2nd Slide Set Cloud Computing

Prof. Dr. Christian Baun

Frankfurt University of Applied Sciences
(1971–2014: Fachhochschule Frankfurt am Main)
Faculty of Computer Science and Engineering
christianbaun@fb2.fra-uas.de

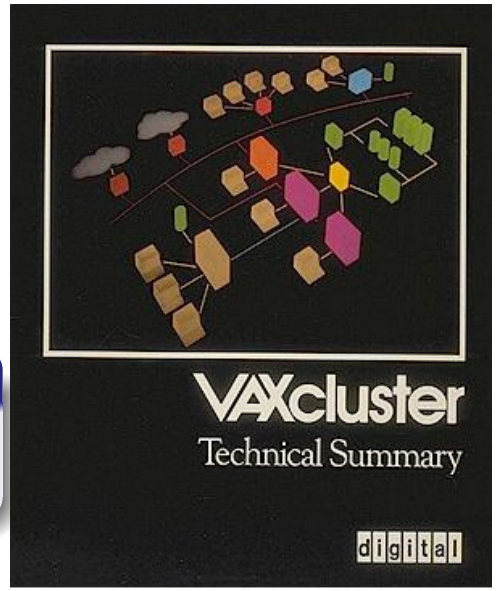
Agenda for Today

- Cluster computing
  - History of cluster computing
  - Distinguishing criteria
    - Structure (homogeneous, heterogeneous)
    - Installation concepts (Glass-house, Campus-wide)
    - Fields of application
    - High Availability Clustering
    - High Performance Clustering
    - High Throughput Clustering
    - Behaviour in the event of failed nodes (Active/Passive, Active/Active)
  - Current situation
  - Advantages and drawbacks of clusters
  - Cluster application libraries (PVM, MPI)
  - Gearman

History of Cluster computing

- 1983: Digital Equipment Corporation (DEC) offers for its VAX-11 system a cluster solution (VAXcluster)
  - VAXcluster allows to connect multiple computers via a serial link
  - By combining multiple VAX systems, their computing power and memory can be accessed equal to a single computer system

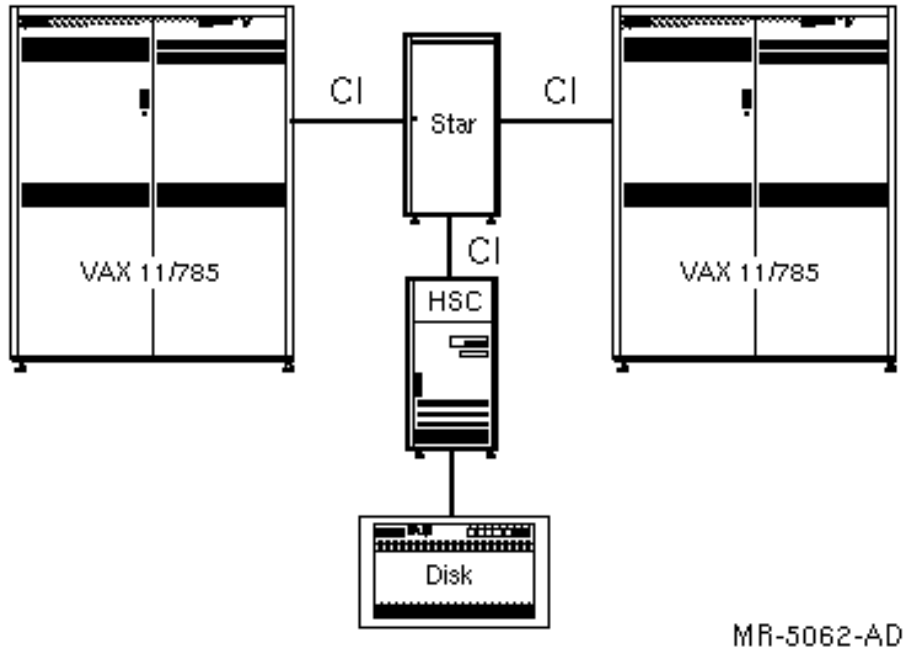
- 1987 DEC sells VAX 8974 and VAX 8978
  - These are clusters, which contain 4 or 8 nodes (VAX 8700 systems) and a MicroVAX II, which is used as console



Further information

VAXcluster system. Digital Technical Journal. Number 5. September 1987
`http://www.dtjcd.vmsresource.org.uk/pdfs/dtj_v01-05_sep1987.pdf`

VAXcluster



CI = ComputerInterconnect
HSC = Hierarchical Storage Controller
Star = Star Coupler

Image sources:

http://hampage.hu/oldiron/vaxen/eikvms1.jpg
http://odl.sysworks.biz/disk$vaxdocmay941/decw$book/d3ywaa51.p37.decw$book
http://www.computerhistory.org/collections/catalog/102635385

History of Cluster computing

- Early 1990s:
  - Client/Server architecture became mainstream
  - UNIX workstations were used in companies and universities
  - Workstations are only utilized 5-10% during their everyday work
    - The biggest part of the potential computing power is wasted
  - Wanted: A middleware, which makes the free computing power available for solving complex tasks
  - PVM and MPI both, implement a **Message Passing Layer**, which is independent of the nodes' architecture and allows to develop cluster applications
- Mid-1990s:
  - Growing popularity of Clusters with commodity hardware nodes
  - Compared to mainframes, Clusters provide for a low price, depending on their hardware and structure, high computing performance and/or availability
  - Flexibility in terms of the application and extensibility

Definition of Cluster Computing

Cluster computing

Clustering is parallel computing on systems with distributed memory

- A cluster consists of at least 2 nodes
  - Each node is an independent computer system
  - The nodes are connected via a computer network
    - In clusters with just a few nodes, inexpensive computer network technologies (Fast or Giga-Ethernet) are used
    - Clusters with several hundred nodes require high-speed computer networks (e.g. InfiniBand)
  - Often, the nodes are under the control of a master and are attached to a shared storage
  - Nodes can be ordinary PCs, containing commodity hardware, workstations, servers or supercomputers

From the user perspective (in a perfect world)...

- the cluster works like a single system $\Longrightarrow$ a virtual uniprocessor system
- Ideally, the users don't know, that they work with a cluster system

Cluster of Workstations / „Feierabendcluster“

- If the nodes are only available at specific times, the cluster is called Clusters of workstations (COWs) or Network of Workstations (NOWs)
- During normal working times, the employees use the nodes of such a cluster system as workstations
  - The concept was popular around the year 2000
  - Today, this concept not popular any more

Oliver Diedrich, *NT-Workstations als Feierabend-Cluster*, c't 22/2000, P.246

- More than 200 computers with Pentium II/III CPUs and with at least 64 MB RAM in the plastics laboratory of BASF in Ludwigshafen
- All computers are centrally administered and run Windows NT 4.0
- A WinSock server runs on every computer as a service all the time
- If the WinSock server receives a request from the central host, it confirms the request
- Next, the central host transmits a file (size: 10-100 KB) via TCP/IP to the server, which includes data for processing
- If the transmission was successful, the WinSock server processes the data and transmits the results back to the central host

Distinguishing Criteria of Clusters

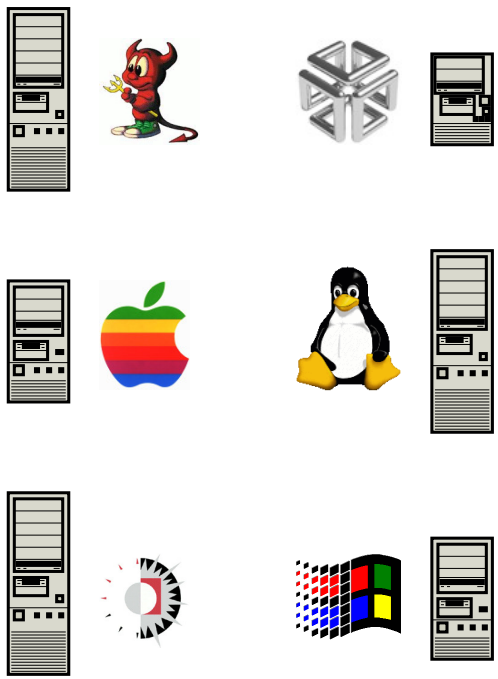
- **Structure**
  - Homogeneous structure
  - Heterogeneous structure
- **Installation concept**
  - Glass-house
  - Campus-wide
- **Fields of application**
  - High Performance Clustering
  - High Availability Clustering
  - High Throughput Clustering
- **Behavior in the event of failed nodes**
  - Active/Passive-Cluster
  - Active/Active-Cluster

Homogeneous and Heterogeneous Clusters

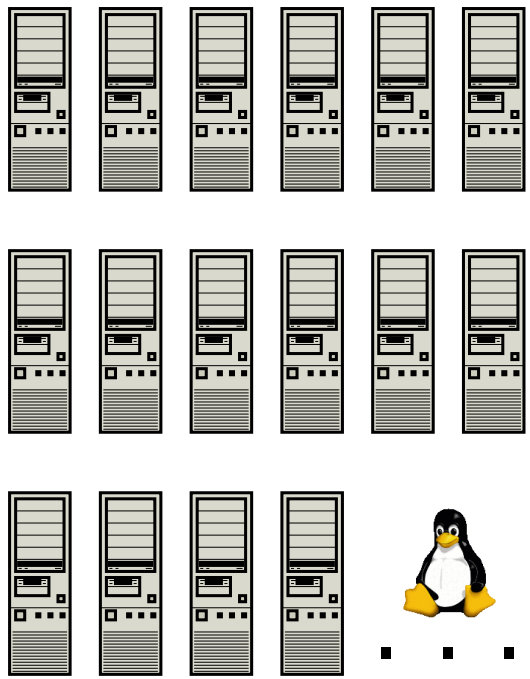
- The structure of clusters can be homogeneous and heterogeneous

I have never seen a heterogeneous cluster with different operating systems in practice...

Heterogeneous structure



Homogeneous structure

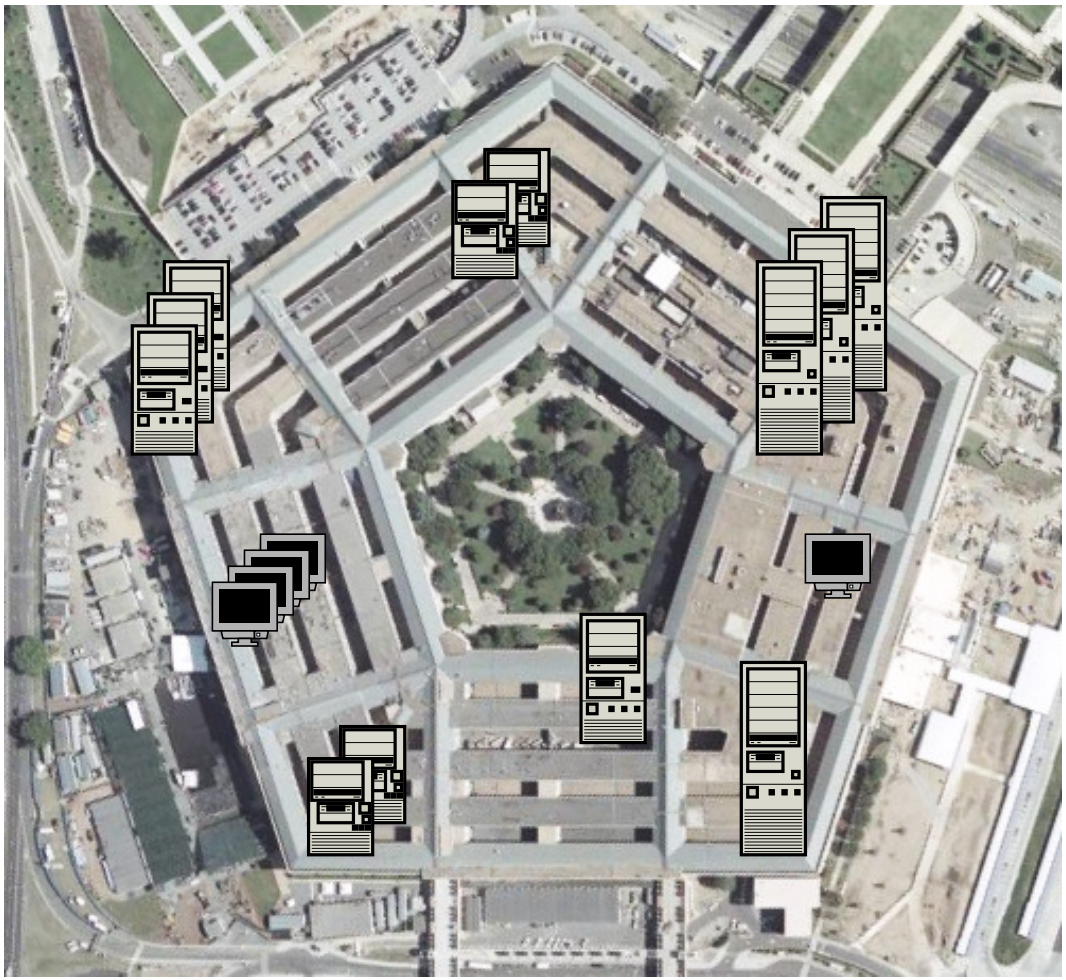


- In practice, the construction of a heterogeneous cluster is generally a bad idea
- The administration of homogeneous clusters is challenging, but the administration of heterogeneous clusters is hell (especially when commodity hardware is used)

Installation Concepts of Clusters (1/2)

- **Glass-house**
  - The cluster is located in a single room or server rack
- Advantages:
  - Fast access to all components for maintenance and troubleshooting
  - Nodes can be connected via high-performance networks
  - Increased protection against sabotage
- Drawbacks:
  - In case of a power failure or fire in the building, the operation of the entire cluster is at risk

Installation Concepts of Clusters (2/2)



- **Campus-wide**
  - The nodes are located in multiple buildings and spread across the site of the research center or company
- Advantages:
  - It is hard to destroy the cluster completely
- Drawbacks:
  - It is impossible to use high-performance computer networks
  - Often, the nodes contain different hardware components

Fields of Application of Clusters

- Clusters for different applications exist

1. **High Availability Clustering**
   - Objective: high availability
2. **High Performance Clustering**
   - Objective: high computing power
3. **High Throughput Clustering**
   - Objective: high throughput

High Availability Clustering (1/2)

- For High Availability Clustering, highly available systems are required
  - This is achieved by:
    - Redundancy of nodes and their components
    - Avoiding a single point of failure

Redundancy

Redundancy comes from the Latin word *redundare* and means that more than enough of something is available

- Redundancy means: A system contains components, which are not required for the functioning of the system, but they can take over the work of identical components in case of error
- By using redundant nodes, it is possible to emulate the technology and benefits of mainframes for a low price and a high level of flexibility is achieved
- **The aim of HA clustering is to archive a high availability**

Availability

- The availability of a system is calculated from the mean uptime (**Mean Time Between Failures**), divided by the sum of the mean uptime and the mean downtime (**Mean Time To Repair**)

$$\text{availability} = \frac{\text{mean uptime}}{\text{mean uptime} + \text{mean downtime}}$$

- To achieve high availability of the cluster, it is not sufficient to use redundant nodes and redundant hardware inside the nodes
  - Uninterruptible power supplys (UPS) are required
  - Protection against improper use, and sabotage
- For systems, which are declared as fail-safe, a replacement computer must always be available and running, which takes over the services and provides them in case of error

High Availability Clustering (2/2)

- For High Availability Clustering, not the availability of the nodes has top priority, but the availability of the offered services
- 2 groups of High Availability Clusters exist, which differ in their behaviour in the event of failures of nodes:
  - Active/Passive-Cluster
  - Active/Active-Cluster

HA-Clustering – Active/Passive and Active/Active

- **Active/Passive-Cluster** (also called: **Hot-Standby-Clusters**)
  - During normal operation, at least a single node is in **passive** state
  - Nodes in passive state do not provide services during normal operation
  - If a node fails, a passive node takes over its services
  - **Failover** = a node takes over the services of a failed node
  - Benefit: The services must not be designed for cluster operation
  - Drawback: Much potentially available performance remains unused in normal operation
- **Active/Active-Cluster**
  - All nodes run the same services
  - All nodes are in **active** state
  - If nodes fail, the remaining active nodes need to take over their tasks
  - Advantage: Better distribution of load between nodes
  - Drawback: Services need to be designed for cluster operation, because all nodes access shared resources (data!) simultaneously

High Availability Clustering – Failover and Failback

- **Failover**: Ability to automatically transfer the tasks of a failed node to another node for minimizing the downtime
  - The failover functionality is usually provided by the operating system
  - Example: Heartbeat for Linux

http://www.linux-ha.org/wiki/Heartbeat

Heartbeat is a daemon that provides cluster infrastructure (communication and membership) services to its clients. This allows clients to know about the presence (or disappearance!) of peer processes on other machines and to easily exchange messages with them. In order to be useful to users, the Heartbeat daemon needs to be combined with a cluster resource manager (CRM) which has the task of starting and stopping the services (IP addresses, web servers, etc.) that cluster will make highly available. Pacemaker is the preferred cluster resource manager for clusters based on Heartbeat.

- **Failback**: If failed nodes are operational again, they report their status to the load balancer and get new jobs assigned in the future
  - From that point in time, the cluster again has the same performance capability, it had before the failure of the nodes

Well written articles about Heartbeat and DRBD

Andreas Sebald. *Linux-HA-Cluster mit Heartbeat und DRBD*. Linux-Magazin 07/2004.
http://www.linux-magazin.de/Ausgaben/2004/07/Reservespieler

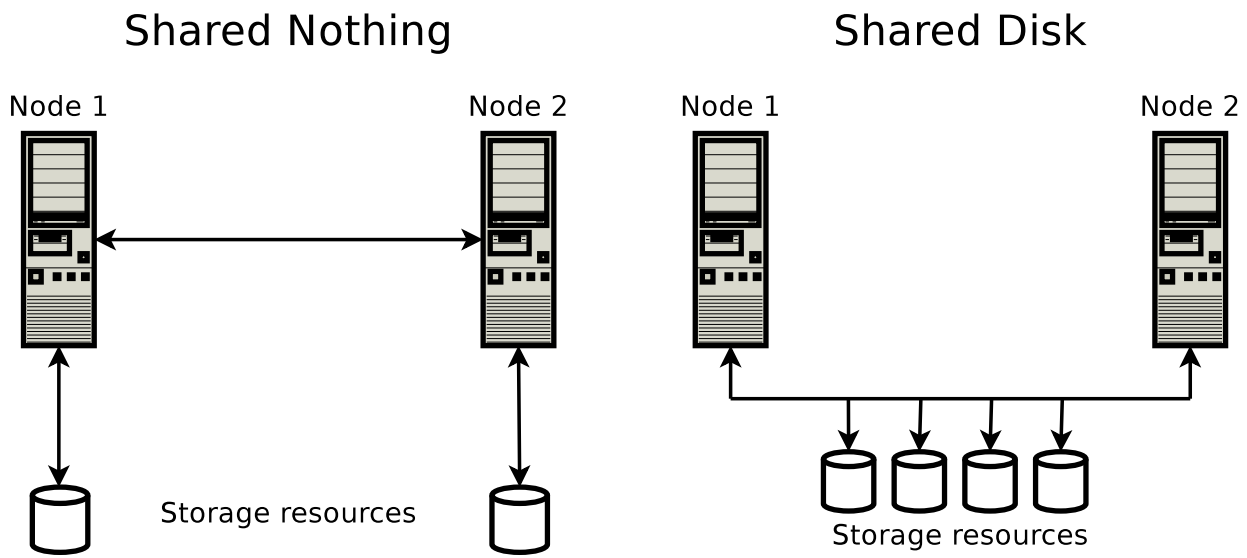
Split Brain

- Connection failure between nodes
  - The nodes still work without any trouble, only the connection between them is disrupted
  - Tools like Heartbeat, which monitor the presence (or disappearance) of nodes, assume that nodes are broken
  - Each node declares itself to be the primary node
    - In Active/Passive-Clusters $\Longrightarrow$ failure of the cluster (offered services)
- **If shared storage is used**, each node tries to write on the storage
  - This can be avoided by using additional hardware and distinguishing the MAC addresses
    - One possible solution to avoid further issues: If simultaneous access attempts are detected from different MAC addresses, are nodes are automatically shut down
- **If distributed storage is used**, write requests cause inconsistent data on the nodes
  - It is difficult to fix the broken consistency without loosing data

More information about the different storage architectures of High Availability Clusters present the next slides

Architectures of High Availability Clustering

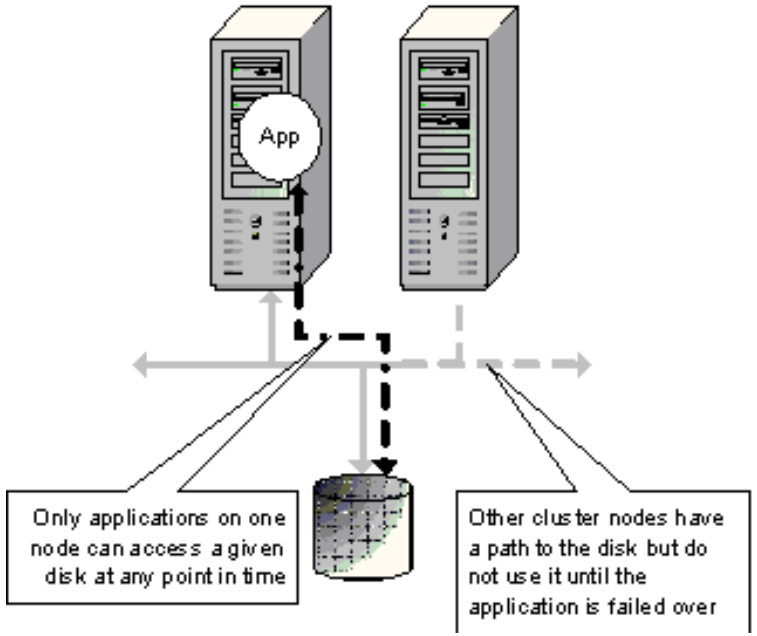
- 2 architectures of High Availability Clustering exist:
  - **Shared Nothing** $\Longrightarrow$ Distributed Storage
  - **Shared Disk** $\Longrightarrow$ Shared Storage



Shared Nothing Architecture

Image Source: technet.microsoft.com

- In a **Shared Nothing** cluster, each node has its own storage resource
- Even, when a resource is physically connected to multiple nodes, only a single node is allowed to access it
  - Only if a node fails, the resource is acquired by another node



- Advantage: No lock management is required
  - No protocol overhead reduces the performance
  - In theory, the cluster can scale almost in a linear way
- Drawback: Higher financial effort for storage resources, because the data can not be distributed in an optimal way

Shared Nothing with DRBD (1/3)

- Distributed Replicated Block Device (DRBD)
  - Free software to build up a network storage for Shared Nothing clusters, without an expensive Storage Area Network (SAN)
- Shared storage is always a single point of failure, since only the cluster nodes are designed in a redundant way
  - Redundant SAN solutions are expensive ($>$ 100.000 €)

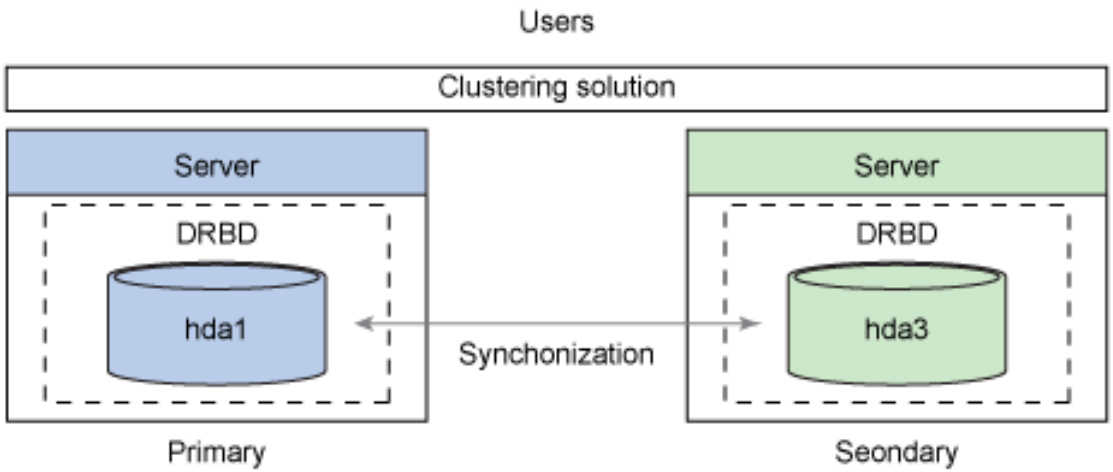


Image Source: M. Jones, `https://www.ibm.com/developerworks/library/l-drbd/index.html`

Well written articles about DRBD

iX 3/2010. Florian Haas. *Hochverfügbare Shared Nothing Cluster mit DRBD*. P.120-123
M. Tim Jones. High availability with the Distributed Replicated Block Device. 2010.
`https://www.ibm.com/developerworks/library/l-drbd/`

Shared Nothing with DRBD (2/3)

- Functioning:
  - A primary server and a secondary server exist
    - Write requests are carried out by the primary server and afterwards are send to the secondary server
    - Only if the secondary server reports the successful write operation to the primary server, the primary server reports the end of the successful write operation
  - Practically, it implements RAID 1 via TCP
  - Primary server fails $\Longrightarrow$ secondary server becomes primary server
    - If a failed system is operational again, only the data blocks, which have changed during the outage are resynchronized
  - Read access is always carried out locally ($\Longrightarrow$ better performance)

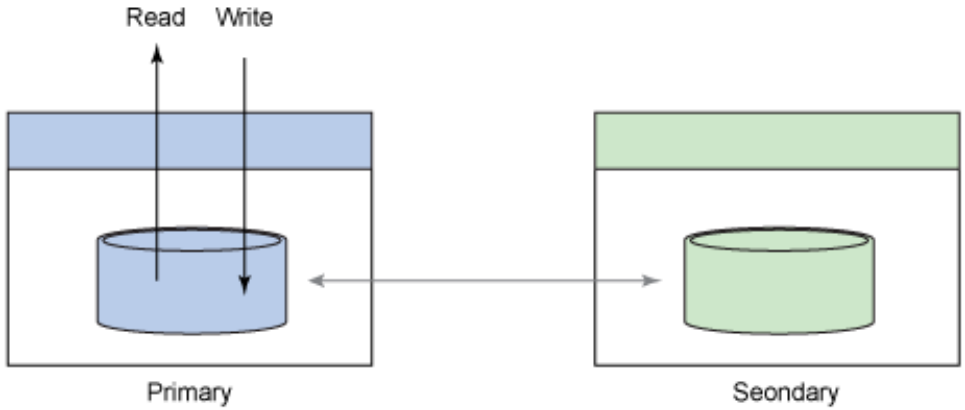


Image Source: M. Jones, `https://www.ibm.com/developerworks/library/l-drbd/index.html`

Shared Nothing with DRBD (3/3)

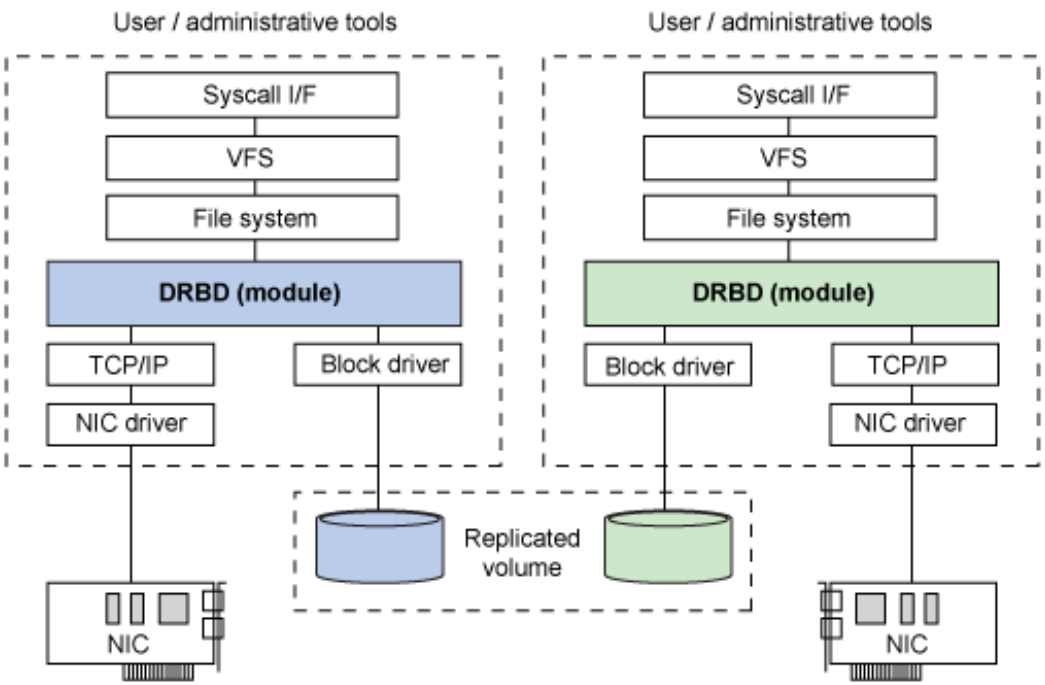


Image Source: M. Jones, `https://www.ibm.com/developerworks/library/l-drbd/index.html`

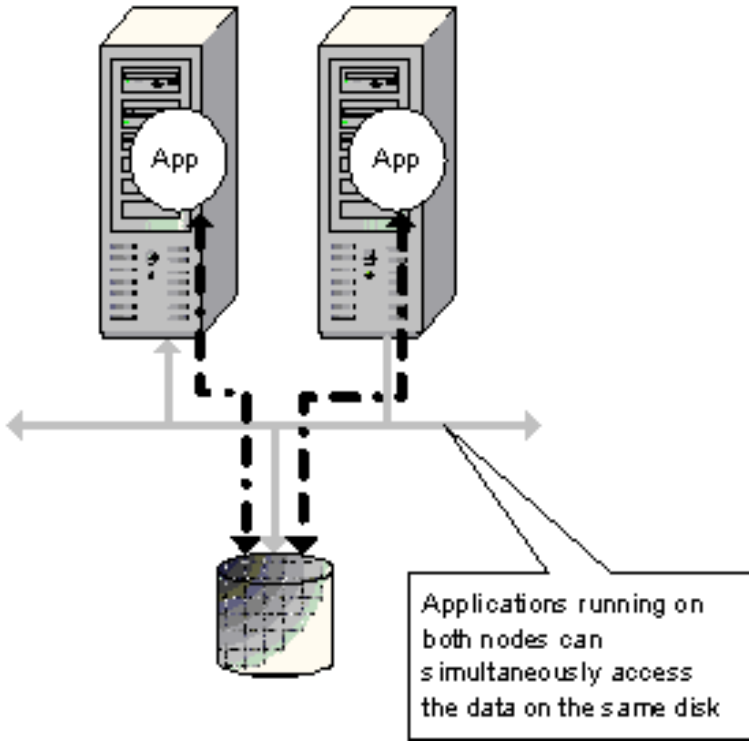
- DRBD is a part of the Linux kernel since version 2.6.33 (February 2010)
- Because DRBD operates inside the Linux kernel at block level, the system is transparent for the layers above it

- DRBD can be used as a basis for:
  - Conventional file systems, such as ext3/4 or ReiserFS
  - Shared-storage file systems, such as Oracle Cluster File System (OCFS2) and Global File System (GFS2)
    - If shared-storage file systems are used, all nodes must have direct I/O access to the device
  - Another logical block device, such as the Logical Volume Manager (LVM)

Shared Disk Architecture

Image Source: technet.microsoft.com

- In a **Shared Disk** cluster, all nodes have access to a shared storage
- Several possible ways exist to connect the nodes to the storage:
- **SAN** (Storage Area Network) via Fibre Channel
  - Expensive, but provides high performance
  - Provides block-level access to storage devices via the network.
- **NAS** (Network Attached Storage)
  - Easy-to-use file server
  - Provides file system-level access to storage devices via the network
  - Can also be implemented as a pure software solution
    - Examples: FreeNAS and Openfiler
- **iSCSI** (Internet Small Computer System Interface)
  - SCSI protocol via TCP/IP
  - SAN-like access via the IP-network



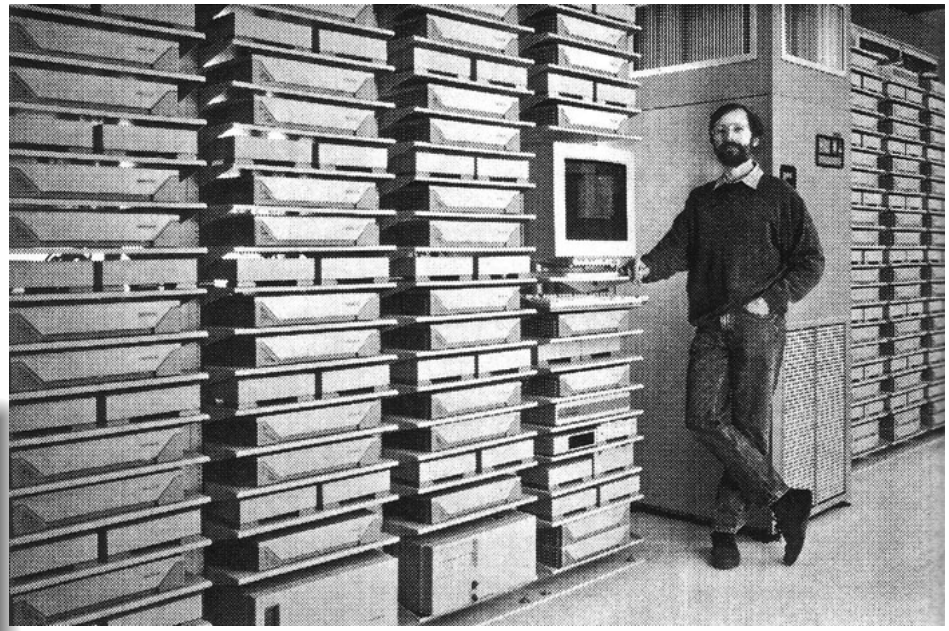
High Performance Clustering (1/2)

- Objective: High computing power
  - Also called: **Clustering for Scalability**
- High Performance Clusters provide the performance of mainframe computers for a much lower price
- These clusters are usually made of commodity PCs or workstations
- Typical application area:
  - Applications, which implement the Divide and Conquer principle
    - Such applications split big tasks into multiple sub-tasks, evaluates them and puts together the sub-task results to the final result
  - Applications, used for analyzing large amounts of data

High Performance Clustering (2/2)

Image Source: Reddit

Application examples: Crash test simulation, weather forecast, optimization of components, Monte Carlo simulation, flight path calculation, data mining, flow simulation, strength analysis, rendering of movies or clips, simulation of the night sky, variant calculating for chess, prime number computation,. . .



In 1995 Pixar rendered Toy Story on a 294 x 100MHz CPU Sun SPARCstation 20 cluster

Each SPARCstation 20 (single processor) had SunOS 5.4 installed and a HyperSPARC 100 MHz with 27.5066 MFLOPS
$\Longrightarrow$ The theoretical maximum performance of the setup was 294 * 27.5066 = 8086.94 MFLOPS

- Advantages:
  - Low price and vendor independence
  - Defective components can be obtained in a quick and inexpensive way
  - It is easy to increase the performance in a short time via additional nodes
- Drawback:
  - High administrative and maintenance costs, compared with mainframes

High Performance Clustering – Beowulf Cluster

- If a free operating system is used $\Longrightarrow$ **Beowulf** cluster
- If a Windows operating system is used $\Longrightarrow$ **Wulfpack**
- A Beowulf cluster is never a cluster of workstations (COW)
  - Beowulf clusters consist of commodity PCs or workstations, but the nodes of a Beowulf cluster are used only for the cluster
- The cluster is controlled via a master node
  - The master distributes (schedules) jobs and monitors the worker nodes
- Worker nodes are only accessible via the network connection
  - They are not equipped with I/O devices like screens or keyboards
- Worker nodes contain commodity PC components and are not redundant ($\Longrightarrow$ designed for high availability)
  - A potential issue is the failure of the cooling of the system components
  - Fans in nodes and power supplies have a limited lifetime and fail without any warning
  - Modern CPUs cannot operate without adequate cooling

Stone SouperComputer (1/2)

Image source: `http://www.climatemodeling.org/~forrest/linux-magazine-1999/`

- Example for a Beowulf cluster, made of discarded office computers
- `http://www.extremelinux.info/stonesoup/`

Stone SouperComputer (2/2)

- Built in 1997
- Mostly 486DX-2/66 Intel CPUs
- Some Pentiums
- 10 Mbit/s Ethernet
- RedHat Linux, MPI and PVM
- Extremely heterogeneous structure
- No purchase costs
- High setup and administration effort
- Everything handmade

Image source: `http://www.climatemodeling.org/~forrest/linux-magazine-1999/`

Later Generations of Beowulf Clusters (1/2)

Image source: `http://archiv.tu-chemnitz.de/pub/2000/0089/data/clic.html`

- Vendors, such as Megware in Chemnitz, sell complete Beowulf clusters
- Image: Chemnitzer LInux Cluster (CLIC) from 2000

Later Generations of Beowulf Clusters (2/2)

Image source: `http://tina.nat.uni-magdeburg.de`

- Tina (Tina is no acronym) in Magdeburg from 2001

State of the Art of Cluster Computing

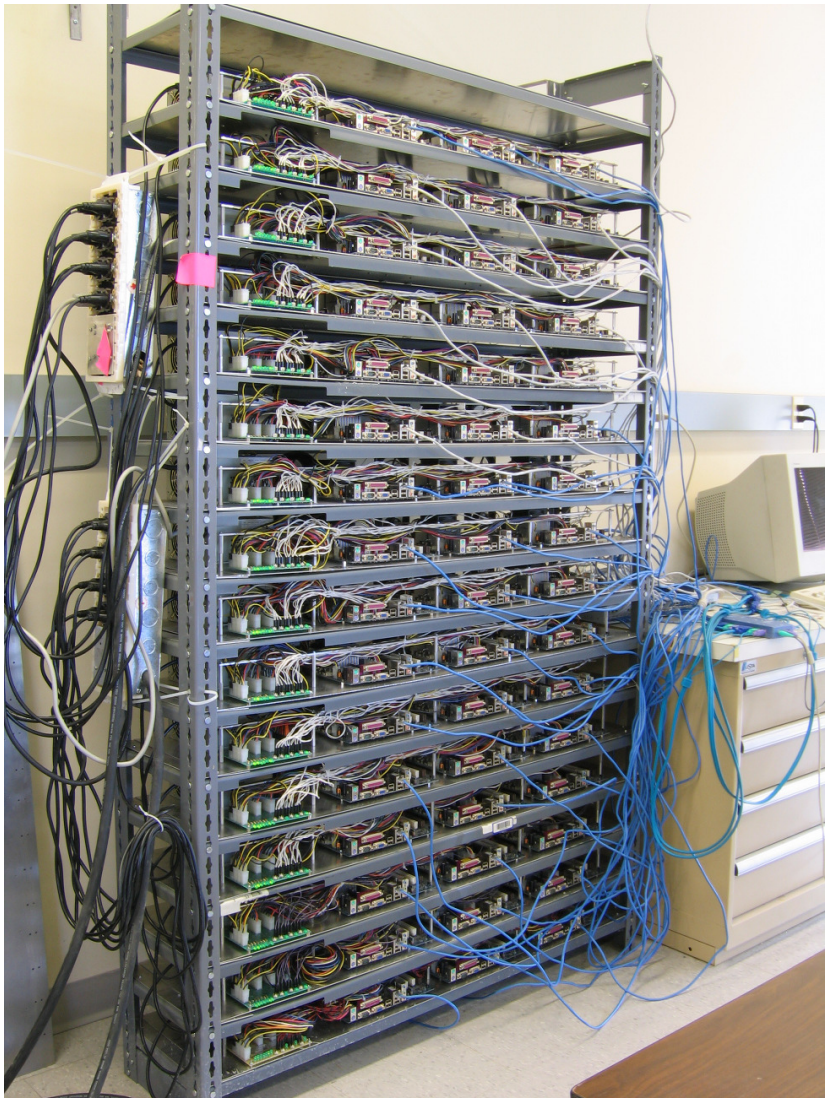


Image source (right image):
`http://physics.bu.edu/~sandvik/clusters.html`

A Cluster of Motherboards

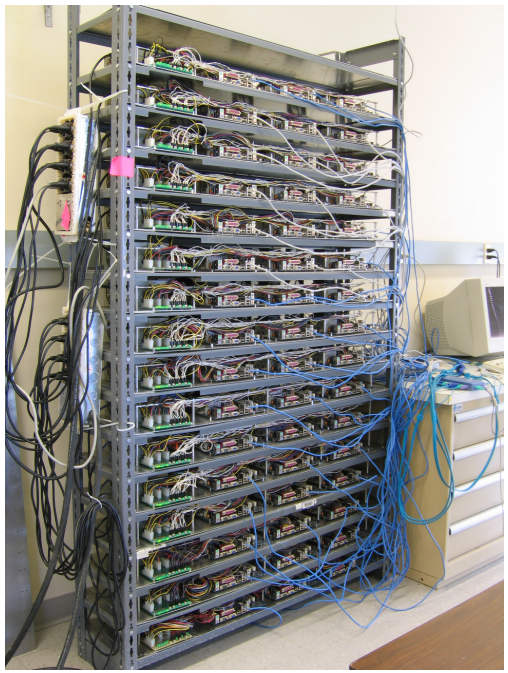
The cluster in the right image has 48 nodes

High Throughput Clustering

- Objective: Maximize throughput
- Such clusters consist of servers, which are used to process incoming requests
- Such clusters are not used for extensive calculations
  - Tasks must not be split into sub-tasks
  - The individual tasks (requests) are small and a single PC could handle them
- Typical fields of application of High Throughput Clustering:
  - Web servers
  - Internet search engines
- Large compute jobs $\Longrightarrow$ High Performance Cluster
- Multiple small compute jobs (in a short time) $\Longrightarrow$ High Throughput Cluster

Today: Clusters at Universities

http://cs.boisestate.edu/~amit/research/beowulf/



http://physics.bu.edu/~sandvik/clusters.html

- Beowulf clusters, built up from commodity hardware
  $\Longrightarrow$ low acquisition cost
- High effort for administration (handmade)
  $\Longrightarrow$ irrelevant, because students do the administration

Today: Research and Industry (Example: HP C7000)

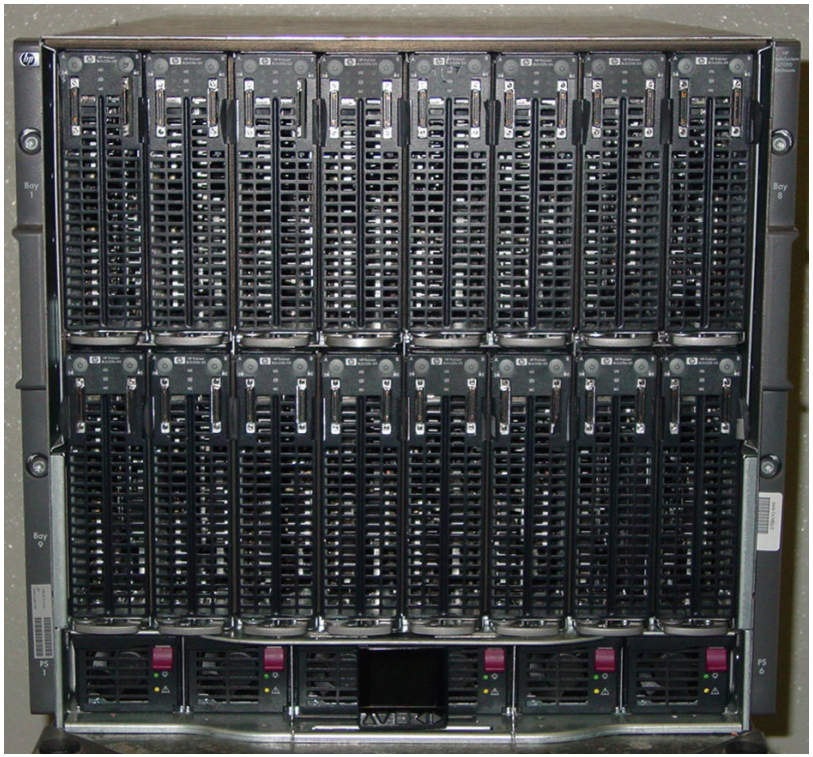


Image source: `http://imagehost.vendio.com/bin/imageserver.x/00000000/pdneiman/DSC04040.JPG`

- Compact blade servers or so-called *pizza boxes*
- Professional management tools (like HPE iLO) and redundant components simplify the administration

Calculation Example about the Possible Packing Density

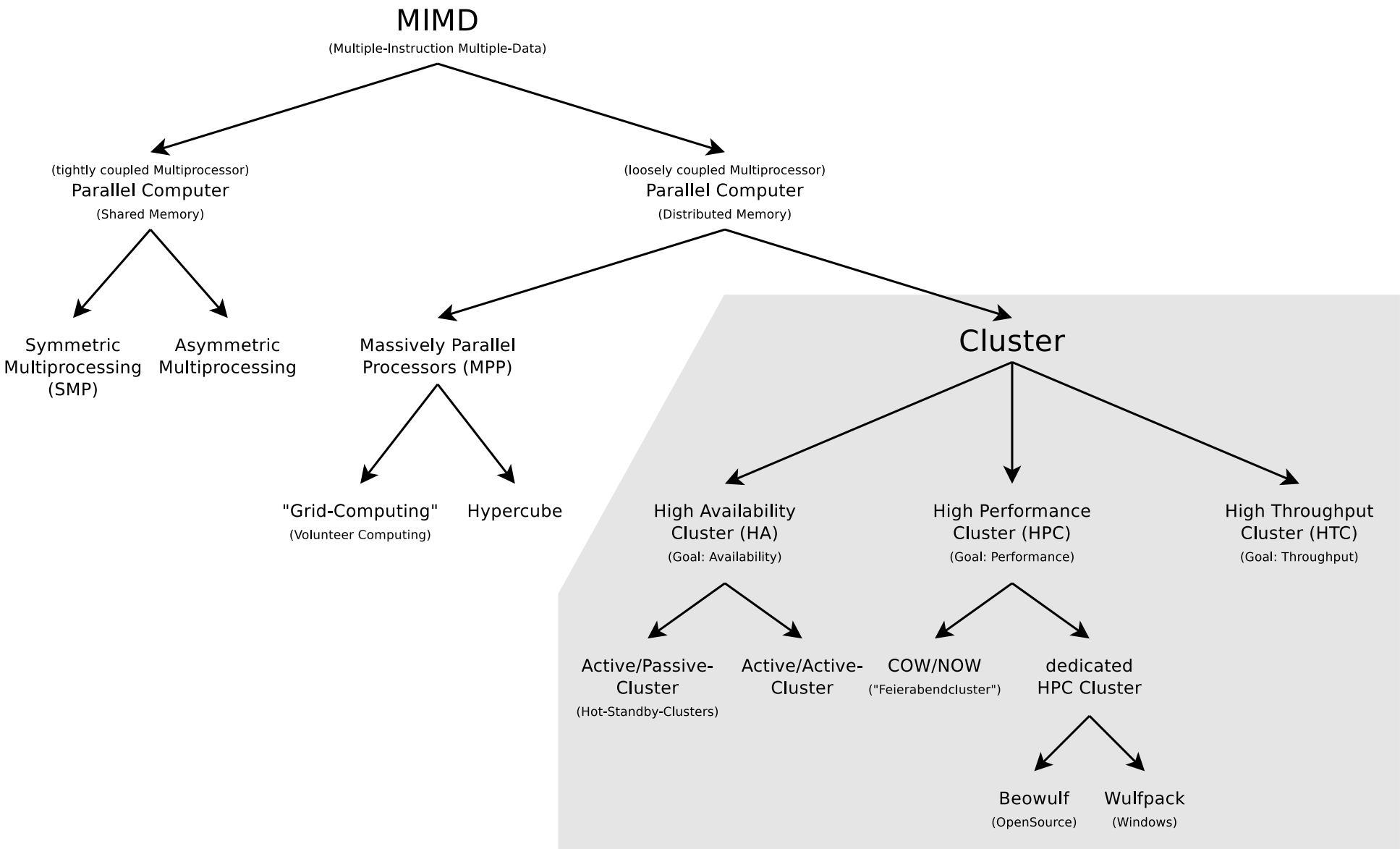
- A 19 inch rack contains up to 4 blade enclosures (BladeCenters)
- A HP C7000 BladeCenter provides 16 blade slots
- Blades exist, which conain 2 independent servers
  - e.g. HP Blade ProLiantBL2x220c G5
  - 2 servers per blade. Completely independent computers
  - Each server contains: 2x Intel Quad Core Xeon (2,33 GHz) and 16 GB RAM

$\Longrightarrow$ 8 cores per server
$\Longrightarrow$ 16 cores per blade
$\Longrightarrow$ 256 cores per blade enclosure (BladeCenter)
$\Longrightarrow$ 1024 cores per 19 inch rack

The packing density increases

Intel Xeon processors with 6 cores (*Dunnington*), with 8 cores (*Nehalem-EX*), with 18 cores (*Haswell-EX*) and with 22 cores (*Broadwell*) are already available. AMD offers the Opteron (*Magny-Cours*) with 12 cores and the Ryzen (*Threadripper*) with 32 cores

Classification of Clusters



Advantages and Drawbacks of Clusters

Advantages:

- Flexibility and extensibility
  - The number of nodes of a cluster can be dynamically increased or decreased according to the needed resources
- Lower purchase price campared with supercomputers
- Simple replacement of commodity hardware components

Drawbacks:

- Errors occur more often compared with a single supercomputer
- Clusters consist of many independent systems
  $\Longrightarrow$ higher administrative costs and personnel expenses compared with a single or few supercomputers
- High effort for distributing and controlling applications
  - If the number of nodes is increased, the effort increases too

Libraries for Cluster Applications

- For the development of cluster applications, several libraries exist, which provide **message passing** functionality, independent of the cluster structure and the specific hardware
- Popular message passing systems:
  - **Parallel Virtual Machine** (PVM)
  - **Message Passing Interface** (MPI)

Libraries for Cluster Applications (PVM)

- **Parallel Virtual Machine (PVM)**
- Developed from 1989 onwards
- Provides a uniform programming interface for the creation of a parallel computing unit with distributed memory
- Supports C/C++ and Fortran 77/90
- Consists of a daemon, libraries and tools
  - PVM is not a programming language!
- Especially suited for heterogeneous environments
  - Converts e.g. automatically between little endian and big endian
- Focus is not performance, but portability

Good PVM book (freely available in PostScript and HTML)

Janusz Kowalik (1994). **PVM: Parallel Virtual Machine**.
A Users' Guide and Tutorial for Networked Parallel Computing.
`http://www.netlib.org/pvm3/book/pvm-book.html`

Libraries for Cluster Applications (MPI)

- **Message Passing Interface (MPI)**
- Development started in 1993-94
- Collection of functions (e.g. for process communication) to simplify the development of applications for parallel computers
- The library can be used with C/C++, Fortran 77/90 and Python
  - MPI is not a programming language!
- Contains no daemon
- Implements message-based communication (message passing)
- Especially suited for homogeneous environments
- Focus: Performance and security
- MPI implements > 100 functions and several constants
- Implementations: LAM/MPI (obsolete), OpenMPI, MPICH2,...

MPI tutorial of Wes Kendall, Dwaraka Nath, Wesley Bland: `https://mpitutorial.com/tutorials/`
MPI tutorial of Stefan Schaefer and Holger Blaar: `http://www2.informatik.uni-halle.de/lehre/mpi-tutorial/index.htm`

MPI Functions – Selection of important Functions (1/5)

- `MPI_Init(&argc,&argv)`
  - Initialization routine $\Longrightarrow$ starts the MPI environment
  - Defines the communicator `MPI_COMM_WORLD`
    - A communicator contains a group of processes and a communication context
    - `MPI_COMM_WORLD` contains all processes
  - The arguments `argc` and `argv` are pointers to the parameters of the main function `main`
    - The `main` function always receives 2 parameters from the operating system
    - `argc` (**arg**ument **c**ount) contains the number of parameters passed
    - `argv[]` (**arg**ument **v**alues) contains the parameters itself
    - The names of the variables can be freely selected, but they are usually named `argc` and `argv`
    - Not command-line parameters passed $\Longrightarrow$ `argc = 1`

`https://web.archive.org/web/20070909174518/http://www2.informatik.uni-jena.de/cmc/racluster/mpi-leitfaden`

MPI Functions – Selection of important Functions (2/5)

- `MPI_Comm_Size(MPI_Comm comm, int size)`
  - Determines the number of processes in a communicator
  - `size` is the output

```
1 #include "mpi.h"
2
3     int        size;
4     MPI_Comm   comm;
5     ...
6     MPI_Comm_size(comm, &size);
7     ...
```

MPI Functions – Selection of important Functions (3/5)

- `MPI_Comm_Rank(MPI_Comm comm, int rank)`
  - Determines the rank (identification number) of the calling process in the communicator
  - `rank` is the output
  - The rank is used by MPI for process identification
  - The rank number is unique within a communicator
  - Processes are numbered sequentially, starting from zero

```
1  #include "mpi.h"
2
3      int        rank;
4      MPI_Comm   comm;
5
6      ...
7      MPI_Comm_rank(comm, &rank);
8      if (rank==0) {
9          ... code for process 0 ...
10     }
11     else {
12         ... code for the other processes ...
13     }
```

https://web.archive.org/web/20070909174518/http://www2.informatik.uni-jena.de/cmc/racluster/mpi-leitfaden

MPI Functions – Selection of important Functions (4/5)

- `MPI_Get_processor_name(char *name, int *resultlen)`
  - Determines the name of the processor
  - `name` is the output
  - The length (number of characters) of the name is returned in `resultlen`
  - The name identifies the hardware, where MPI runs
    - The exact output format is implementation-dependent and may by equal with the output of `gethostname`

```
1 #include "mpi.h"
2 int MPI_Get_processor_name(
3         char *name,
4         int *resultlen)
```

MPI Functions – Selection of important Functions (5/5)

- `MPI_Finalize()`
  - Stops the MPI environment
  - All processes need to call `MPI_Finalize()`, before they kill themselves
- `MPI_Abort(MPI_Comm comm, int errorcode)`
  - Terminates the MPI environment
  - `comm` = Communicator (handle), whose processes are terminated
  - `errorcode` = Error code, which is returned to the calling environment

```
1  #include "mpi.h"
2
3  int main(int argc, char *argv[]) {
4
5      int         errorcode;
6      MPI_Comm    comm;
7
8      ...
9      MPI_Abort(comm, errorcode);
10     ...
11 }
```

Simple MPI Example (1/3)

- Start a MPI cluster with 3 nodes (1 master, 2 slaves) in EC2
  - Start 3 instances (ami-06ad526f) with Ubuntu 11:04 in US-East
- Install the required packages in all instances:

```
$ sudo apt-get -y install make gcc g++ openmpi-bin openmpi-common libopenmpi-dev
```

- Generate public key on the master:

```
$ ssh-keygen -t rsa
```

- Append the contend of `.ssh/id_rsa.pub` (master) to `.ssh/authorized_keys` (slaves)
- Insert into `/etc/hosts` on the master:

```
10.252.186.133    domU-12-31-38-00-B5-77.compute-1.internal    master
10.223.49.141     domU-12-31-38-07-32-63.compute-1.internal    node1
10.253.191.213    domU-12-31-38-01-B8-27.compute-1.internal    node2
```

- Create file `hosts.mpi` on the master with this content:

```
master
node1
node2
```

Simple MPI Example (2/3) – Say Hello to the Processors

```c
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  // variable definitions
  int size, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  // Start MPI environment
  MPI_Init(&argc, &argv);

  // How many processes contains the MPI environment?
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // What is out number we in the MPI environment?
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // What is the name of the processor?
  MPI_Get_processor_name(processor_name, &namelen);

  // Output of each process
  printf("Ich bin Prozess Nr. %d von %d auf %s\n", rank, size, processor_name);

  // Stop MPI environment
  MPI_Finalize();

  // Kill application with exit code 0 (EXIT_SUCCESS)
  return 0;
}
```

Simple MPI Example (3/3)

- Compile program:

```
$ mpicc hello_world.c -o hello_world
```

- Distribute the program in the cluster:
  - The program must be stored on each node in the same directory!

```
$ scp hello_world node1:~
$ scp hello_world node2:~
```

- Program execution (6 processes) in the cluster:

```
$ mpirun -np 6 --hostfile hosts.mpi hello_world
Ich bin Prozess Nr. 0 von 6 auf domU-12-31-38-00-20-38
Ich bin Prozess Nr. 1 von 6 auf ip-10-126-43-6
Ich bin Prozess Nr. 2 von 6 auf domU-12-31-38-00-AD-95
Ich bin Prozess Nr. 4 von 6 auf ip-10-126-43-6
Ich bin Prozess Nr. 3 von 6 auf domU-12-31-38-00-20-38
Ich bin Prozess Nr. 5 von 6 auf domU-12-31-38-00-AD-95
```

- The CPUs respond in random order
  - What is the reason?

MPI Functions – Send-/Receive (1/3)

- `MPI_Send(int buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Sends a message (blocking) to another process in the communicator
    - `buffer` = first address of the transmit buffer
    - `count` = number of elements in the transmit buffer (not negative)
    - `datatype` = MPI data type of the elements in the transmit buffer
    - `dest` = rank of the receiver process in the communicator
    - `tag` = ID for distinguishing the messages
    - `comm` = communicator
  - All parameters are input parameters
  - The function sends `count` data objects of type `datatype` from address `buffer` ($\Longrightarrow$ transmit buffer) with the ID `tag` to the process with rank `dest` in communicator `comm`

MPI Data Types

MPI data type	C data type	Used for...	Size	Value range
MPI_CHAR	signed char	Chars	1 Byte	-127 ... +127
MPI_UNSIGNED_CHAR	unsigned char	Chars	1 Byte	0 ... 255
MPI_SHORT	signed short int	Integers	2 Bytes	-32.768 ... 32,767
MPI_UNSIGNED_SHORT	unsigned short int	Integers	2 Bytes	0 ... 65.535
MPI_INT	signed int	Integers	2-8 Bytes	Depends on the architecture
MPI_UNSIGNED	unsigned int	Integers	2-8 Bytes	Depends on the architecture
MPI_LONG	signed long int	Integers	4 Bytes	-2.147.483.648 ... 2.147.483.647
MPI_UNSIGNED_LONG	unsigned long int	Integers	4 Bytes	0 ... 4.294.967.295
MPI_FLOAT	float	Floating point numbers	4 Bytes	Single precision
MPI_DOUBLE	double	Floating point numbers	8 Bytes	Double precision
MPI_LONG_DOUBLE	long double	Floating point numbers	16 Bytes	Quadruple precision
MPI_BYTE	—	Floating point numbers	1 Byte	0 ... 255

- The integer value range depends on the used C compiler used and architecture (2, 4 or 8 Bytes)

MPI Functions – Send-/Receive (2/3)

- `MPI_Recv(int buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status status)`
  - Receive a message (blocking)
    - `buffer` = first address of the receive buffer $\Longleftarrow$ *output parameter*
    - `count` = number of elements in the receive buffer (not negative)
    - `datatype` = MPI data type of the elements in the receive buffer
    - `source` = rank of the sender process in the communicator or `MPI_ANY_SOURCE`
    - `tag` = ID for distinguishing the messages. For receiving messages with any identifier, the constant `MPI_ANY_TAG` is used
    - `comm` = communicator
    - `status` = contains the rank of the sender process `source` and the message identifier `tag` $\Longleftarrow$ *output parameter*

MPI Functions – Send-/Receive (3/3)

- `MPI_Get_count(status, datatype, count)`
  - Determines the number of received elements
    - `count` = number of received elements (not negative) $\Longleftarrow$ *output parameter*
    - `status` = status upon the return of the receive operation
    - `datatype` = MPI data type of the elements in the receive buffer

```
1  #include "mpi.h"
2  #define MAXBUF  1024
3
4      int             i, count;
5      void            *recvbuf;
6      MPI_Status      status;
7      MPI_Comm        comm;
8      MPI_Datatype    datatype;
9
10     ...
11     MPI_Recv(recvbuf, MAXBUF, datatype, 0, 0, comm, &status);
12     MPI_Get_count(&status, datatype, &count);
13     for (i=0; i&ltcount; i++) {
14         ...
15     }
16     ...
```

Simple MPI Example (1/2) – Send and Receive

```
1  #include "mpi.h"
2  #include <stdio.h>
3
4  int main(int argc, char *argv[]) {
5    int size, rank, dest, source, rc, count, tag=1;
6    char inmsg, outmsg='x';
7    MPI_Status Stat;
8    MPI_Init(&argc,&argv);
9    MPI_Comm_size(MPI_COMM_WORLD, &size); // Get number of processes in the MPI environment
10   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get nrocess number in the MPI environment
11
12   if (rank == 0) {
13     dest = 1;
14     source = 1;
15     MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
16     MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
17     }
18   else if (rank == 1) {
19     dest = 0;
20     source = 0;
21     MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
22     MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
23     }
24
25   MPI_Get_count(&Stat, MPI_CHAR, &count);
26   printf("Task %d: Received %d char(s) from task %d with tag %d \n",
27          rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
28   MPI_Finalize();                    // Stop MPI environment
29   return 0;                          // Kill application with exit code 0 (EXIT_SUCCESS)
30 }
```

Simple MPI Example (2/2) – Send and Receive

Source of the example

http://coewww.rutgers.edu/www1/linuxclass2005/lessons/lesson13/sec_8.html

- Compile program:

```
$ mpicc sendrecv.c -o sendrecv
```

- Distribute the program in the cluster:
  - The program must be stored on each node in the same directory!

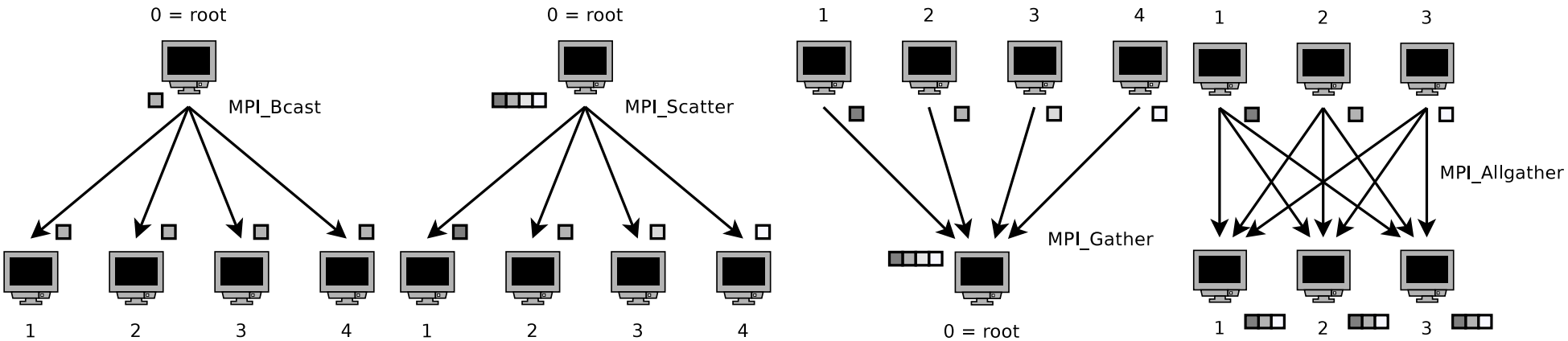
```
$ scp sendrecv node1:~
$ scp sendrecv node2:~
```

- Program execution (2 processes) in the cluster:

```
$ mpirun -np 2 --hostfile hosts.mpi sendrecv
Task 0: Received 1 char(s) from task 1 with tag 1
Task 1: Received 1 char(s) from task 0 with tag 1
```

More Send and Receive Operations

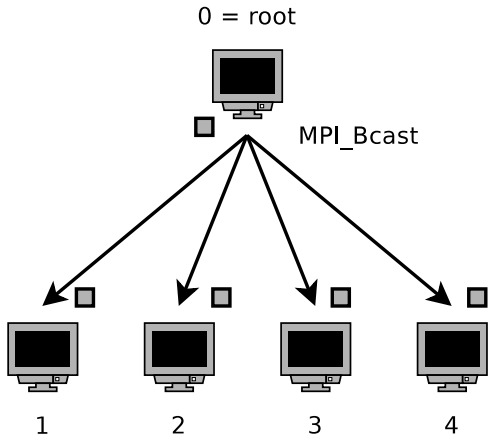
- `MPI_Send` and `MPI_Recv` implement *one-to-one* communication
- `MPI_Bcast` and `MPI_Scatter` implement *one-to-many* communication
- `MPI_Gather` implements *many-to-one* communication
- `MPI_Allgather` implements *many-to-many* communication



The functions `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather` and `MPI_Allgather` must be called by all processes in the communicator!

MPI Functions – Broadcast Sending (1/2)

- `MPI_Bcast(int buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`



- Send a message of process `root` to all other processes in the communicator
  - `buffer` = first address of the transmit buffer
  - `count` = number of elements in the transmit buffer (not negative)
  - `datatype` = MPI data type of the elements in the transmit buffer
  - `root` = rank of the sender process in the communicator
  - `comm` = communicator
- All processes in the communicator must call the function

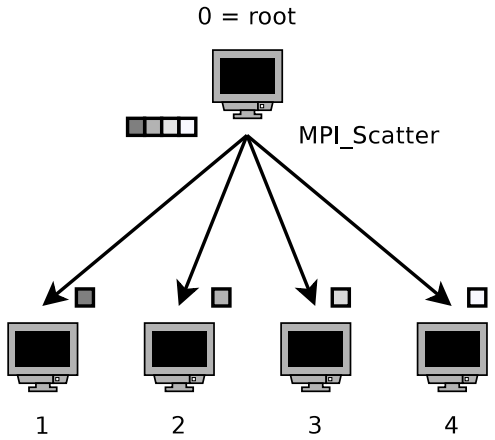
MPI Functions – Broadcast Sending (2/2)

```
1  #include "mpi.h"
2  #define ROOT 0
3
4      int    myid, *buffer, bufsize;
5
6      ...
7      MPI_Comm_rank(MPI_COMM_WORLD, &myid);
8      if (myid==ROOT) {
9          ... get or create data ...
10         MPI_Bcast(buffer, bufsize, MPI_INT, ROOT, MPI_COMM_WORLD);
11         ...
12     }
13     else {
14         ...
15         buffer=malloc(bufsize * sizeof(int));
16         MPI_Bcast(buffer, bufsize, MPI_INT, ROOT, MPI_COMM_WORLD);
17         ...
18     }
19     ...
```

https://web.archive.org/web/20140113174355/http://www.tu-chemnitz.de/informatik/RA/projects/mpihelp/mpi_fkt_liste.html

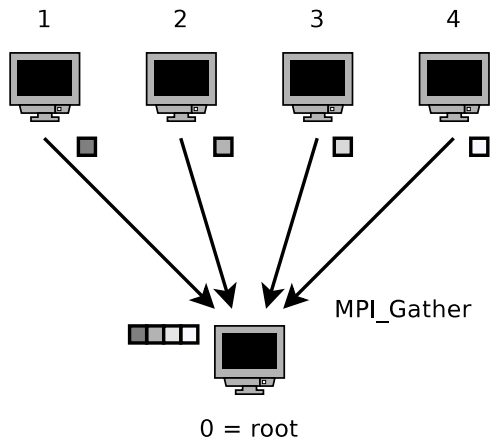
MPI Functions – Scatter

- `MPI_Scatter(int sendbuf, int sendcount, MPI_Datatype sendtype, int recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`



- While `MPI_Bcast` sends the same piece of data to all processes, `MPI_Scatter` sends chunks of an array to different processes
  - `sendbuf` + `recvbuf` = first address of the transmit/receive buffer
  - `sendcount` + `recvcount` = number of elements in the transmit/receive buffer (not negative and typically equal size)
  - `sendtype` + `recvtype` = MPI data type of the elements in the transmit/reiceive buffer
  - `root` = rank of the sender process in the communicator
  - `comm` = communicator
- All processes in the communicator must call the function

MPI Functions – Gather

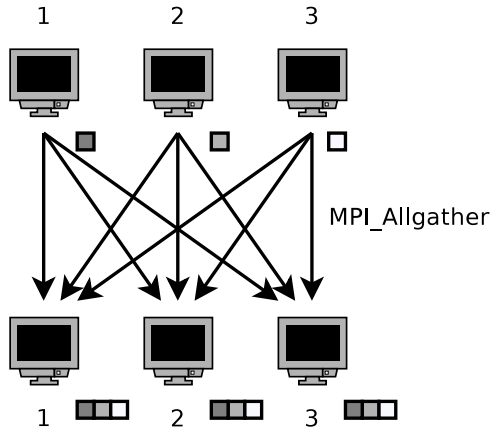


- `MPI_Gather(int sendbuf, int sendcount, MPI_Datatype sendtype, int recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
- `MPI_Gather` is the inverse of `MPI_Scatter`
- Instead of distributing elements from one process to many processes, `MPI_Gather` takes elements from many processes and gathers them to one single process.
- All parameters are equal to `MPI_Scatter`
- All processes in the communicator must call the function

Source: `http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/`

MPI Functions – Allgather

- `MPI_Allgather(int sendbuf, int sendcount, MPI_Datatype sendtype, int recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`



- Given a set of elements distributed across all processes, `MPI_Allgather` will gather all of the elements to all the processes.
- `MPI_Allgather` is basically an `MPI_Gather` followed by an `MPI_Bcast`
- All parameters are equal to `MPI_Scatter` and `MPI_Gather` with the difference that there is no root process in `MPI_Allgather`
- All processes in the communicator must call the function

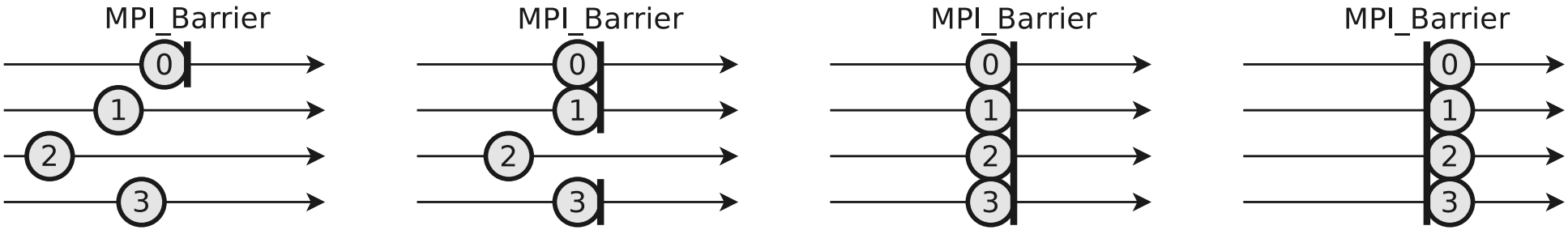
Source: `http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/`

MPI Functions – Barrier

- `MPI_Barrier(MPI_Comm comm)`
  - Blocks the execution of the calling process, until all processes in the communicator `comm` have called the barrier function
    - `comm` = communicator

```
1 #include "mpi.h"
2
3     MPI_Comm comm;
4
5     ...
6     MPI_Barrier(comm);
7     ...
```

- `MPI_Barrier` can be used to synchronize a program



1) Process 0 calls MPI_Barrier first

2) Some other processes arrive at the barrier

3) All processes have arrived at the barrier

4) All processes continue to run

https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

Time Measurements in MPI

- `double MPI_Wtime( void )`
  - Provides a number of seconds as a double-precision floating-point number
  - Time measurements require multiple calls of this routine
    - comm = communicator

```
1  #include "mpi.h"
2
3      double       starttime, endtime, time_used;
4
5      ...
6      starttime=MPI_Wtime();
7      ... program part, whose time will be measured ...
8      endtime=MPI_Wtime();
9      time_used=endtime-starttime;
10     ...
11 }
```

https://web.archive.org/web/20140113174355/http://www.tu-chemnitz.de/informatik/RA/projects/mpihelp/mpi_fkt_liste.html

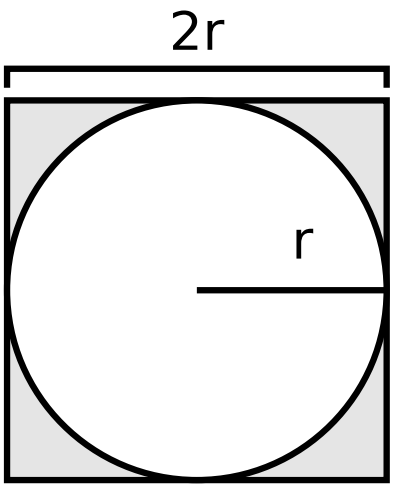
Reduces Values on all Processes to a single Value

- `MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
  - Reduces values on all processes to a single value on process `root`
    - `sendbuf` = address of send buffer (input parameter)
    - `recvbuf` = address of receive buffer on `root` (output parameter)
    - `count` = number of elements in the transmit buffer (not negative)
    - `datatype` = MPI data type of the elements in the transmit buffer
    - `op` = reduce operation
    - `root` = rank of the root process in the communicator
    - `comm` = communicator (all processes in the communicator must call the function)

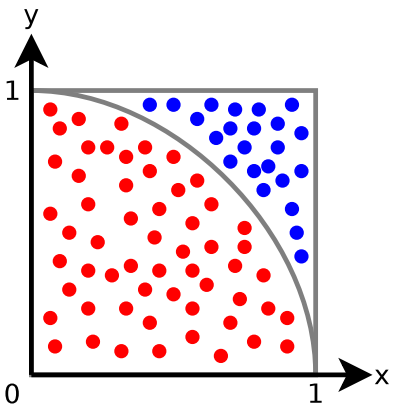
The reduction operations defined by MPI include:

MPI_MAX (Returns the maximum element)
MPI_MIN (Returns the minimum element)
MPI_SUM (Sums the elements)
MPI_PROD (Multiplies all elements)
MPI_MAXLOC (Returns the maximum value and the rank of the process that owns it)
MPI_MINLOC (Returns the minimum value and the rank of the process that owns it)

Example: Calculation of π via Monte Carlo Simulation



r = Radius
A = Surface ratio
C = Circle
S = Square



- Inscribe a circle of radius r inside a square with side length $2r$
- Generate random dots in the square
  - The number of dots in A_C in relation to the number of dots in A_S is equal to the surface ratio

$$\frac{A_C}{A_S} = \frac{\pi \cdot r^2}{(2 \cdot r)^2} = \frac{\pi \cdot r^2}{4 \cdot r^2} = \frac{\pi}{4}$$

- The dots can be generated (X/Y axis values via `random`) in parallel by the workers
- The master receives from each worker the number of calculated dots in A_C and calculates:

$$\frac{4 \cdot \text{dots in } A_C}{\text{dots in } A_S} = \pi$$

MPI Example – Calculate π (1/3)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include "mpi.h"
5
6  int main(int argc, char *argv[]) {
7
8    int myid,numprocs;
9
10   double PI25DT = 3.14159265358979323846264643;
11   double t1, t2;
12
13   long long npts = 1e11;
14   long long i,mynpts;
15
16   long double f,sum,mysum;
17   long double xmin,xmax,x;
18
19   // Initialization routine => starts the MPI environment
20   // Defines the communicator MPI_COMM_WORLD
21   MPI_Init(&argc,&argv);
22   // Determines the number of processes in a communicator
23   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
24   // Determines the rank (id) of the calling process in the communicator
25   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

This Source Code is influenced a lot by this Source...

https://web.archive.org/web/20160812014841/http://chpc.wustl.edu/mpi-c.html

MPI Example – Calculate π (2/3)

```
1  // Data decomposition. Each process gets a part of the work
2  mynpts = npts/numprocs;
3
4  if (myid == 0) {
5    // Returns the time in seconds since an arbitrary time in the past
6    t1 = MPI_Wtime();
7  }
8
9  mysum = 0.0;
10 xmin = 0.0;
11 xmax = 1.0;
12
13 // Seed the pseudo random number generator
14 srand(time(0));
15
16 for (i=0; i<mynpts; i++) {
17   // (long double)rand()/(long double)RAND_MAX
18   // returns a random number between 0 and 1.
19   // (long double)rand()/(long double)RAND_MAX*(xmax-xmin)
20   // returns a random number between 0 and max - min.
21   // the whole expression will return a random number between 0+min and min+(max-min)
22   // => between min and max.
23   x = (long double)rand()/(long double)RAND_MAX*(xmax-xmin) + xmin;
24   // Each process does a partial sum over its own points.
25   mysum += 4.0/(1.0 + x*x);
26 }
```

How to generate a random number between 0 and 1?

https://stackoverflow.com/questions/6218399/how-to-generate-a-random-number-between-0-and-1/6219525

MPI Example – Calculate π (3/3)

```
1   // Take all the processes values of mysum and add them up into sum on process 0.
2   MPI_Reduce(&mysum,&sum,1,MPI_LONG_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
3
4   if (myid == 0) {
5     // Returns the time in seconds since an arbitrary time in the past
6     t2 = MPI_Wtime();
7
8     f = sum/npts;
9
10    printf("Pi calculated with %lld points. \n",npts);
11    printf("Pi calculated:       %.16f \n",f);
12    printf("Correct value of Pi: %.16f \n",PI25DT);
13    printf("Error is:            %.16f \n",fabs(f-PI25DT));
14    printf("Elapsed time [s] for the relevant part of the program: %f\n", t2 - t1);
15  }
16
17  // Stop the MPI environment
18  MPI_Finalize();
19 }
```

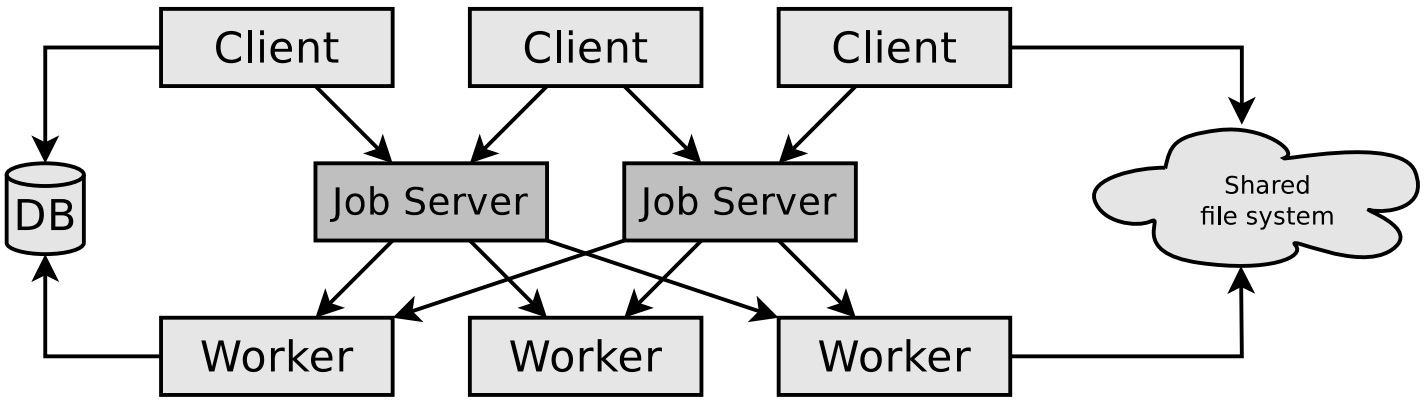
```
$ time mpirun -np 512 --hostfile hosts_4cores_128.mpi  /mnt/cluster_128/pi
Pi calculated with 100000000000 points.
Pi calculated:       3.1415785751520118
Correct value of Pi: 3.1415926535897931
Error is:            0.0000140784377813
Elapsed time [s] for the relevant part of the program: 37.651207

real    0m46.394s
user    0m18.860s
sys     0m3.020s
```

Gearman

http://www.gearman.org

- Framework for developing distributed applications
  - Free Software (BSD License)
  - Supports C, Pearl, PHP, Python, C#, Java, .NET and UNIX shell
- Assigns one of 3 roles to every computer involved
  - **Clients** transfer jobs to the Job Servers
  - **Job Server** assign jobs of the clients to the Workers
  - **Worker** register themselves at Job Servers and execute jobs



Gearman

- The name *Gearman* is an anagram for *manager*
  - Garman only distributes jobs
- Gearman should only be used in secure private networks
  - The communication is not encrypted and uses port 4730
  - No mechanism for the authentication of the systems is implemented
- Clients and workers access shared data
  - Cluster file systems like GlusterFS or protocols such as NFS or Samba can be used

Helpful article about Gearman (in German language)

Garman verteilt Arbeit auf Rechner im LAN, *Reiko Kaps*, c't 24/2010, P.192

- The next slides contain an application example from the article

Gearman – Example of a Worker Script

- Client and worker both, access via `/src/media` a shared file system
  - The shared file system contains images that need to be resized
- The workers scale via ImageMagick `convert`
- Shell script `resizer-worker.sh`

```
#/bin/bash
INFILE="$1"

echo "Converting ${INFILE} on $HOSTNAME" >> /src/media/g.log

convert "${INFILE}" -resize 1024 "${INFILE}"-small.jpg
```

- Register the worker script (`-w`) at the Job Server „gman-jserver“ (`-h`) with the function name „Resizer“ (`-f`):
  - `gearman -h gman-jserver -w -f Resizer xargs resizer-worker.sh`

Gearman – Example of a Client Job

- This command starts the image processing

```
find /srv/media/images -name "*.jpg" \
-exec gearman -b -h gman-jserver -f Resizer {} \;
```

- `find` searchs for JPG images in the path `/srv/media/images`
- Via `-exec`, the file path is submitted to the Gearman client
- The client submits the file path to the Job Server, who passes it with the function `Resizer` to the worker
- Because of the argument `-b`, the jobs are executed in background and the client is released immediately