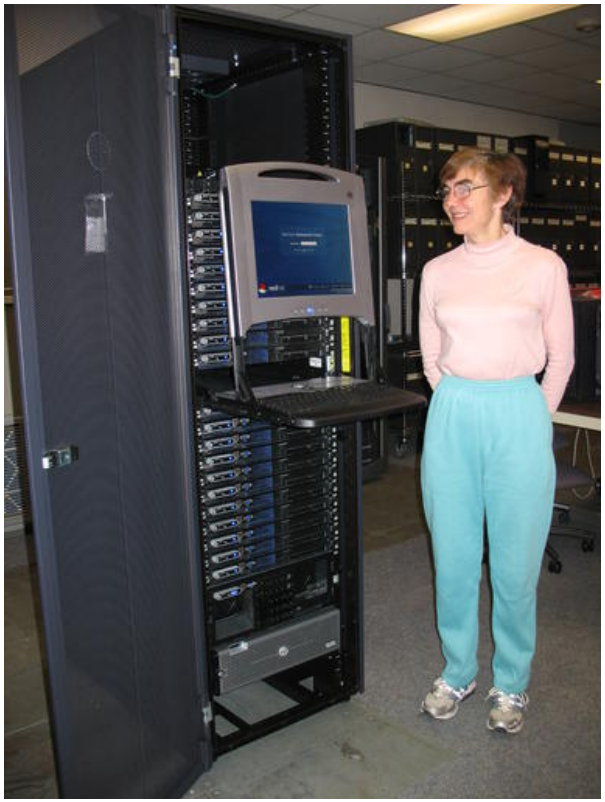
The Statistics Beowulf Cluster

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

September 6, 2007

The Statistics Beowulf Cluster



Basic Configuration

- Master/head node `beowulf.stat.uiowa.edu`, also `node00`
  - 4 dual core Opteron 8216 (2.4GHz)
  - 16 GB RAM
  - 1 TB disk, NFS-mounted on client nodes
- 21 Client nodes (`node01` – `node21`)
  - 2 dual core Opteron 2216 (2.4GHz)
  - 8 GB RAM
  - 150 GB local disk
- `node00` has two network connections
  - one to the campus network
  - one to the internal Gbit network
- The individual nodes are connected to the internal network

Some Performance Results

- The High-Performance Linpack (HPL) benchmark is commonly used for measuring performance of parallel computers.
- The benchmark solves a random dense linear system in parallel.
- Benchmark code is available from

  `http://www.netlib.org/benchmark/hpl/`

- Some results (Gflops) running the benchmark on `beowulf`:

			N			
Procs.	30000	20000	16384	8192	4096	1024
1						2.7
4				10.0	8.9	3.4
16				28.0	18.0	3.2
32				33.0	18.0	2.1
64	76.0	54.0	45.0	25.0	13.0	1.7

- Theoretical maximal performance: $2.7 \times 92 = 248.4$ Gflops.

Accessing The Nodes

- Using the nodes involves, at some level, using `ssh`.
- Probably the easiest way is to do this once:
  - Make sure the machine you will be logging in from
    - has your private ssh key installed
    - is running an ssh agent (standard Linux consoles and nomachine do this)
  - Add your public key to the `.ssh/authorized_keys2` file on `beowulf`.
- Then each time you want to use `beowulf`:
  - On the machine running the ssh agent add your ssh identity with `ssh-add`.
  - Log into beowulf with

    ```
    slogin -AX beowulf.stat.uiowa.edu
    ```

    You can avoid the `-AX` by setting up a `.ssh/config` file.
  - From `beowulf/node00` you can log into `node01` with

    ```
    slogin -X node01
    ```

Poking Around

- The current cluster load is available (from within `uiowa.edu`) at

  `http://beowulf.stat.uiowa.edu/ganglia/`
- A list of current active nodes is in file `/cluster/scripts/active`
- The script `/cluster/scripts/dcmd` may be useful:

  `/cluster/scripts/dcmd uptime`

  gives status information on `node01`–`node21`.
- The `ps`, `kill`, and `killall` may become familiar.
- Finding all my R processes:

  `/cluster/scripts/admin/dcmd 'ps -u luke | grep R'`
- The `top` command can also be useful
  - hitting `1` toggles display of individual CPU loads.
  - hitting `q` exits.

Some Available Software

- Compilers:
  - `gcc` and friends
    - Default is currently the 3.x series with `gcc`, `g77`.
    - The 4.x series is also available as `gcc4` and `gfortran`.
    - The 4.x series supports OpenMP.
    - Mixing the two may not work (especially for FORTRAN code).
  - Intel compilers `icc`, `ifc`
- Linear algebra libraries:
  - BLAS, LAPACK
  - ACML accellerated BLAS (and some LAPACK)
  - PaLAPACK, ScaLAPACK
  - IMSL (VNI)
- Message-passing libraries:
  - PVM
  - LAM MPI
  - MPICH
- R, with some parallel computing packages.

Outline of Parallel Computing

- Parallel computing involves splitting work among several processors.
- Processor memory can be shared or distributed.
- Shared memory parallel computing typically has
  - single process
  - single address space
  - multiple *threads* or *light-weight processes*
  - all data is shared
  - access to key data needs to be synchronized
- Distributed memory computing usually has
  - multiple processes, possibly on different computers
  - each process has its own address space
  - data needs to be exchanged explicitly
  - data exchange points are usually points of synchronization
- Intermediate models are possible.

Issues and Pitfalls

- Some issues are common to all variants:
  - Deadlock
  - Uneven sub-problem size and load balancing
  - Overhead of synchronization
- Other issues that vary or affect one more than the other:
  - unintended sharing or insufficient synchronization with share memory
  - node or communication failure with distributed memory
  - communication overhead for distributed memory
  - synchronizing access to standard input/output and files

Some Common Patterns

- Scatter-compute-gather, master-worker:
  - master program/thread divides up the work
  - several worker programs/threads do the work
  - master continues once all workers are done
- Pipeline, producer-consumer:
  - Sequence of process stages $P_1, \ldots, P_N$
  - Output of P_i is input to P_{i+1}
  - Once the pipeline is warmed up all stages can run in parallel

  A generalization is a *systolic array*.
- More complex forms:
  - sequences of scatter-compute-gather steps
  - array bases communication topologies
  - ...

Outline of Distributed Memory Parallel Computing

- Some basic requirements:
  - Need to start a collection of programs on multiple computers.
  - Programs need a way to exchange data.
  - Need so shut things down cleanly
- Supporting frameworks
  - Message-passing library
    - PVM
    - MPI (some flavor)
  - Batch scheduler
    - Condor
- Utilities
  - `xpvm` or `xmpi`
  - load monitoring web site
  - `dcmd` script, `ps`, and friends

A Simple MPI Example

- First example program from Pacheco's MPI book

  `http://www.stat.uiowa.edu/~luke/classes/295-hpc/examples/mpigreetings.c`

- Minimal MPI program using
  - `MPI_Init`
  - `MPI_Comm_rank`, `MPI_Comm_size`
  - `MPI_Send`, `MPI_Recv`
  - `MPI_Finalize`
- To run using LAM-MPI
  - compile program with `mpicc`
  - start LAM with `lamboot`
  - run program with `mpirun`
  - shut lam down with `lamhalt`
- It's not a bad idea to check for stray `lamd` processes.

A Simple PVM Example

- A variant of the `forkjoin.c` example in the PVM book:

  `http://www.stat.uiowa.edu/~luke/classes/295-hpc/examples/pvmgreetings.c`

- Program does its own process spawning with `pvm_spawn`
- Send/receive involves a few more calls
- To run
  - compile program with appropriate flags
  - start pvm with `pvm` (the PVM console)
  - run program as `./pvmgreetings 5`
  - shut pvm down with `halt` in the PVM console
- It's not a bad idea to check for stray `pvmd` processes.

Some Comparisons and Comments

- PVM is a single implementation, MPI is a standard.
- MPI implementations can be tuned to particular kinds of hardware.
- There are several versions of the MPI standard.
- MPI 2.0 (and LAM) allow process spawning; older MPI versions do not.
- Some system administrators disable process spawning for MPI 2.0.
- MPI provides simpler basic communication.
- MPI has more built-in support for complex communication patterns.
- PVM has some support for fault-tolerance; so far MPI does not.
- PVM supports multiple architectures.
- `xpvm` is superior to MPI analogs I have found (e.g. `xmpi`).
- Both provide a rich set of tools.
- Both need to be used with care to avoid deadlock.