

---

## Towards Parallelism: Part 1

Parallelizing is painful... especially the first time... it takes time, so let's start preparing...

- Start by reading about [Parallel Computing](#) and about [Distributed Computing](#) on Wikipedia.  
Our interest is on *distributed memory computing* via *message passing*, so pay more attention to those...
- It will be done by "domain decomposition", using the MPI library.
- Current Intel terminology on [clusters-nodes-sockets-cores-...](#)
- Parallelization can be done and tested on your own laptop, but don't expect to see speedup...
- After debugging/testing your code on your laptop, you will port it to a real cluster, which takes some work...
- Got us a class account on UT's ISAAC cluster (successor to Newton cluster), see below.  
If you already have an account on ISAAC/Newton cluster, let me know.
- Also take a look at the [Guide to High Performance Computing](#) by Shane Sawyer
- Then take a look at the excellent, extensive tutorial from Cornell:  
[Parallel Programming Concepts and High-Performance Computing](#)

---

### About MPI ( Message Passing Interface )

- Read about [MPI](#) on wikipedia, to get an idea. We'll talk about it in class...
- Check if your Unix/Linux system has some MPI installed.  
Try: **which mpiexec** it should return the path to it. If you get nothing, it may still be there somewhere...  
Try: **which mpirun** , Try: **which mpicc**  
If your Linux system has some MPI installed, fine ... BUT be aware:  
*mpicc and mpirun must come from the same MPI implementation.*  
*It is best to install* your own **openMPI** (or MPICH) and put the path to it in your Makefile.
- In any case, you can easily install your own 'openMPI', under Linux, as follows:
  1. Get the latest stable version ( v4.1.0 ?) from [open-MPI.org](#)
  2. Unpack it: **gtar xzf openmpi-X.Y.Z.tar.gz**
  3. **cd openmpi-X.Y.Z**
  4. read the INSTALL file and follow the instructions.Another option is to install [MPICH](#), which is another implementation of the MPI standard.  
This also exists for MS\_Win, but then you'll need MS\_Win compiler... not sure if this can work out...  
best by far is to install VirtualBox and Linux (ubuntu is probably the most user-friendly).

Note: *Do NOT confuse "openMPI" (an implementation of MPI) with "OpenMP" (shared memory programming standard)!*

MPI has native bindings for Fortran and for C/C++. However, now Python is also an option with:

- [mpi4py](#), you can call Fortran and C code directly through wrappers
- or
- [boost](#) module, has native python bindings built directly on the C++ mpi bindings.
- A good, detailed, reference is this [MPI tutorial](#), from LLNL.

---

### ISAAC: UTK's Advanced Computing Facility (ACF)

Connect to ISAAC portal with your NetID, and request to be added to **ACF-UTK0151** class account:

- [ISAAC portal](#) to ACF UTK's Advanced Computing Facility

## Parallel Computing

refers to using many CPUs to concurrently process bigger / many jobs / users  
"in parallel"

Great variety today (since 2010): cloud

grid

cluster

:

Evolved from vector computers of early 1980s

High clock speeds went up to 3.5 GHz (Xeon, Opteron)  
but too hot, now dialed back to ~2 GHz and use many "cores"

multicore: 2, 4, 8, 16 execution units <sup>(cores)</sup> on one chip <sup>(node)</sup> with shared memory on a bus = SMP node

Have become commodity now, your laptop has i3 (2 cores) or i5 (4 cores) or ...

Does not scale up, went up to 16 cores. GPUs came in, attached to CPUs - shared memory

SMP = symmetric multi-processor

distributed computer = distributed memory multiprocessor

many "nodes" (= board of multicore, local memory, power)

cluster

connected by (Gigabit) ethernet (infiniband) or Cray (fastest)

Scales up to millions

MPP = massively parallel processor = big cluster of nodes connected with high speed (expensive!) network

e.g. IBM BlueGene, ORNL Jaguar, <sup>Cray</sup> UTK Kracken <sup>Cray</sup> (JICS Dart) <sup>dead</sup>

over 200K cores! now dead

Tiger pFloop (dead)

Summit 2018 ~ 500K CPUs + GPUs

FPGA = field programmable gate array, reconfigurable,  
died out

GPU = graphics processing unit (as opposed to CPU = central processing unit (a "core" today))

the latest craze! very fast and very cheap multithreading (~256 threads) shared memory

from gaming industry (NVIDIA), low energy consumption

Attached to a CPU

Since ~2010: energy use is the limiting factor! made GPUs desirable

## Parallel Distributed Computing

with distributed memory, via message passing (MPI)

Serial machines are reaching physical limit of how fast electricity conducts in wire!  
only alternative for further speedup is parallelism.

## Supercomputing (High Performance Computing)

started with vector processing on Cray machines in early 1980s,  
by mid-80s clear that parallelism / distributed computing only way to go...

Modes: SIMD = Single Instruction Multiple Data  
SPMD  
CM-2, MasPar, shared memory: Sequent, N-cube  
SMP (Symmetric MultiProcessing) clusters  
SUN, SGI, HP, DEC,  
good enough for web servers, databases

MIMD = Multiple Instruction Multiple Data

Intel iPSC, i860 hypercube, Paragon  
(serial #1 at ORNL)  
dead 1998

CM-5, KSR (multithreading), IBM SP2 (16 nodes, only \$1M)

Cray T3D, XT3, XT4, ..., Jaguar

Networked workstations thanks to PVM (March 1991)

Architectures: shared memory: easier to program (OpenMP), does not scale

distributed memory: harder to program, extensible to thousands of CPUs

clusters of "nodes" (each with 4, 8, 16 <sup>cores</sup> CPUs on one board share memory)  
connected with fast local network

Parallelism: • functional (task)

• data via domain decomposition: most scalable

• embarrassingly parallel (independent tasks, like Monte Carlo)



layers: hardware: cluster of nodes, each with several cores (4 or 8 or 16 or 32)  
connected by network/switch: Gigabit ethernet, Infiniband, ...

software: scheduler: PBS torque/moab

compiler: mpiC6, mpiF77, mpiF90 : compiler wrappers with MPI lib  
mpicxx

code calls MPI library routines, bindings for C, Fortran, Python  
C++ mpiCxx mpiF77 mpiF90 mpiCxx

### Steps to parallelize:

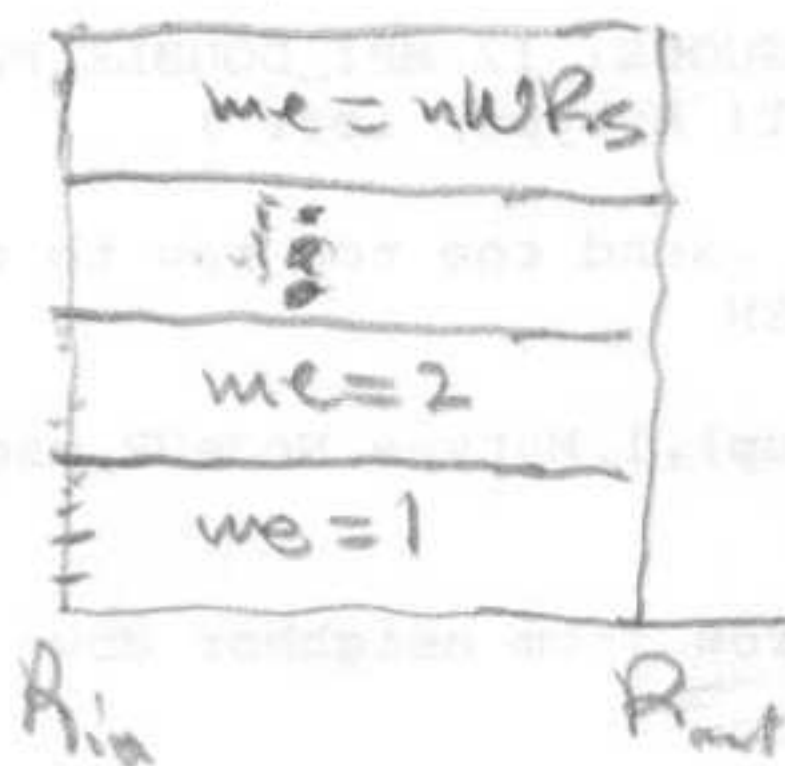
1. debug serial code, clean it up
2. parallelize serial code to parallel version (can use SVN or git for version control)
3. compile using a Makefile
4. run on a cluster (ACF) with PBS via PBSscript

on your machine with openMPI: mpxexec -n NPROC ./code.x

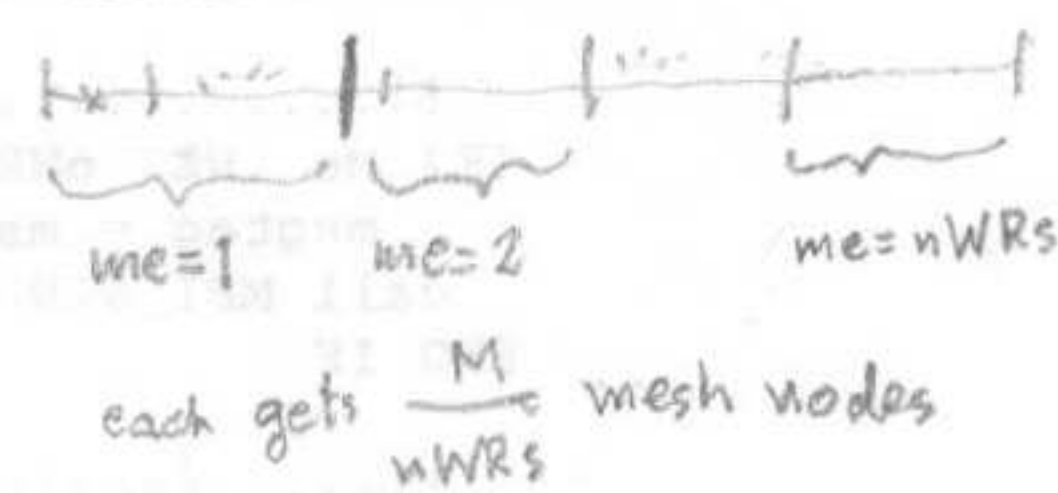
1D strategy via domain decomposition (in one direction only)  
implemented in Master-Workers style

domain decomposition in 2-direction

1. decompose  $[0:M]$  into  $nWRs$  pieces  
and assign each portion to one task process ~~node~~  
 $me=1, 2, \dots, nWRs$



for 1-D



so should choose  $M$  even  
and a multiple of  $nWRs$

e.g. to run on 8 workers:  $M=8, 16, \dots$

2. To be run on  $nPROC = nWRs + 1$  MPI processes  
with Master ( $me=0$ ) handling: I/O, distributing tasks to Workers

All workers solve the "same" problem (execute the same routines)  
but on their portion of the mesh.

At each time step, they need to exchange their "boundary" values  $U_{i,0}$ ,  $U_{i,Mz+1}$   
with their neighbors via msg passing

At each tout, they all send their values to Master  
who assembles them on the global mesh and prints out

- a. Communication is much slower than computation! should be minimized
- b. debugging is very hard! do it right the first time!
- c. Objective is to reduce wall-clock time (speed up how long it takes to do job)

$$\text{Speedup} = \frac{T_{\text{Serial}}}{T_{\text{Parallel}}}$$

$$\text{Efficiency} = \frac{T_{\text{Serial}}}{p T_p}$$

use  $T_{\text{Serial}} = (\text{CPU time on 2}) \times 2$

ideal is  $p$  on  $p$  processors  
called linear speedup

ideal = 1

Rarely achieved due to communication overhead

but "superlinear speed" can occur! mostly due to cache/memory gains

A big problem may fill memory and use virtual memory (disc!, very slow)  
but by splitting problem to many CPUs may make it fit in memory.

MPI processes may be assigned all to one processor (CPU) or to a few CPUs or one to one.

Can install MPI-enabled compiler and MPI library on your own laptop, even if it has single CPU!

MPI processes will run on one (or few) CPUs, so no speedup, but can debug!