Scientific Computing: Lecture 24

- General Introduction to Parallel Processing
- Model for parallelization (hardware)
- Memory architectures
- Programming models
- GPUs – Graphical Processor Units

CLASS NOTES

- HW09 due Monday.
- Reading in handout.
- WORK ON PROJECTS!

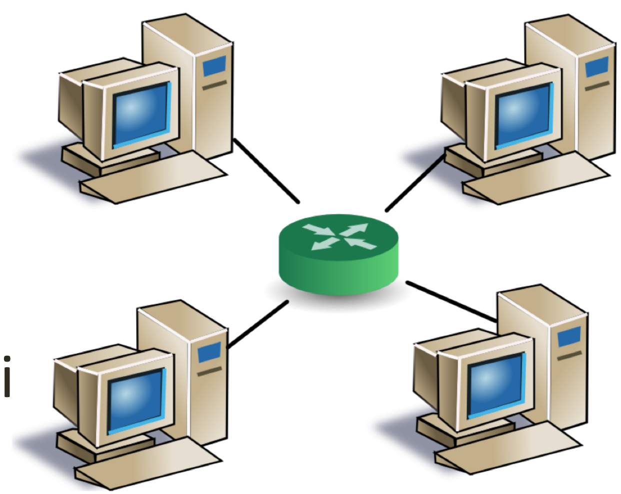
University of Mississippi
Dept. of Physics and Astronomy
Phys 730, Dr. Gladden

Introduction to parallel processing

- Parallel computing is a very broad term describing schemes by which to break up large problems into multiple smaller problems.
- Some problems are easy to cast in a parallel form:
  - Need to fit experimental data to a model at 100 different temperature points.
  - Have 5 different machines work on 20 different data sets (temperatures) at the same time.
  - <u>Important characteristic:</u> each job is independent of the results of the previous jobs.

Introduction to parallel processing

- Other problems are more difficult to parallelize
  - Molecular dynamics:
    - each time step in the simulation depends on the state at the previous time step.
    - Break up by space – have different CPUs work on the same time step, but different sets of atoms.
    - The ‘boundary’ atoms are tricky!
  - “I know how to make 4 horses pull a cart. I don’t know how to make 1024 chickens do it!”
    ~~Enrico Clementi

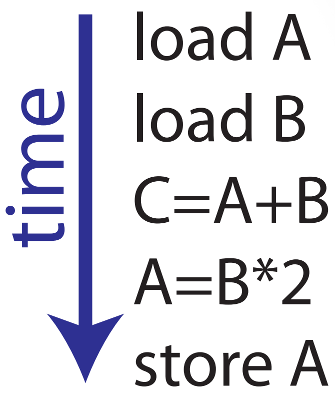


Introduction to parallel processing

- Traditional:
  - Serial computing instructions and data are streamed to CPU in sequence.
- Parallel:
  - Problem is compartmentalized.
  - A series of instructions are generated for each part and sent to multiple CPUs.
  - Results are recombined for the overall solution.
- Seriously parallel problems
  - Climate models, molecular dynamics, signal processing, fluid dynamics.
  - Written in compiled languages (C, C++, Fortran,…)

Models – Flynn's Taxonomy

- Single Instruction, Single Data
  - Traditional serial computer
  - 1 source of data (memory),
    1 instruction executed at a time.



- Multiple Instruction, Single Data
  - 1 source of data to multiple CPUs, but each CPU performs different instructions on the same data.
  - This is very rare and only a few such machines have been built to solve very specific problems.

Models – Flynn's Taxonomy

- Single Instruction, Multiple Data
  - Multiple processing units (CPUs), each execute the SAME instruction at the SAME time, but on different data.
  - Pretty specialized. Vector machines like Cray C90 and NEC SX-2.

CPU#1	CPU#2	CPU#3
load A(1)	load A(2)	load A(3)
load B(1)	load B(2)	load B(3)
C(1)=A(1)+B(1)	C(2)=A(2)+B(2)	C(3)=A(3)+B(3)
store C(1)	store C(2)	store C(3)

time

Models – Flynn’s Taxonomy

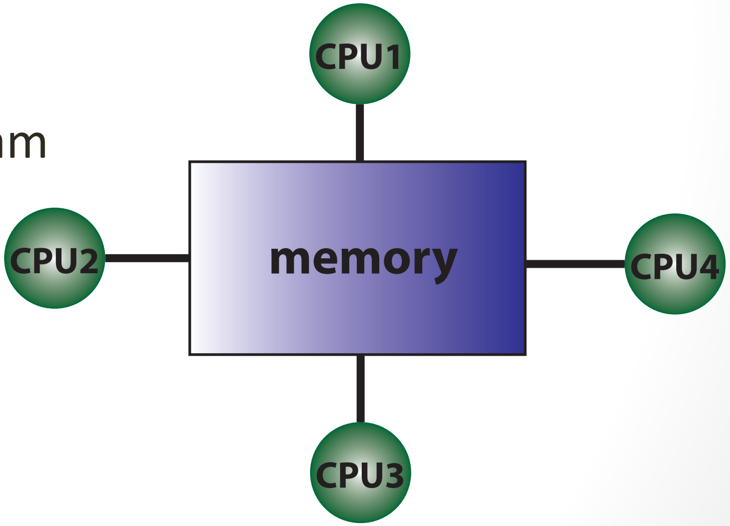
- Multiple Instruction, Multiple Data
  - Each CPU executes different instructions on different data streams.
  - Provides the highest flexibility and easiest to implement.
  - Most common model. Examples: Multicore CPUs, clusters, grids.

time

CPU#1	CPU#2	CPU#3
load A(1)	call funct	i=0
load B(1)	x=funct(y)	i +=1
C(1)=A(1)+B(1)	sum=x**2	…
store C(1)	store sum	…

Memory Architectures – Shared Memory

- Shared Memory
  - All CPUs see the same memory space all the time.
  - When CPU#1 changes an element in an array, all CPUs immediately have access to the new value
  - Advantages:
    - Global addresses, easier to program
    - Data sharing is fast
  - Disadvantages
    - Lack of scalability – more CPUs means more I/O traffic.
    - Programmer must be careful that the order of instructions on each CPU is correctly timed.

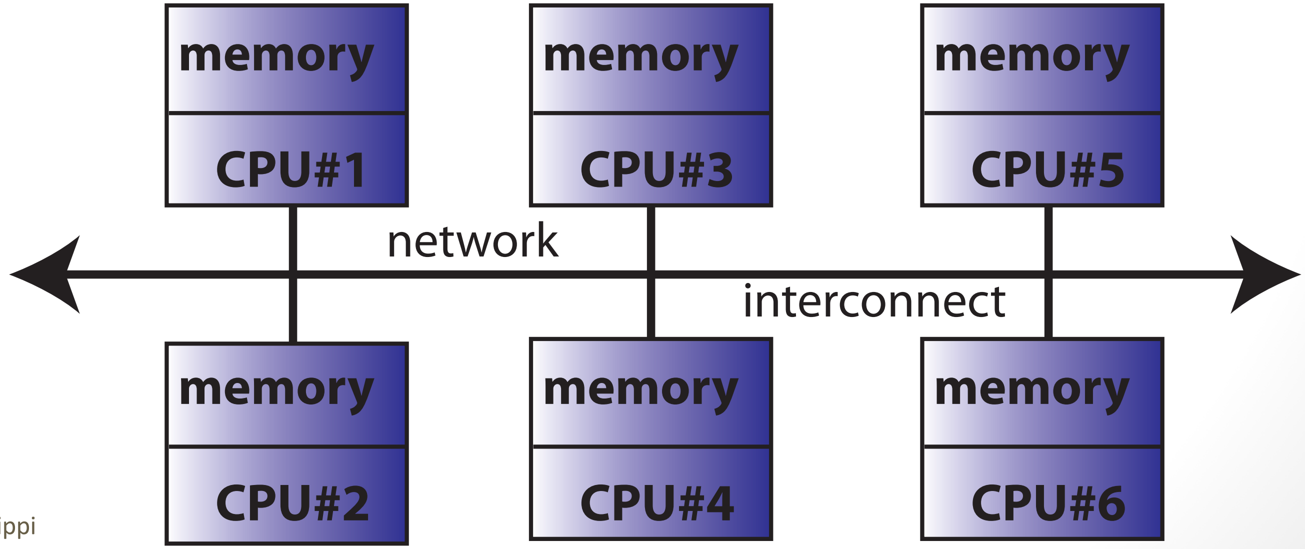


Distributed Memory

- Each CPU has it's own memory
- Communication is required to move data from one block to another
- Advantages:
  - Scalable: only 1 CPU per memory block
  - Easy and cheap to build – just a pile of PCs will do.
- Disadvantages
  - Programmer is responsible for lots of details for flow and access of data.
  - Traditional data structures may not be easily mapped from traditional global memory model.

Distributed Memory

- Each CPU has it's own memory
- Communication is required to move data from one block to another.
- 'Interconnects' become the bottleneck
  - Gigabit ethernet, fiber optic, infinni-band.



HPC Cluster Examples inMS

MS Center for Supercomputing Research (UM, Oxford)

Sequia: 1304 cores,

Catalpa: 320 cores, 2.5 TB

Maple: 1228 cores, 29 GPUs, 3.3 TB

DoD Supercomputing Research Center (ERDC, Vicksburg)

Cray XE6: 150,912 cores, 1509 TFLOPS

SGI Altix: 7,680 cores, 172 TFLOPS

Cray XE6: 14,976 cores, 138 TFLOPS

Programming Models

- There are MANY ways to break larger problems into smaller ones and methods to implement them. We'll discuss 2 most common.
- Threads
  - Subroutines are branched off to processors while program continues to execute.
  - Threading has been supported for years.
  - Specifics depend on OS
    - POSIX threads: UNIX flavors and Mac OS X
    - Open MP: UNIX and Windows NT
    - Microsoft proprietary implementation
  - Python has several threading modules.

Programming models

- Message Passing Interface (MPI)
  - Most common model on large machines
  - Tasks share data by sending and receiving messages.
  - Require cooperation: a ‘send’ message must coordinate with a ‘receive’ operation.
  - MPI is pretty much industry standard.
  - Several proprietary libraries as well as open source (openMPI) available for all OS’s.

Design of parallel programs

- Automatic
  - Take existing serial code and let a special compiler break loops into tasks for different CPUs.
  - Usually not very efficient – does not achieve optimal speed up. In fact, performance can actually get worse!
- Programmer Directed
  - Manually edit code using MPI commands
  - More fine tuning and optimization IF you know what you are doing.
  - Can be difficult and time consuming.

MPI commands with pympi

- pyMPI module requires MPI (like openMPI) libraries to be installed and configured (independent of python)
- Commands after 'import mpi'
  - mpi.size() – number of processors
  - mpi.rank() – specific processor. mpi.rank=0 is called the 'root' processor that acts like a traffic cop directing the other CPUs.
  - Broadcast – broadcast data to all processors.
    Code on root: mpi.bcast(some_array)
    Code on rest: some_array = mpi.bcast()

MPI commands with pympi

- Commands after ‘import mpi’
  - Reductions: Inverse of broadcast – root requests data from all other tasks.
    - Example:
      `totalArea = mpi.reduce(localArea, mpi.SUM)`
      where localArea are areas computed by each task and mpi.SUM adds all the localAreas as they come in to finally result in the totalArea.
  - Point to point communication (to a specific task) with mpi.send(message, task#) and msg,status=mpi.recv(task#)

MPI commands with pympi

- Commands after ‘import mpi’
- Scatter/gather methods
  - Break sequence into even parts and send each part to a different task for processing.
  - After processing, partial results are gathered and reassembled by root.
  - Example:
```
seq=[1,2,3,4,5,6]
local_seq = mpi.scatter(seq)
```
    if mpi.size=3, then local_seq = [1,2] on task 0, [3,4] in task 1, and [5,6] in task 3.
```
new_seq = mpi.gather(local_seq)
```

Parallel Python

- While MPI is an industry standard for very large machines, pyMPI is a bit awkward to use – MPI libraries (not python) need to be loaded and configured on all machines, syntax is not very intuitive.
- Parallel Python is a more intuitive and flexible way to taking advantage of many CPUs.
  - Can be used on a multicore processor (SMP) or a large cluster – even widely distributed processors.
  - Syntax is more “pythonic” and intuitive.
  - Written 100% in python – easy to get “under the hood” to see what is happening.
  - Does NOT come with Enthought, but can be loaded as an add-on.

Parallel Python

- Model and Syntax
  - Each node (server) must be running a small program called ‘ppserver.py’ which listens for requests.
  - The controller (your program) contacts each listed server and requests a computation through a socket.
  - Each server returns it’s result and controller stitches the results back together.
  - Servers indicated by:

```
ppservers =
(‘myhost.olemiss.edu’,’myhost2.olemiss.edu’)
```

Parallel Python – Starting Jobs

- Establish connections to server pool:
```
job_server = pp.Server(numcpus,\
ppservers = ppservers)
```
- Start a job by sending a function to evaluate, usually in a loop):
```
jobs[i] = job_server.submit(myfunct,\
args=(functargs),\
depfuncs = (funct1,funct2,..))
```
- Compile results:
```
result = sum( [ jobs[i]() \
for i in range(len(jobs)) ] )
```

Parallel Python - Gotchas

- Parallel python is based on the subprocess module which starts new forks for each request. Will happily add 1000 forks even if run on a machine with only 4 processors.
  - Need to check how many processors are actually free.
  - 'mpstat –P ALL' is useful for this on linux systems.
- If too many <u>remote</u> processes are requested, the subprocess module can fail with a 'too many open files' error. I found this systems fails for Nproc > 12.
- You take a BIG hit on speed if your processes are remote (over ethernet or internet).
- Conversely, very efficient if all processes are local.
  - See code run remotely vs. locally.

Graphical Processing Units

- Relatively new paradigm in parallel processing.
- They are in the class of a vector processor.
- GPUs have long been around and used to process, control, and update displays. They have inherently operated in a highly fashion
  - Controlling thousands to millions of completely independent pixels on the screen.
  - So thousands of cores on a single chip!
- HOWEVER, these cores are NOT CPUs and have limited operation sets!

Applications of GPUs

- Types of applications of GPUs
  - Problems that exhibit a high degree of "data-parallelism"
  - Single Instruction, Multiple Data
- These limitations mean GPUs can only be used for a subset of problems.
  - Ray tracing
  - Some large matrix operations
  - Signal processing
- HPC applications are now including support for GPUs
  - LAMPS, NAMD, Caffe (artificial vision), MATLAB

GPU Hardware and Programming

- Commercial GPU systems
  - TESLA K80 by NVIDIA: 4992 cores
  - FirePro by ATI: 2816 cores
- Programming environments
  - CUDA
  - OpenCL – Apple, Inc.

GPUs and Python

- GPU programming is still pretty low level.
- Python implementations and tools for GPU programming as still quite immature – but rapidly evolving!
- All still require writing some code directly in c++ as a string which gets passed through Python to the underlying libraries.
  - pyCUDA
  - pyOpenCL