PBS
  - Stampede: SLURM (Simple Linux Utility for Resource Management)
- Usually a job script is written and submitted.

# Parallel Job Scheduler



Example PBS script:

```
#!/bin/bash
#PBS -l nodes=64
#PBS -l walltime=95:59:59
#PBS -N example_run
#PBS -A MyProjectName
#PBS -o outputFileName
#PBS -e errorFileName

#PBS -V

NCPU=‘wc -l < $PBS_NODEFILE‘
NNODES=‘uniq $PBS_NODEFILE | wc -l‘
cd $PBS_O_WORKDIR

echo "Starting at "
date

# Source tutorial run functions
. $WM_PROJECT_DIR/bin/tools/RunFunctions

#define the number of processors for parallel
numProc=$NCPU

mpiexec -np $NCPU pimpleFoam -parallel >> example_run.log


echo "Run completed at "
date
```

Example SLURM script:

```
#!/bin/bash
#SBATCH -J example_job
#SBATCH -o example.o%j
#SBATCH -A TG-CTS14xxxx
#SBATCH -n 64
#SBATCH -p normal
#SBATCH -t 10:10:10
#SBATCH --mail-user=address@engr.psu.edu
#SBATCH --mail-type=begin  # email me when the j
#SBATCH --mail-type=end    # email me when the j

export MV2_ON_DEMAND_THRESHOLD=128

ibrun pimpleFoam -parallel
```

# Job Scheduler



Some tips using PBS:

- use `qstat` to check the status of jobs. It has options for a specific user.
- use `qdel` to delete a submitted job.
- dependent jobs use option `-W depend=afterok`. For example, you can write a shell script:

```
#!/bin/bash
FIRST=‘qsub job_1.sh‘
echo $FIRST
SECOND=‘qsub -W depend=afterok:$FIRST job_2.sh‘
echo $SECOND
THIRD=‘qsub -W depend=afterok:$SECOND job_3.sh‘
echo $THIRD
FOURTH=‘qsub -W depend=afterok:$THIRD job_4.sh‘
echo $FOURTH
exit 0
```

# Job Scheduler



Some tips using PBS:

- dependent jobs are useful is your simulation takes longer than the maximum time allowed in a supercomputer system.
- For example, run long DNS or LES simulations and do `fieldAverage` operation.
- In OpenFOAM® , you can use the following strategy:
  - run your simulation through the transition period till equilibrium; do not enable `fieldAverage` function object
  - Resume the simulation from the stored equilibrium solution and activate the `fieldAverage`. Run sufficiently long such that the statistics do not change anymore.
  - use dependent jobs if necessary
  - If you are only interested in the `fieldAverage`, you can do the following in the `controlDict` file:

```
stopAt nextWrite
writeControl clockTime;
writeInterval 86400;  //the maximum allowed single job time
```

# Parallel computation in OpenFOAM



- We already know from the user level, we can use the following tools:
  - `decomposePar`: controlled through file `decomposeParDict`; it is essentially domain decomposition.
  - `runParallel` script or `mpirun`
  - `reconstructPar`
- Many of the utilities tools can run in parallel too. So it is not necessary to reconstruct the whole domain.
- There are several ways to do domain decomposition. In fact, it self is a research area.
- The aim of the decomposition is to minimize the communication cost by say minimize the sub-domain interface area.
- OpenFOAM® provides several options in file `decomposeParDict`:
  - `simple`: specify *nx*, *ny*, and *nz*
  - `scotch`
  - `metis`
  - ...
- It also provides some options: `preserveFaceZones` and `preservePatches`

# Parallel computation in OpenFOAM



- From more advanced point of view, the majority part for parallel communication is implemented in `src/Pstream`: a wrapper around `MPI` for OpenFOAM®
- The specific implementation of MPI is selected in the `etc/bashrc` file:

```
#- MPI implementation:
#    WM_MPLIB = SYSTEMOPENMPI | OPENMPI | MPICH | MPICH-GM | HPMPI
#               | GAMMA | MPI | QSMPI | SGIMPI
export WM_MPLIB=OPENMPI

...

# Source project setup files
# ~~~~~~~~~~~~~~~~~~~~~~~~~~
_foamSource $WM_PROJECT_DIR/etc/config/settings.sh
_foamSource $WM_PROJECT_DIR/etc/config/aliases.sh
```

# Parallel computation in OpenFOAM



- Based on the selection of `WM_MPLIB`, further setup is done in `etc/config/settings.sh`:

```
case "$WM_MPLIB" in
SYSTEMOPENMPI)
    # Use the system installed openmpi, get library directory via mpicc
    export FOAM_MPI=openmpi-system

    libDir=`mpicc --showme:link | sed -e 's/.*-L\([^ ]*\).*/\1/'`

    # Bit of a hack: strip off 'lib' and hope this is the path to openmpi
    # include files and libraries.
    export MPI_ARCH_PATH="${libDir%/*}"

    _foamAddLib     $libDir
    unset libDir
    ;;

OPENMPI)
    export FOAM_MPI=openmpi-1.6.3
    # optional configuration tweaks:
```

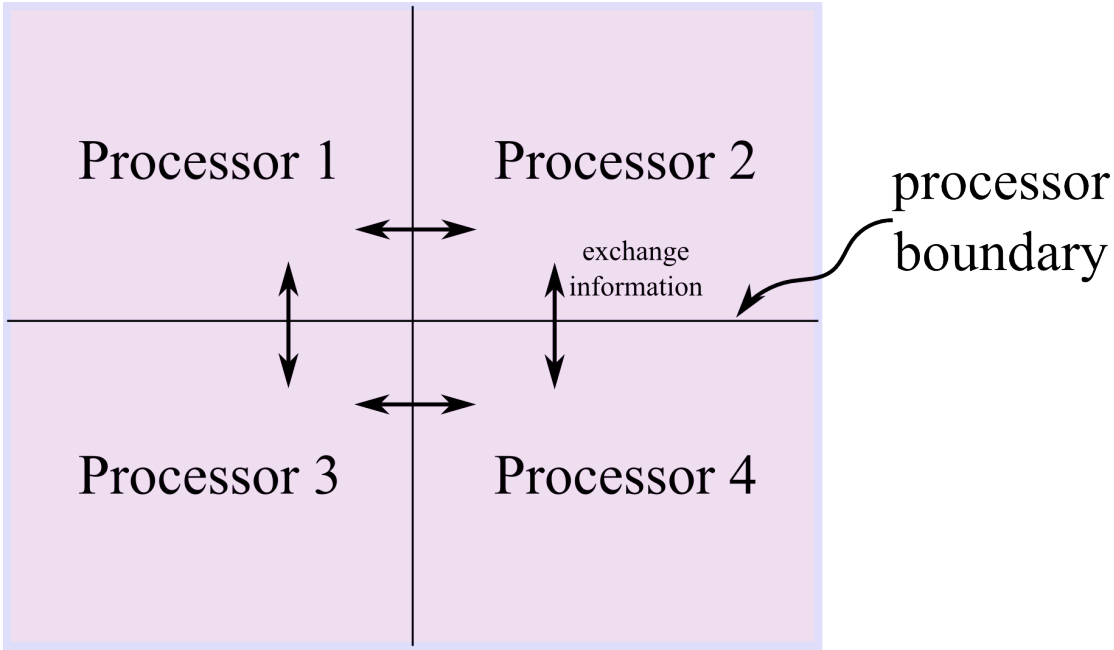
# Parallel computation in OpenFOAM



- From the point of view of FVM discretization, the exchange of information across the processor boundary is through a special boundary condition called `processor`

  `src/finiteVolume/fvMesh/fvPatches/constraint/processor`
  `src/finiteVolume/fvMesh/fvPatches/basic/coupled`
  `src/finiteVolume/fields/fvPatchFields/constraint/processor`

- The `processor` boundaries are resulted from `decomposePar` operation



# Further readings



Further readings on parallel computation:

- https://computing.llnl.gov/tutorials/parallel_comp/

# Questions?