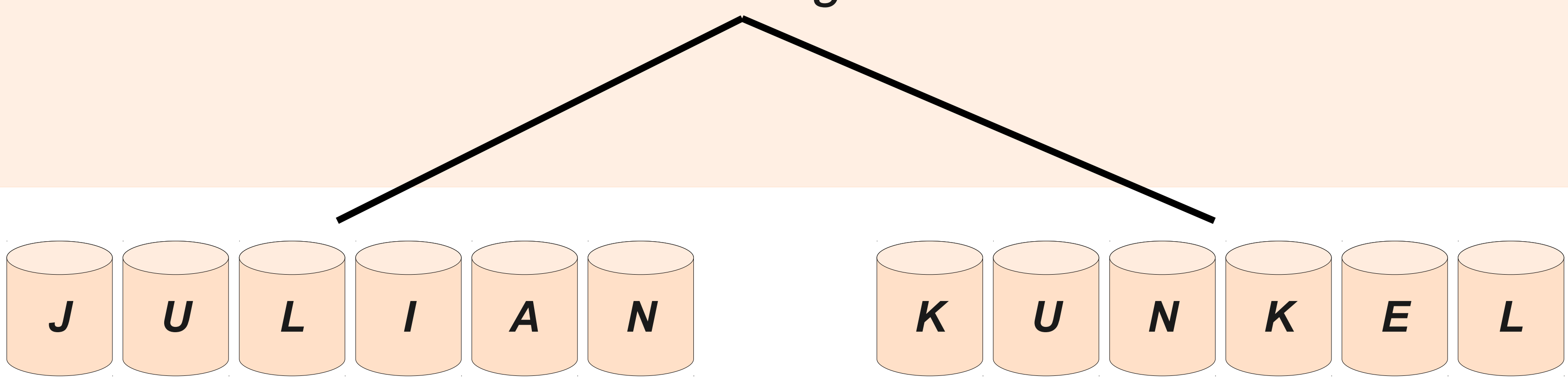
Parallel Programming with Respect to I/O

Heidelberg 2011



Agenda Overview

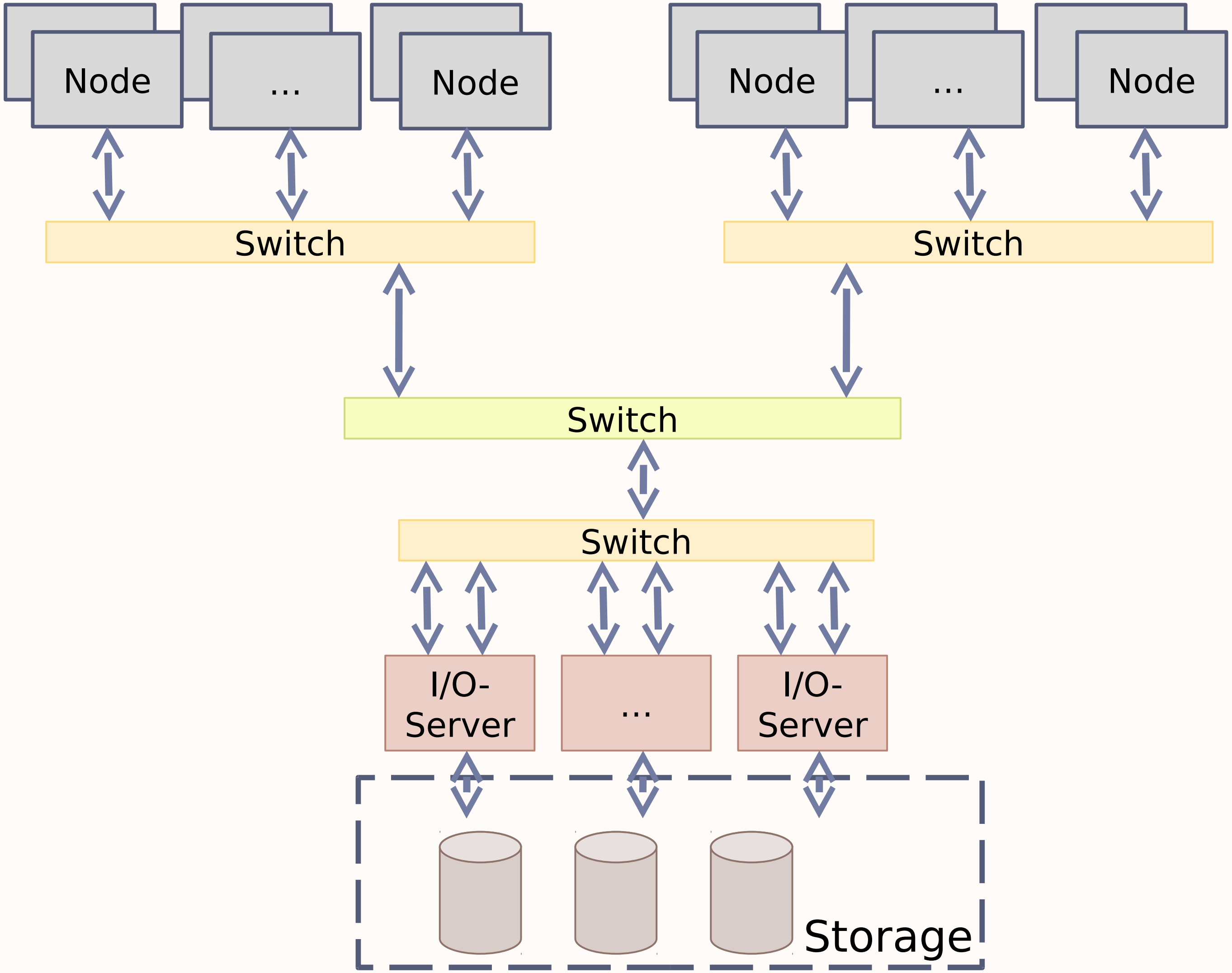
- 12.01 – Morning
  - Introduction to Parallel Programming
  - Performance Estimation
- 12.01 – Afternoon
  - Storage & File System concepts
- 13.01 – Morning
  - Programming (Parallel) I/O

Agenda

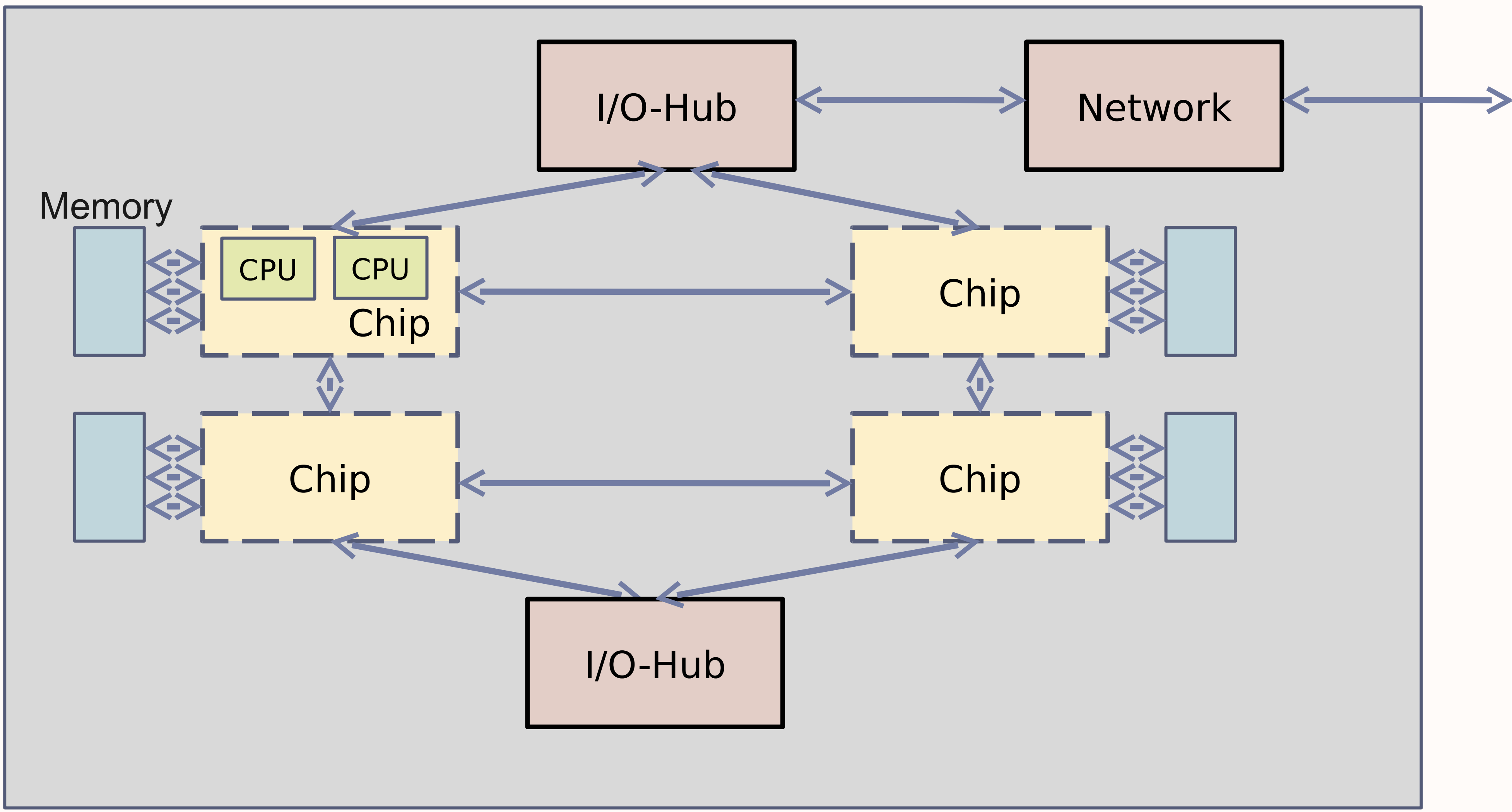
- Beowulf Clusters
- Parallel Processing in a Nutshell
  - State-of-the-art interfaces
  - Development cycle of parallel programs
  - Work partitioning
- Performance analysis of parallel programs
  - Theory
  - Hardware impact
  - Performance estimations

Introduction

Exemplary Beowulf Cluster



Cluster Node

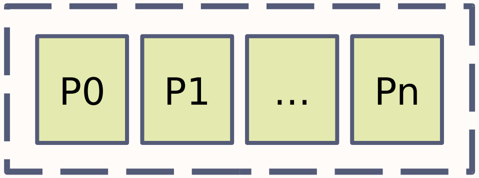


Parallel Processing

- Hardware provides resources
  - Computation
  - Storage
  - Network
- Program utilizes resources to solve a problem
- Communication enables collaboration
- Established standards are MPI and OpenMP

Message Passing Interface (MPI)

- Defines a standard API for inter-process communication
- Communication & I/O are programmed explicitly
  - A programmer is responsible for proper usage
  - Be careful with the semantics of the standard!
  - Processes are enumerated:



- An MPI implementation realizes the standard
  - By providing libraries and an execution environment
  - MPI maps the processes to the existing hardware

MPI Example

- Problem: Compute 200 independent values
  - For example compute f(x) for x in {1, … 200}
- Two processes collaborate
  - Process 0 performs I/O and user interaction
  - Both compute 100 values
  - Process 1 sends results to process 0
  - Process 0 outputs results

MPI Schematic Example

```
#include "mpi.h"
Int main(int argc, char **argv) {
    int myrank;          /* Used to store the process' unique number among all process */
    int nprocs;          /* Number of processes */
    Int values[200];    /* Data which will be communicated between processes */
    ...
    MPI_Init(&argc, &argv);  /* Initialize MPI to enable inter-process communication */
    /* Determine information about collaborating processes  */
    MPI_Comm_rank(MPI_COMM_WORLD, & myrank);  // My unique number
    MPI_Comm_size(MPI_COMM_WORLD, & nprocs);   // Number of processes
    If (myrank == 0){
        <Compute first 100 values>
         MPI_Recv(values + 100, MPI_INT, 100, 1, 4711, MPI_COMM_WORLD, & status);
    }else if (myrank == 1){
        <Compute second 100 values>
        MPI_Send(values, MPI_INT, 100, 0, 4711, MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

OpenMP

- Semi-automatic parallelization
  - Programmer specifies parallelism in compiler directives
  - OpenMP aware compiler generates parallel code
  - Provides lightweight library to manage parallelism
- Uses threads to utilize available CPUs
  - Works on shared memory machines
- Example code computes vector addition
  - C = A + B

OpenMP Example Code

```
#include <omp.h>
int main (int argc, char *argv[]) {
#pragma omp parallel
  {
  Int tid = omp_get_thread_num();
  if (tid == 0)
    {
    Int nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  #pragma omp for
  for (i=0; i<N; i++) { /* work in this loop is automatically distributed */
    c[i] = a[i] + b[i];
    }
  }  /* end of parallel section */
```

Development of Parallel Programs

- Loop of development:
  1. Parallelization of the algorithm
  2. Implementation
  3. Debugging
  4. Performance analysis & optimization

- Parallelizing an algorithm requires to consider I/O

Partitioning of Work

- Utilize computation resource by assigning work
- Paradigms:
  - Functional decomposition
    - Processes perform different tasks
    - Examples: Pipeline, coupled climate codes
  - Domain decomposition
    - Processes work on different data
    - Examples: Segmenting images, feature extraction
- Flexible, fine grained partitioning is preferable
  - Allows to utilize any number of (different kinds of) CPUs

Performance of Parallel Programs

Performance of Parallel Programs

- Relevant aspects
  - Hardware characteristics
    - Determine how fast resources and communication are
  - Operating system and system configuration
    - How well are local resources utilized
  - Communication and I/O libraries
    - How well could a parallel program utilize the **parallel** computer
  - Code
    - Should perform only computation – relevant for the solution
    - Everything else including communication: **Overhead**

Theoretic Performance Considerations

- Speedup:

$$S_p = \frac{T_1}{T_p}$$

  - Assume a fixed problem
  - Runtime of the best sequential program: T_1
  - Runtime of the parallel program with p CPUs: T_p

- Efficiency:

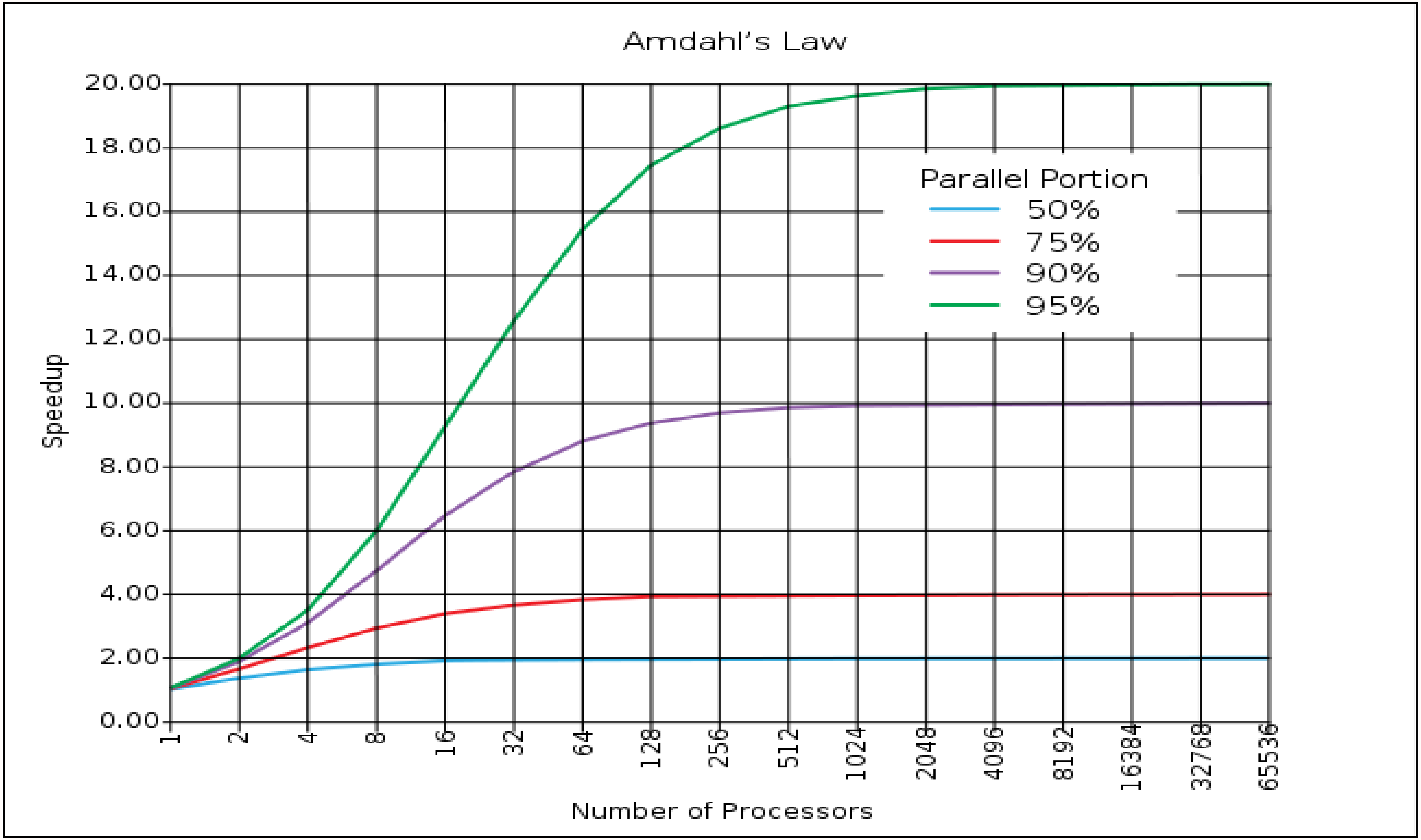
$$E_p = \frac{S_p}{p}$$

- Amdahl's law:

$$S_N \leqslant \frac{1}{(1-P)+\frac{P}{N}}$$

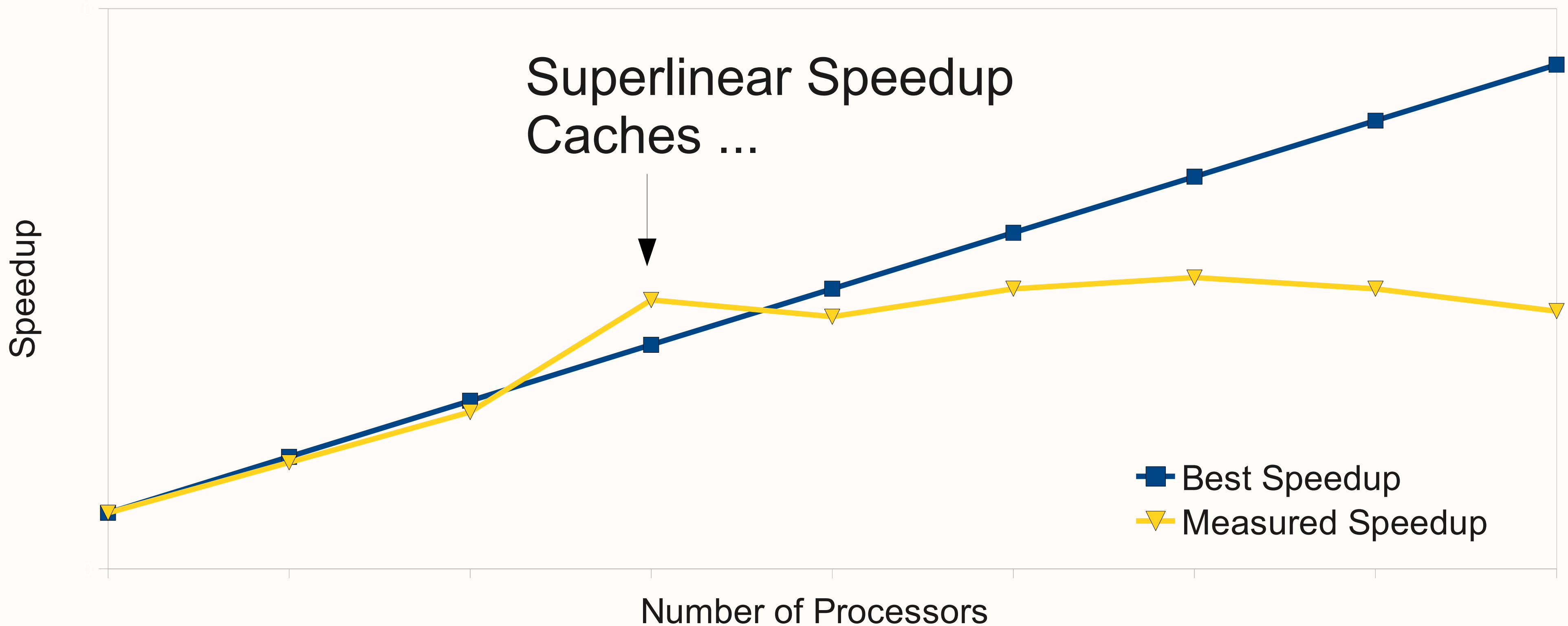
  - P: fraction of parallelizable code
  - N: number of processors

Achievable Speedup



[Image source: Wikipedia]

Realistic Speedup

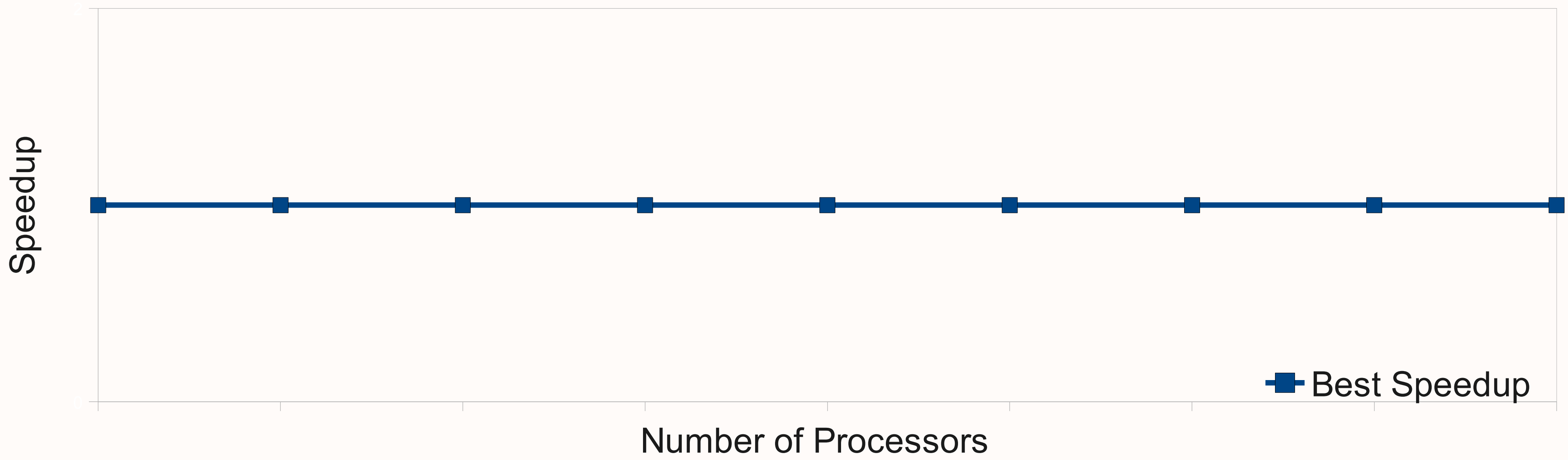


- Scalability of parallel programs is limited!
- At some point overhead > improvement

Scalability

- Scalability:
  - Variation of solution time with the number of processors
- Good scalability:
  - Efficient algorithm & implementation
- Two common terms in HPC:
  - Strong scaling
    - Fixed problem size
    - What we analyzed with Amdahl (Speedup + Efficiency)
    - Scaling of a fixed problem is always limited
  - Weak scaling
    - Fixed problem size **per processor**
    - Easier to achieve than strong scaling

“Speedup” with Weak Scaling



- Speedup definition not appropriate
- Each processor has the same amount of work
- Efficiency more useful:

$$E_p = \frac{T_p}{T_1}$$

Theoretic Performance Considerations

- In Amdahl etc. Hardware was not considered
- Without knowledge of the system we achieve suboptimal performance
  - Gear the algorithm / system / code towards hardware
- Knowledge allows to assess observed performance
- But, hardware components are complex
- Model cluster performance with important characteristics:
  - Simple to understand
  - Sufficient detailed to allow to identify bottlenecks
  - Provides a starting point for further analysis

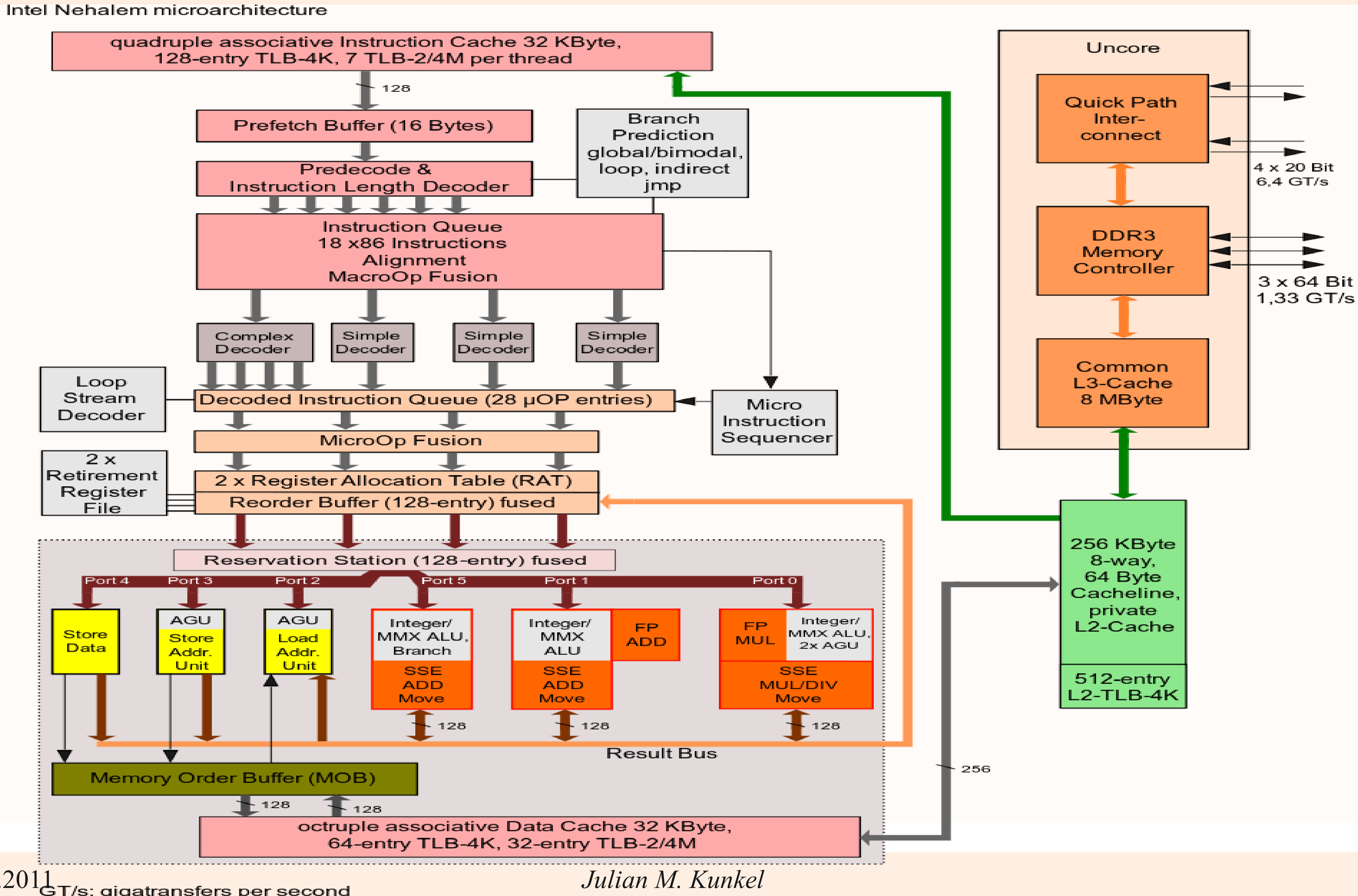
Hardware Characteristics

- How do we obtain hardware characteristics?
  - Vendor information
    - Often optimistic, best case performance
  - Benchmark components
    - Benchmark results are tailored to a given workload
- Assessment of observed performance:
  - How well is a "program" capable to utilize hardware?
  - Software and hardware tries to hide undesired characteristics
  - Measured <= Benchmark <= Theoretic peak

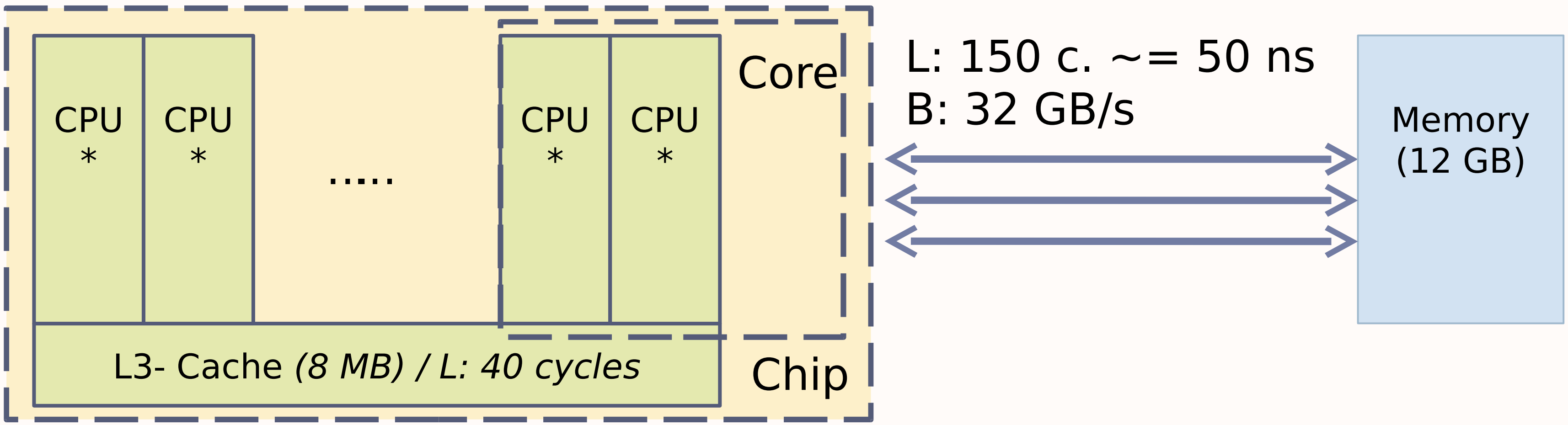
Relevant Characteristics

- Processor
  - Instructions per second
  - Size of L1, L2 and L3 caches
- Memory subsystem
  - Interconnect topology: single bus, bus per chip (e.g. Nehalem)
  - Latency and bandwidth
- I/O-Subsystem
  - Bandwidth (per client node and per server)
  - IOPS – Number of I/O operations per second (for metadata)
- Network
  - Interconnect topology
  - Latency and bandwidth
  - *Remember: communication is considered as overhead!*

Processor Microarchitecture



Logical View of a Processor

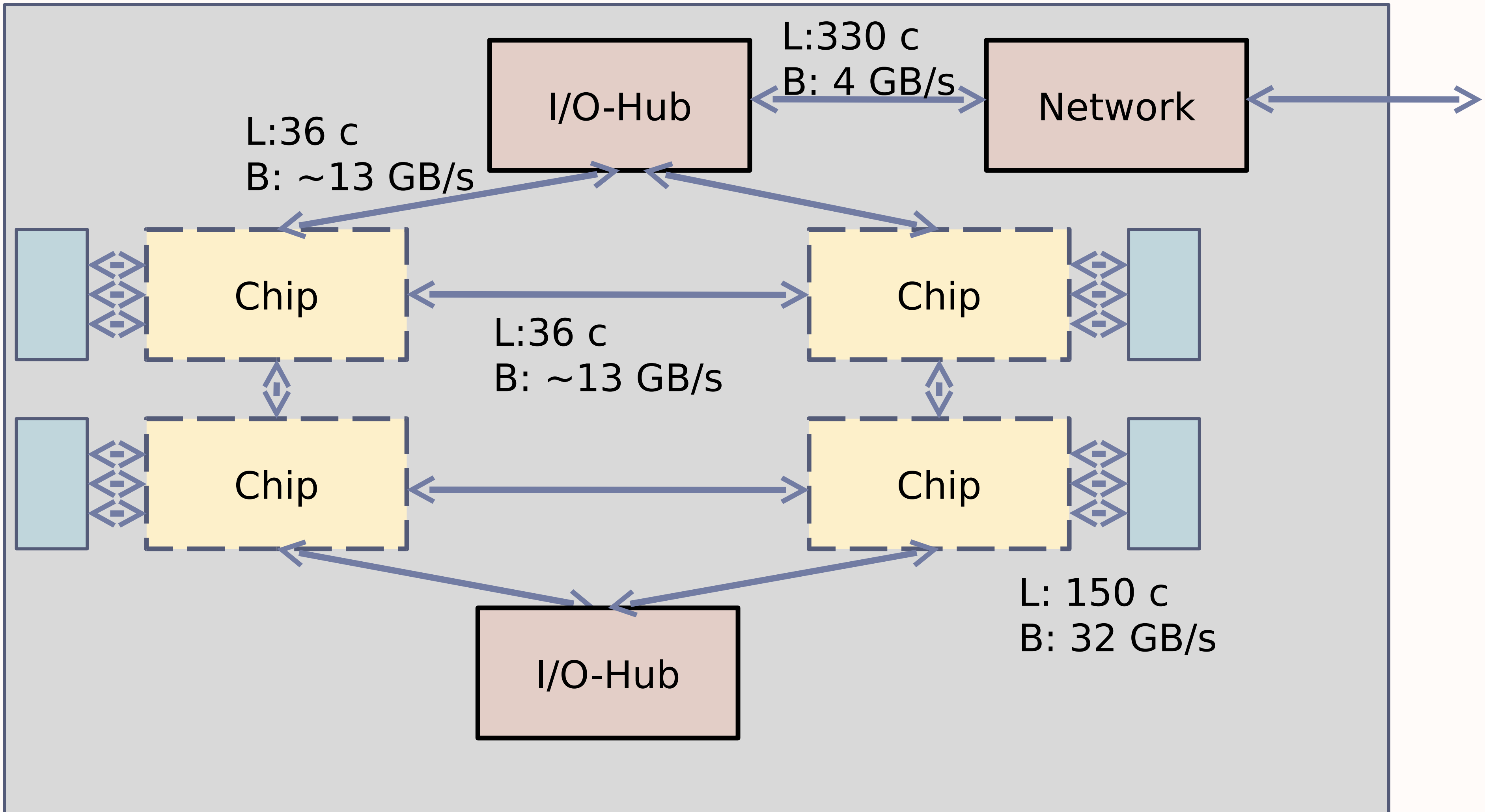


CPU might execute 2 Flop per cycle (Add+Mult)

Per Core:	***Latency:***
L1: 32K Instruction/32K data	*4 cycles*
L2: 256K	*10 cycles*

Data is taken from a Nehalem
@3GHz, depends on Memory

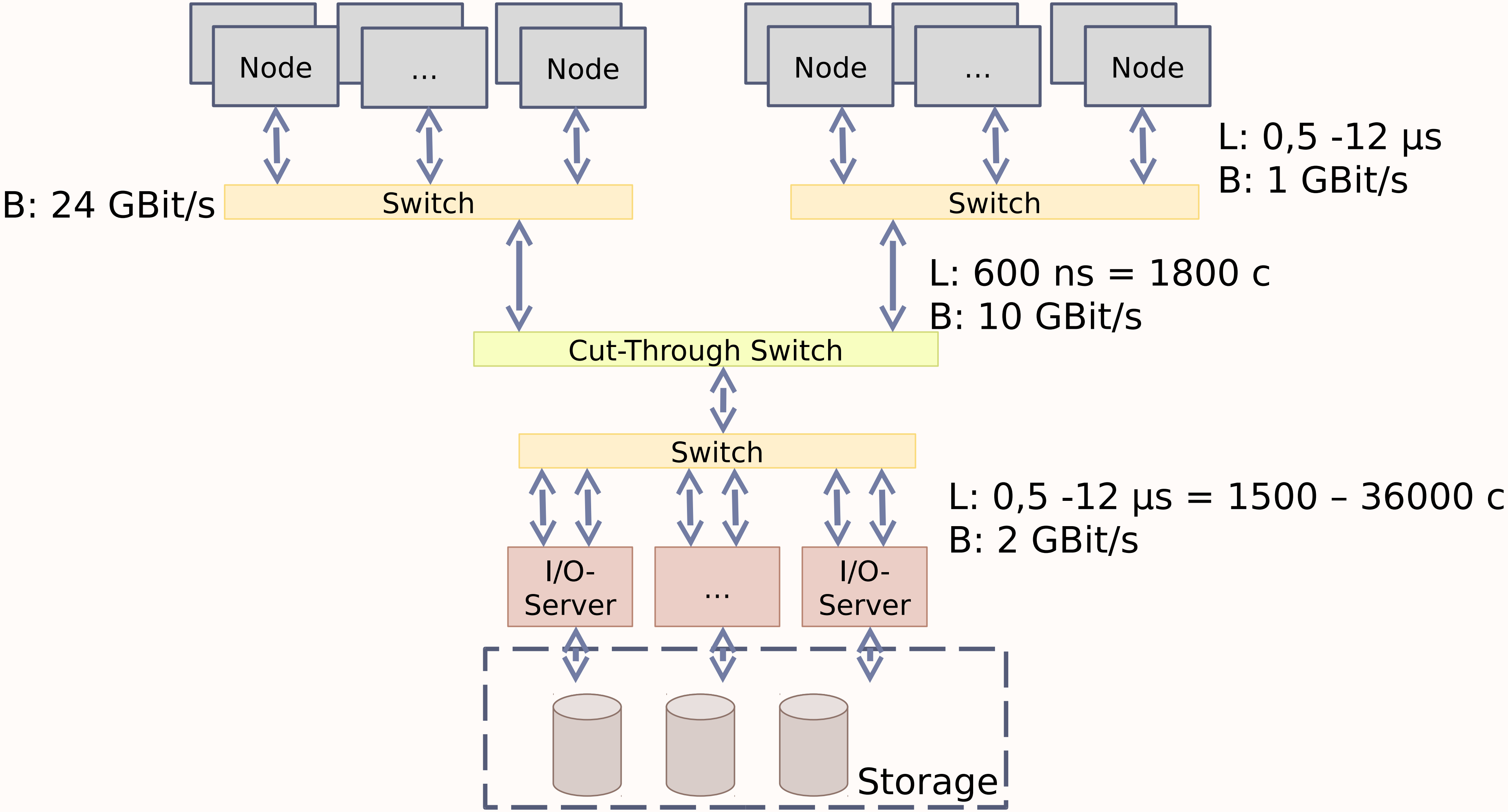
Cluster Node



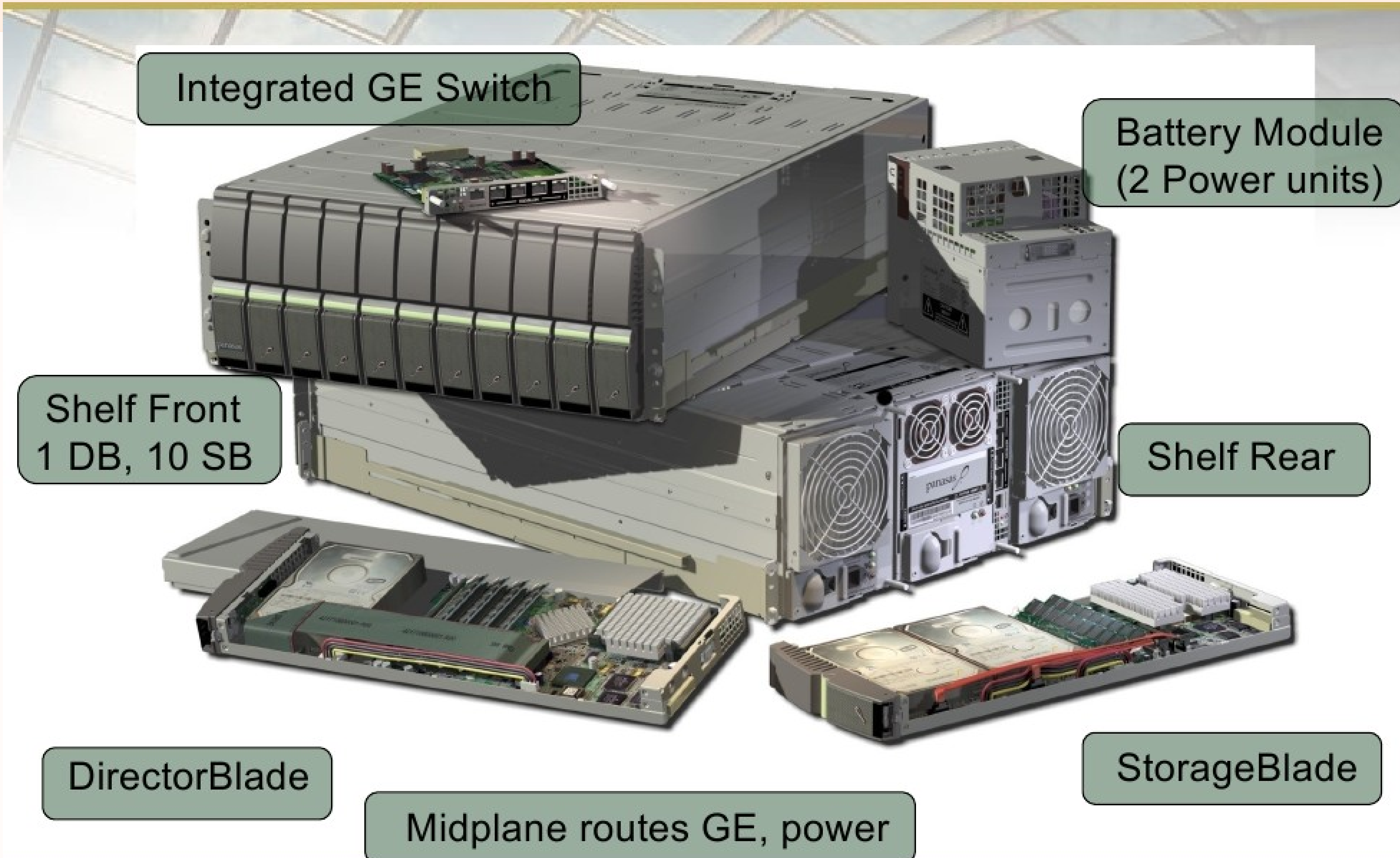
Observations

- Cache sizes vary
- Cache miss is expensive
- NUMA characteristics
- Upper CPUs have faster access to the network
- On some machines: Network may be saturated by simultaneous accesses from multiple CPUs

Exemplary Beowulf Cluster

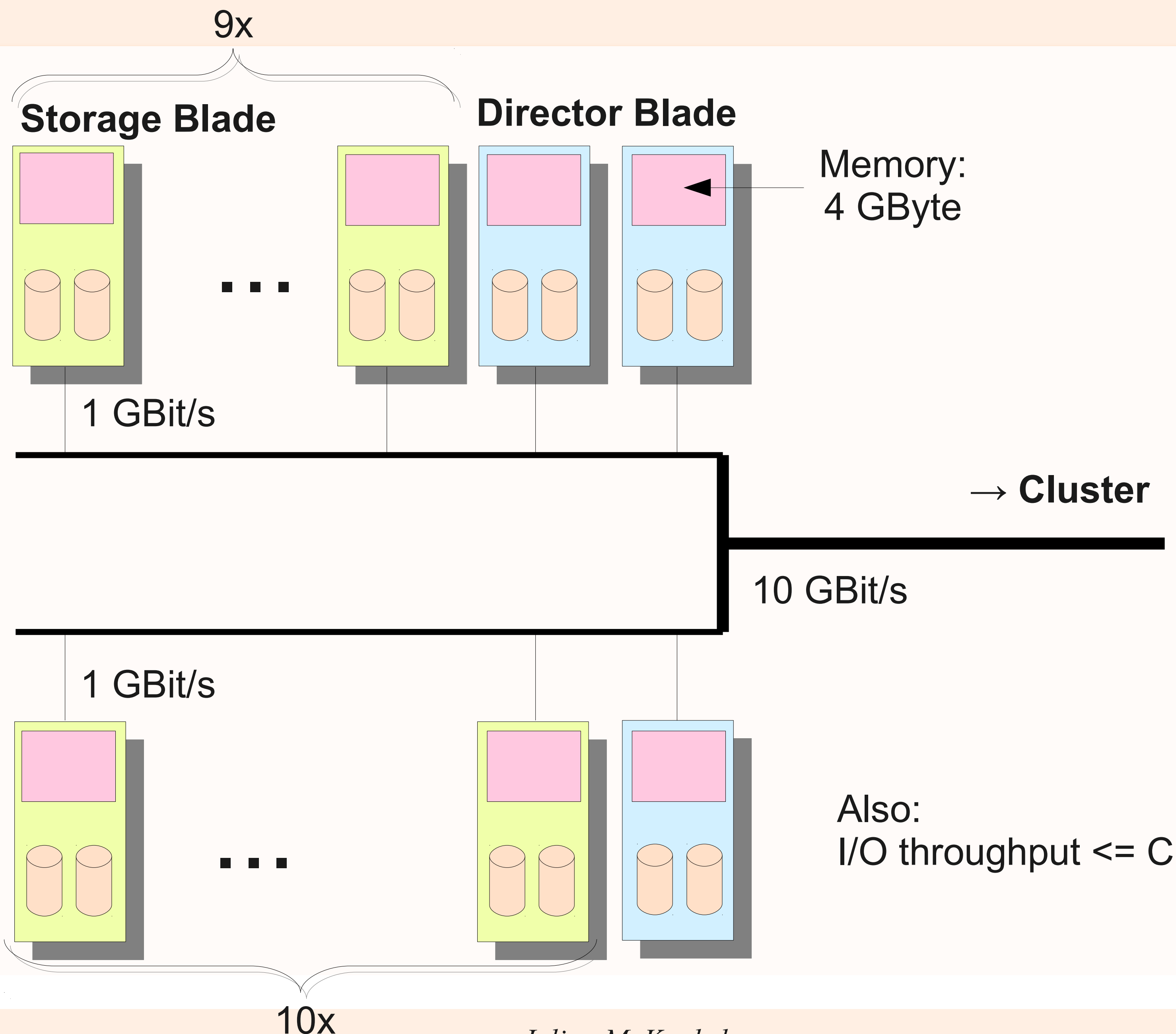


Panasas Hardware



[Taken from: Scalable Performance of the Panasas Parallel File System by Brent Welch]

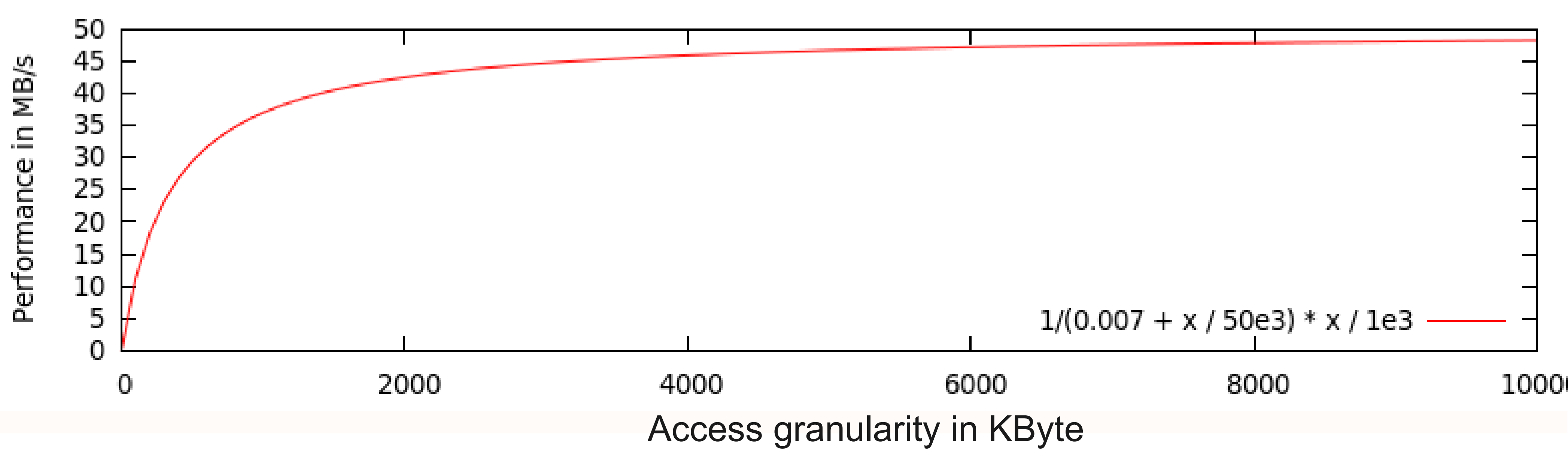
Exemplary Panasas Storage



Also:
I/O throughput <= Client throughput

Theoretical I/O-Performance

- Persistent storage devices
  - Hard disk
    - Latency: ~7 ms = 21.000.000 cycles!
    - Bandwidth: 50 MB/s
      - With access granularity of 100 KB only 11.1 MB/s !



Improving I/O-Performance

- Software and hardware tries to hide I/O penalty
- Caching of data
  - Allows application to continue while I/O completes in the background => Write-behind
  - Allow to aggregate multiple (small) operations into larger operations
  - Read data from disk before it is needed (Read-ahead)
  - *Requires memory! Hiding vs. increased problem size*
- Programming:
  - Overlap I/O (or communication) with computation
    - I/O and communication comes almost for free
    - In the best case a speedup of 2 can be achieved vs. non-overlapped
  - Optimize file format and access pattern (later)

Storage & File Systems

Agenda 2

- Abstraction layers
- Low-level to high-level
- (Parallel) file systems

Abstraction Layers

	Abstraction	Examples
Parallel application	Task	Checkpoint
High-level I/O libraries	Domain specific	HDF5
Low-level I/O interface	Inter-Application/FS	POSIX, MPI
File System	Logical objects	Ext3, FAT32
Block storage	Block	iSCSI, SATA
Storage device	Technology	Controller

Block Storage

- Sequence of blocks which can be accessed (randomly)
  - Nowadays, block size is often 512 Byte
  - Only full blocks can be accessed
- Simple interface to access data
  - Read (offset, size)
  - Write (offset, size, <data>)
  - Additional management commands
  - Size must be multiple of a block
    - Modification might require Read-Modify-Write
- Examples: SATA, SCSI

File System Overview

- Provides an interface to logical (persistent) objects
  - Interface to access and manipulate objects
  - Semantics
- Store information about data (metadata)
  - Permissions, timestamps, ...
- Structure objects by providing a namespace
  - Usually the namespace is hierarchical
- On-disk-format defines physical representation
  - How are physical blocks mapped into logical objects

Low-level I/O interface

- Abstracts from file system
  - Common interface and semantics
  - Linux “Virtual Filesystem Switch” (VFS)
  - Windows “Installable File System”
- Operations:
  - Open(), Read(), Write(), Close()
- Drawbacks:
  - Individual file system features not usable
    - Low-level tools must be used
  - Common semantics does not exploit local features
    - Eventually fast operations must be done “slowly”

Storage Devices

Hard disk drive

Solid state drive

[Source: Wikipedia]

HDD vs. SSD

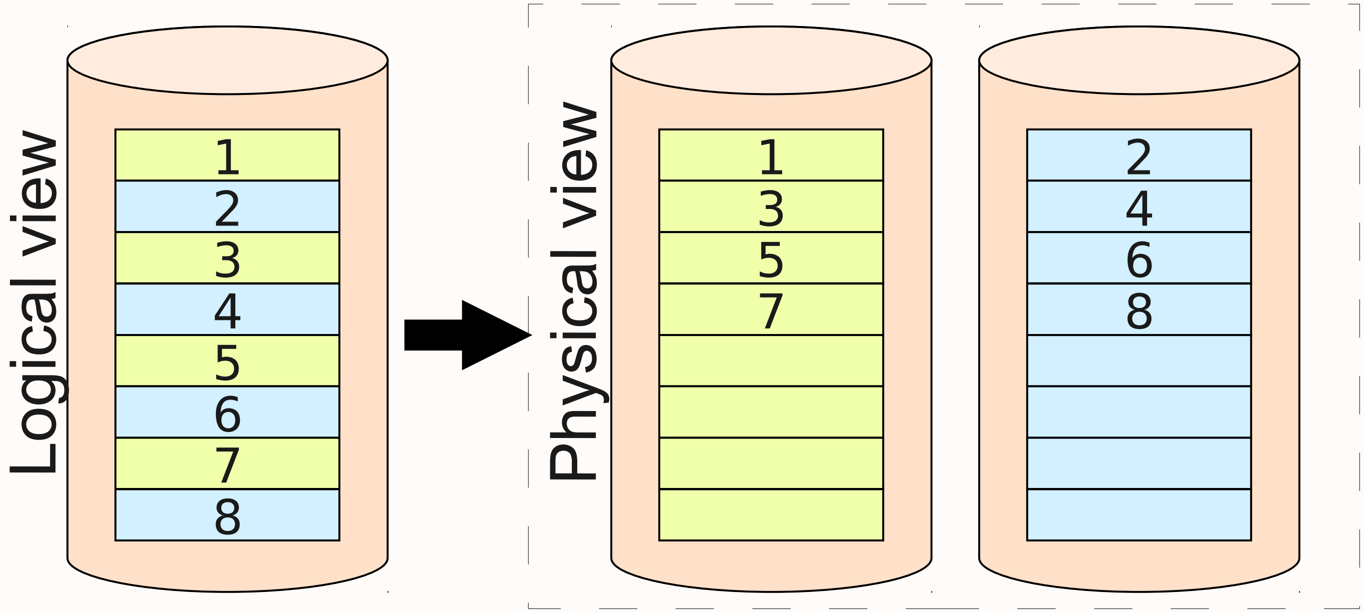
- SSDs have no mechanical parts
  - Technology: Flash-EEPROM
  - Time for random access == sequential access
  - Write-Access in block granularity (e.g. 256 KByte)
  - More expensive for capacity, but faster (200 MB/s)
  - Controller defines performance
- Requires new optimizations in OS and middleware
  - Sequential optimizations (might) harm performance

RAID

- Multiple storage devices are combined into
  - **R**edundant **A**rray of **I**ndependent **D**isks
- Hardware or Software distributes blocks of data
  - Among multiple devices
  - Transparent for the user or OS
- Depending on data distribution function increase
  - Capacity
  - Throughput
  - Availability and Durability

Data Distribution

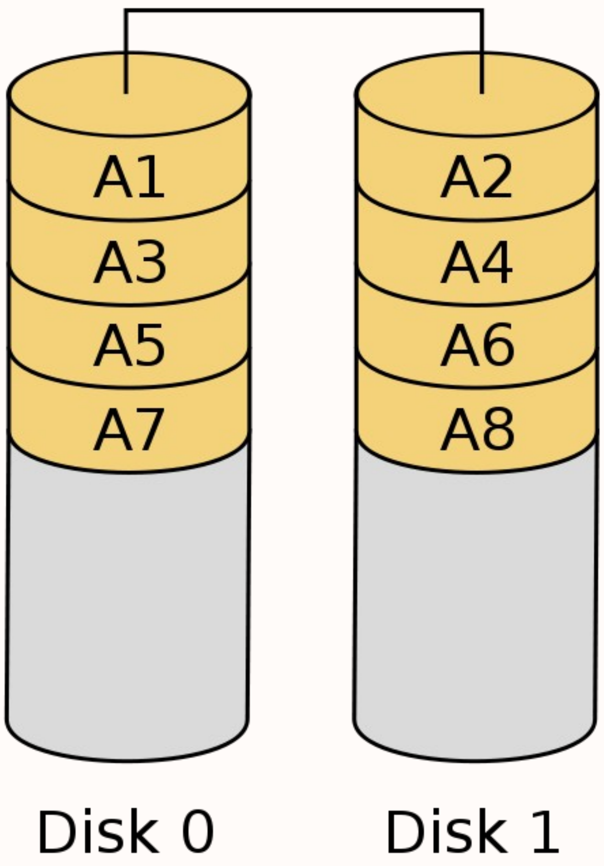
- Simple functions to allow hardware implementation
- Example:



- Block size can be configured towards application
  - Small block size increase parallelism of a single I/O access
  - Large block size increase concurrency of I/Os

RAID-0

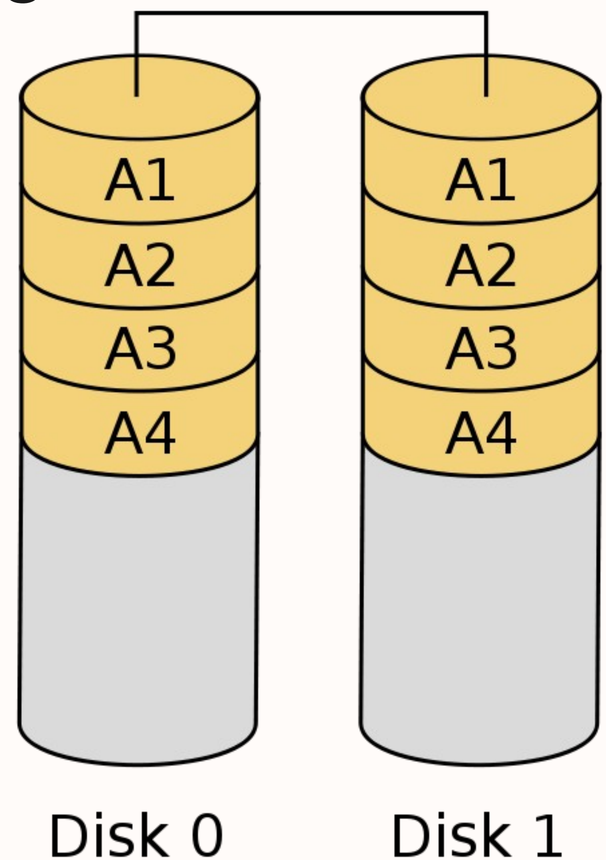
- Round-robin block distribution
- Improves performance
  - All devices participate in I/O
- Improves capacity
  - Sum of all capacities
- Error-prone
  - Not protected against device failure



[Source: Wikipedia]

RAID-1

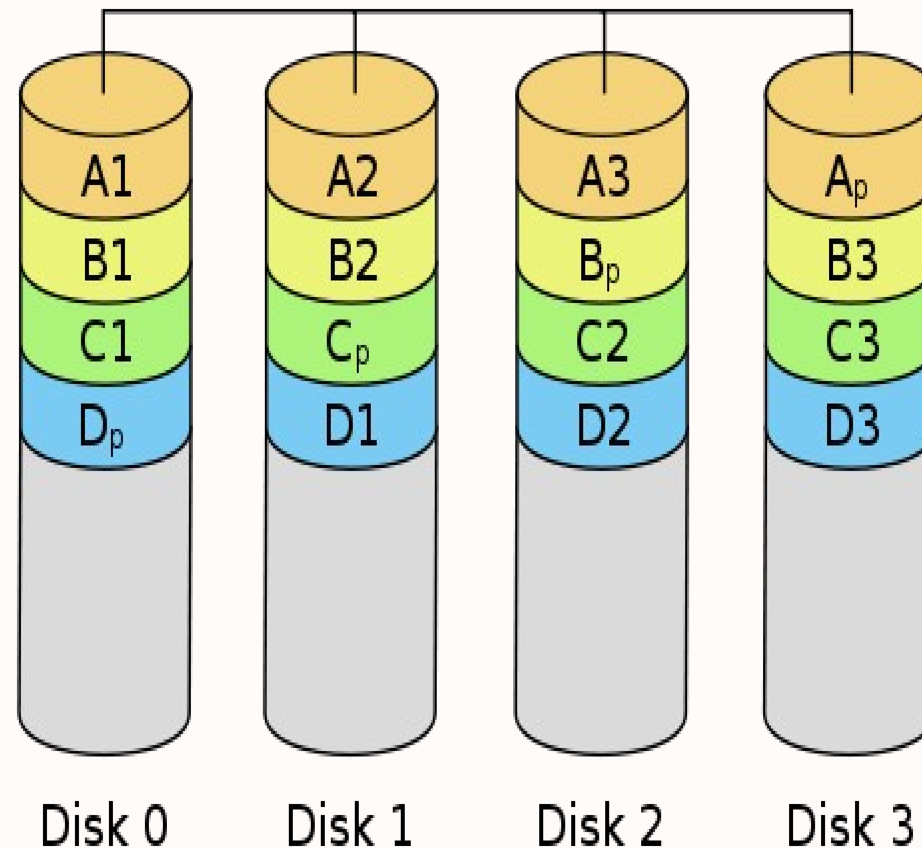
- Mirror data blocks between devices
  - Data is written to all devices
- Improves read performance
  - One device required to read data
- No capacity improvement
  - RAID == smallest device
- Protects against errors
  - Only one device required



[Source: Wikipedia]

RAID-5

- Data blocks are distributed in RR
- Additional parity block per row
  - One device may fail
  - XOR function
  - Requires to maintain parity block
  - Wanders between devices
- Compromise of performance, high availability and capacity



[Source: Wikipedia]

Tolerating Failures

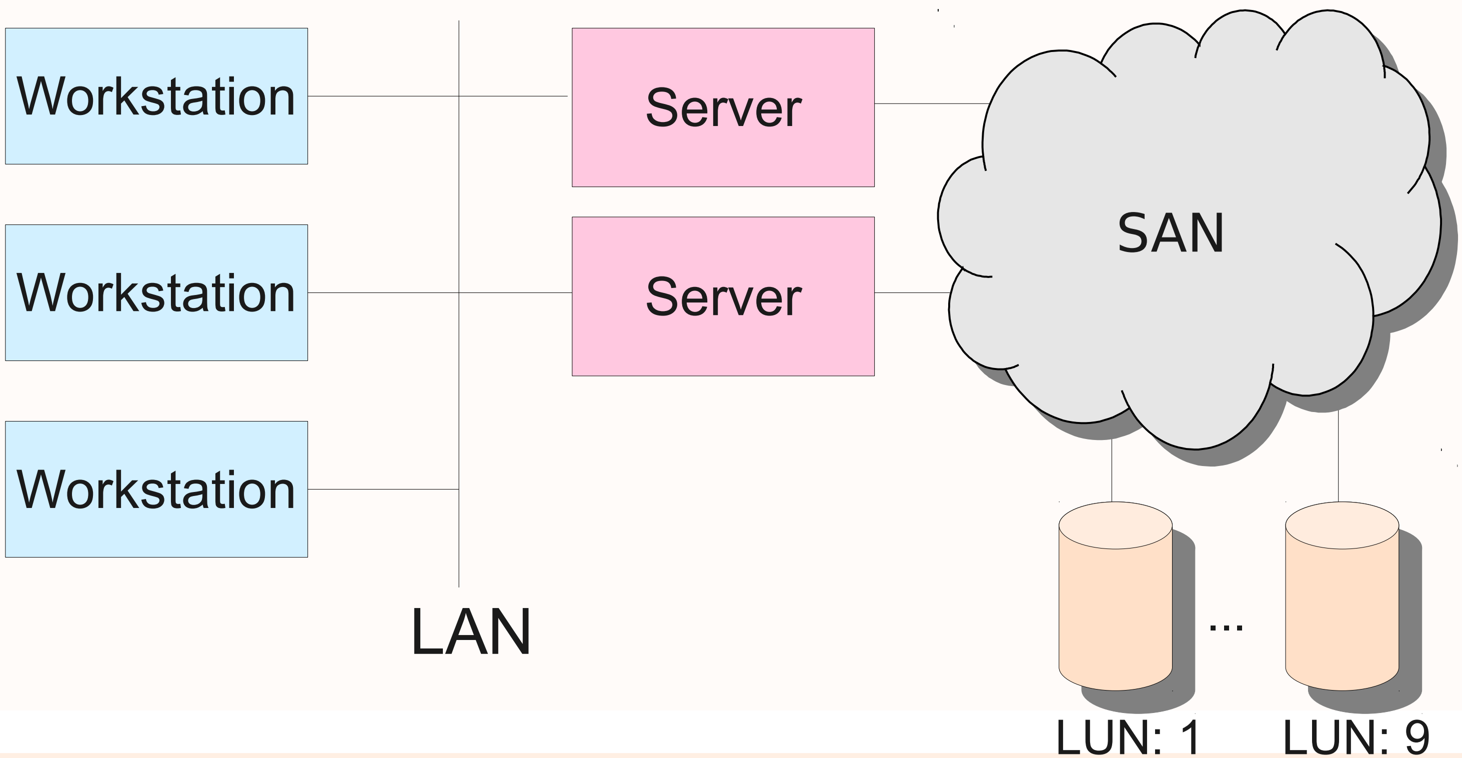
- Rebuild of degraded RAID(-5) systems
  - Requires to recreate blocks on broken device
    - Use XOR on all blocks in a row to compute missing data
  - Transparent to the user (but lower performance)
- Time intensive as device capacity improves
  - During rebuild procedure data protection is lower
  - A second disk failure in RAID-5 and data gets lost!

RAID Conclusions

- RAID distributes data blocks among devices
- Improve availability and persistence
- RAID-levels 2 to 4 are not important
- RAID-10 == RAID-0 over multiple RAID-1
- RAID-6 (2-dimensional parity)
  - Similar to RAID-5
  - Tolerates two hardware failures
  - State-of-the-art in industry
- RAID concept is used in higher abstraction levels

Storage Area Network

- SAN is a network with attached block storage
  - Multiple servers can access the same device
    - Only one server has dedicated access to one unit
    - But, devices can be partitioned into multiple logical units

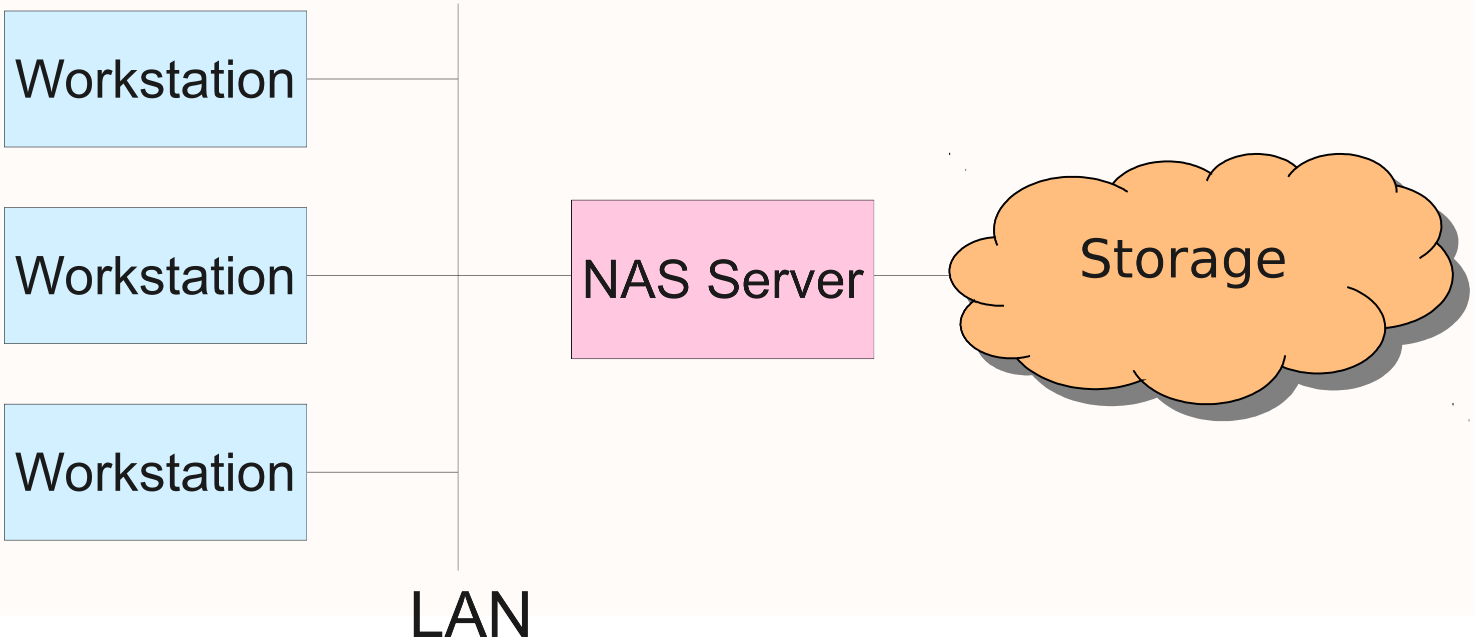


SAN

- Access via low level access in blocks
  - SCSI protocol
  - Servers must provide a file system for workstations
- Can be configured to provide high availability
  - Redundant servers, switches...
  - If one server/switch fails another takes over
- Technology
  - iSCSI (IP network, can use the LAN technology)
  - Fibre Channel

Network Attached Storage

- NAS provides network storage in a box
  - Buy a NAS, plug in, use it
  - High-level access: FTP, CIFS, NFS
  - Sometimes block storage via iSCSI



File Systems Aspects

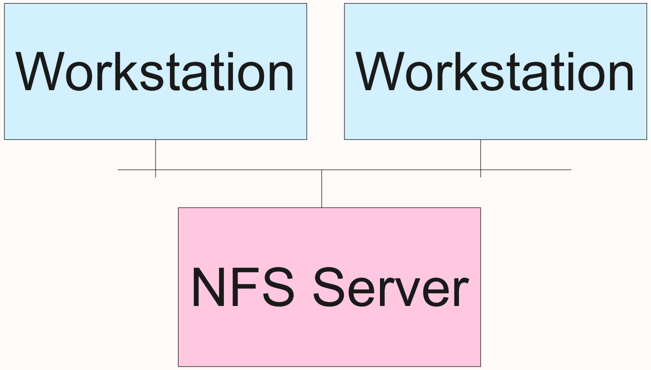
- Data integrity
- Fault-tolerance
- (High)-availability
- Access semantics
  - Concurrent access to file / directories
  - (Meta)Data caching
- Performance
  - Metadata ops per second and I/O-throughput
- Management features
  - Snapshots to ease backup
  - Performance monitoring

File Systems

- Disk
  - Manage/Access (local) direct attached storage
  - Examples: Ext3, FAT32, NTFS
- Shared disk / Cluster
  - Multiple nodes can access one block storage
  - SAN + software to allow concurrent access
  - Examples: GFS, OCFS2

Distributed (Network) File Systems

- Provide storage to many workstations (clients)
  - Like a NAS system
- Examples: NFS, CIFS
- NFS v4 (Network File System)
  - Client is connected to exactly one server
  - Operations are performed via Remote Procedure calls
  - (Meta)-data caching on the client
    - Close-to-open cache consistency
    - Clients can read stale (Meta)-data
    - File delegation
  - File or byte-range locks supported

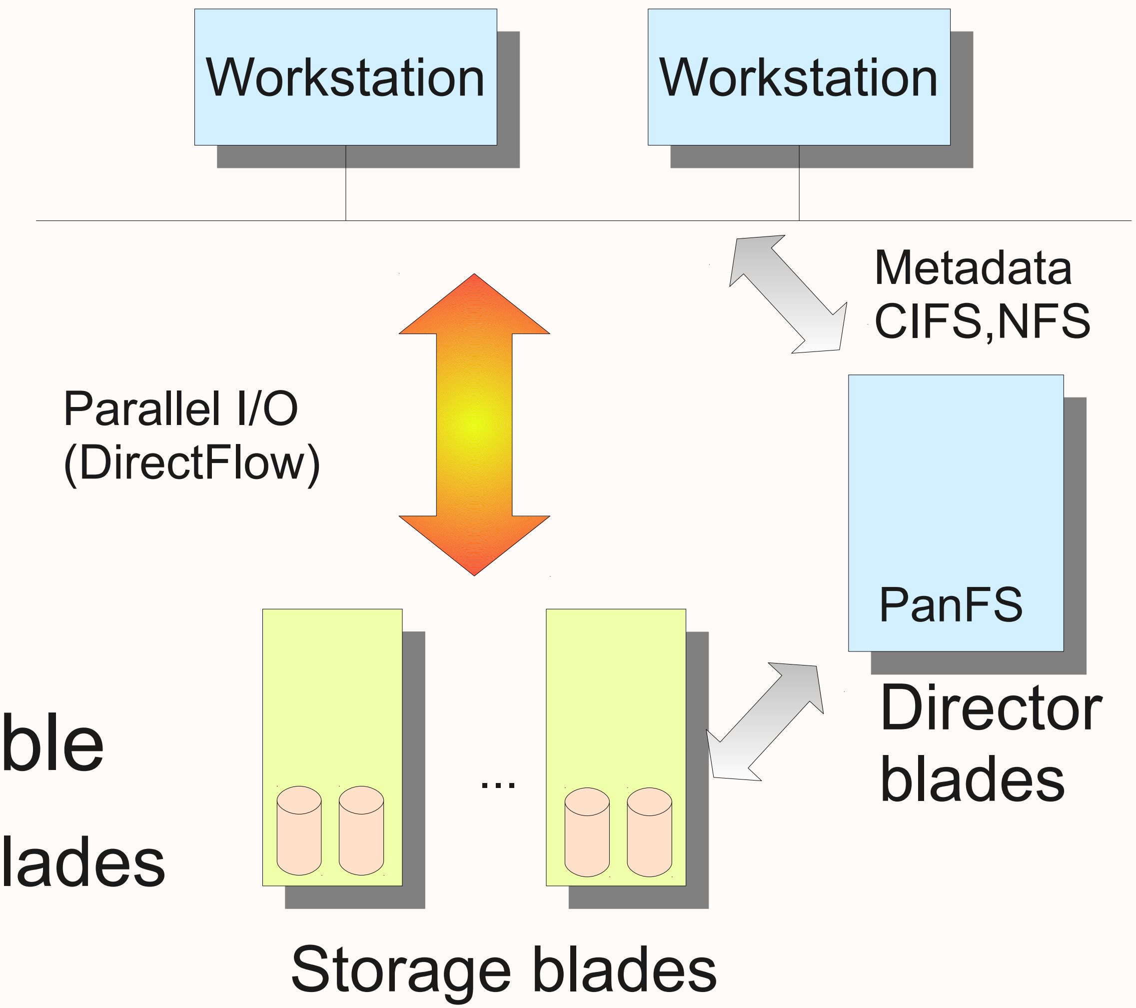


(Distributed) Parallel File Systems

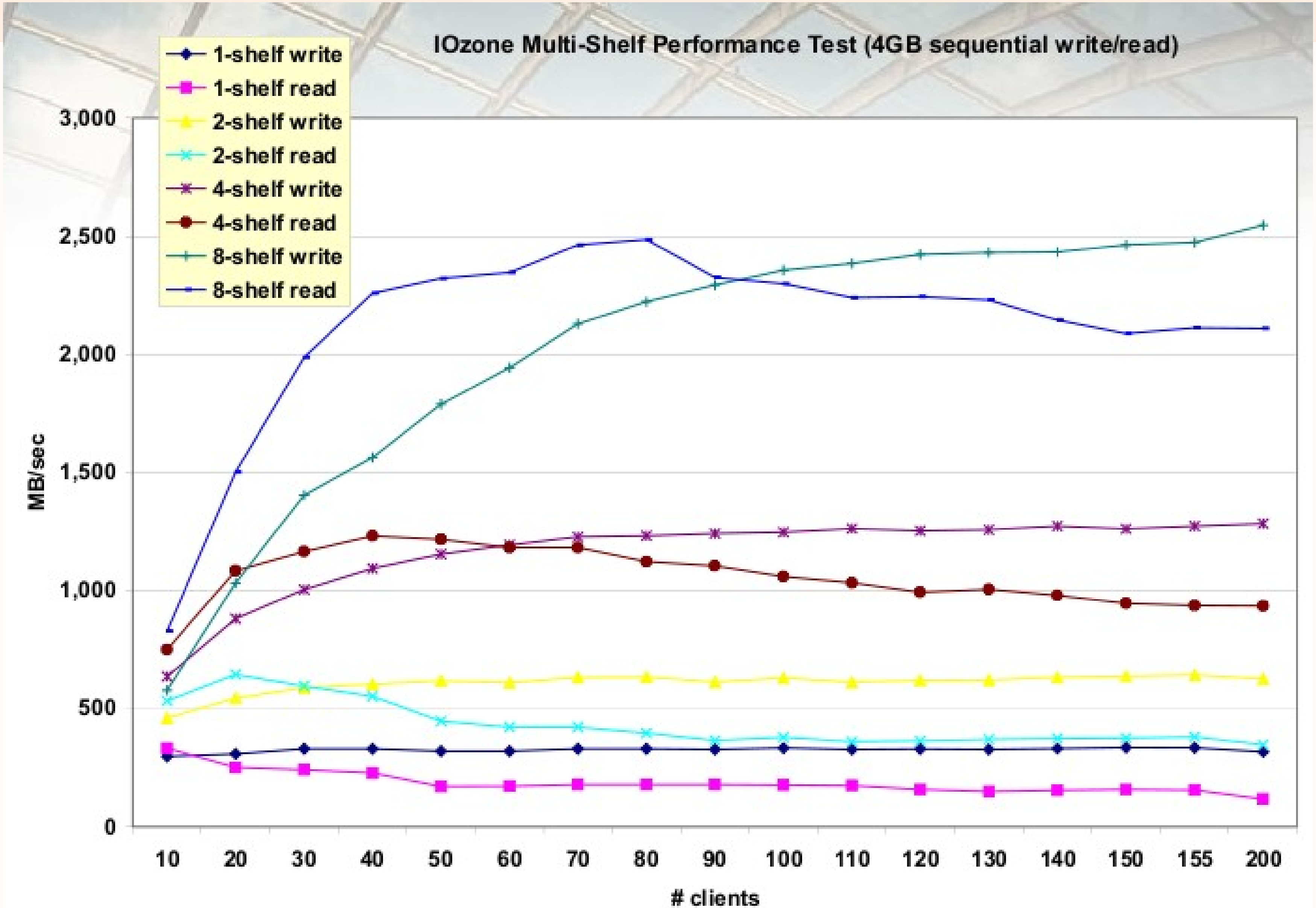
- A file can be accessed concurrently by all clients
- Data of a file is distributed among servers
  - Data is distributed by RAID concepts
  - Scalable performance
- Metadata may be distributed among servers
- Examples: GPFS, PanFS, PVFS2

Panasas Active Scale OS (PanFS)

- Metadata
  - Director blades
- Object storage
  - Storage blades
- Kernel module
  - Enables parallel I/O
- CIFS and NFS possible
  - Routed via director blades
- Dynamic RAID-level
  - Depending on file size

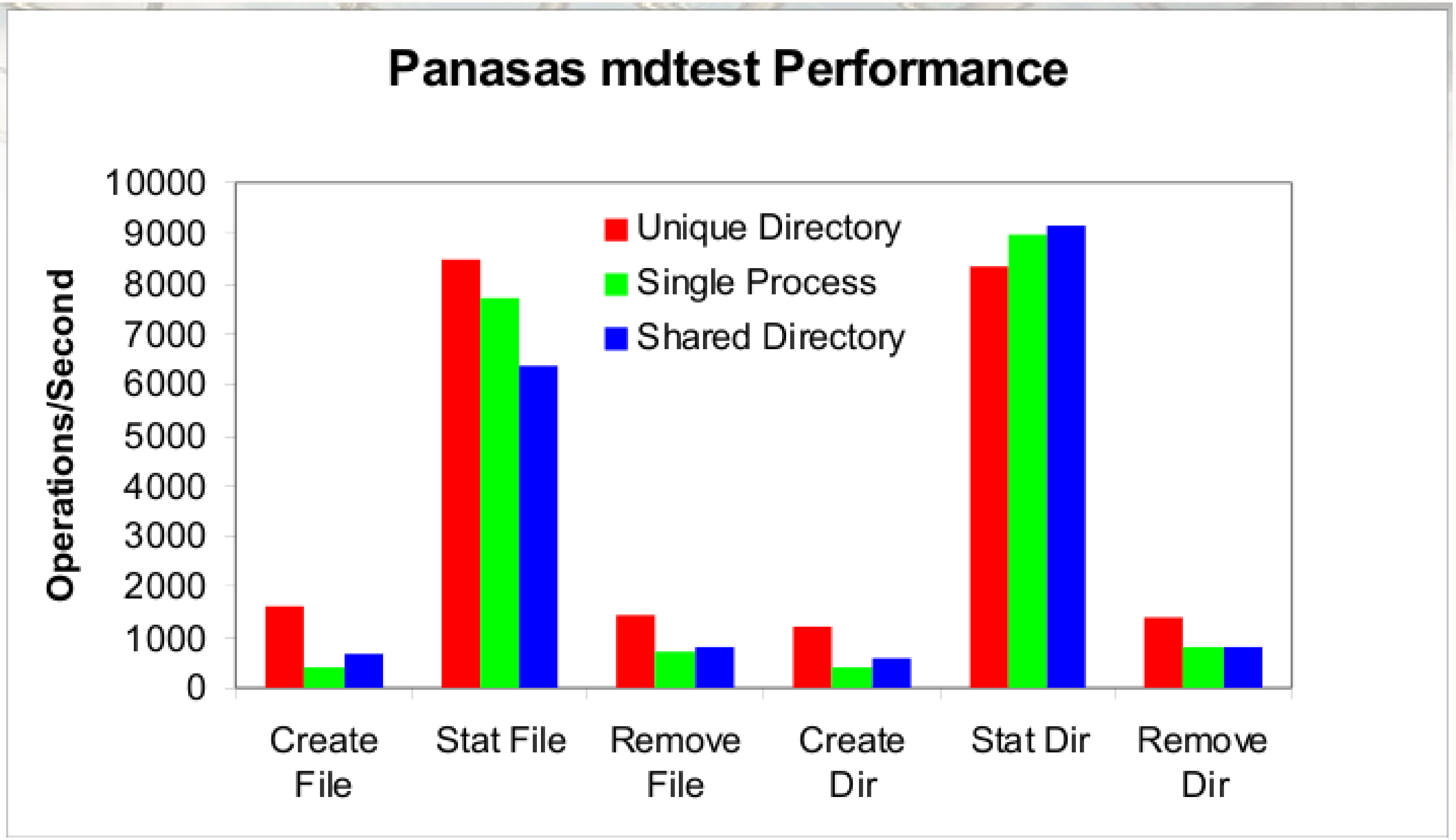


Panasas I/O Performance



[Taken from: Scalable Performance of the Panasas Parallel File System by Brent Welch]

Metadata Benchmark Example



mpirun -n 64 mdtest -d $dir -n 100 -i 3 -N 1 -v -u

[Taken from: Scalable Performance of the Panasas Parallel File System by Brent Welch]

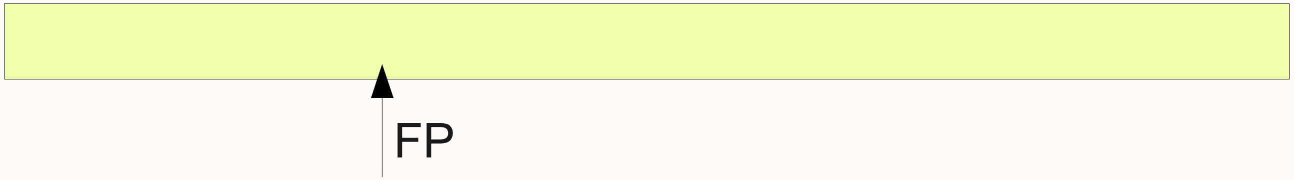
Programming (Parallel) I/O

Agenda

- Low-level I/O concept
- On-Disk-Format
- POSIX interface in brief
- MPI-I/O
- High-level I/O libraries

Low-level I/O Concept

- A file is a sequence of bytes



- File pointer shows last (access) position
- A number of bytes can be read/written
- File pointer can be repositioned

Semantic Gap in I/O

- Applications work with structured data
  - Vectors, Matrixes, 3-dimensional climate data
- A file is just a sequence of bytes

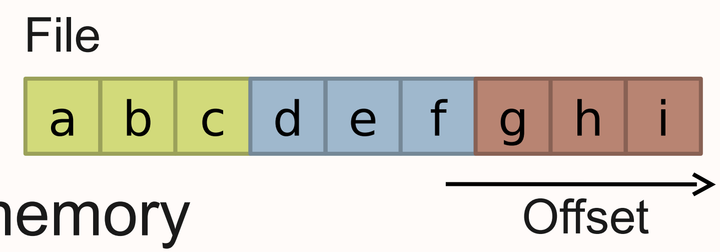
- Applications must serialize data into the file
  - Complex data types?
  - Mapping defines performance

On-Disk-Format

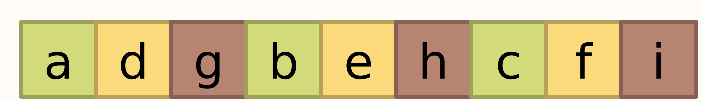
- Depends on application behavior
  - Sequential, large accesses favorable
- Example: Storing a 2-D matrix

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

- Variants:
  - Serialize by row
    - A row can be written directly from memory



  - Serialize by column



POSIX Interface Aspects

- POSIX requires serial processing of I/O operations
  - A read() following a write() must return new data
  - Cheap within one node
  - Expensive in a distributed environment
    - Cache-coherence between nodes
    - Lock mechanisms required
- Exception handling of calls expensive
- In Linux: large file support compile options

POSIX Interface Recommendations

- Use: open(), close(), pwrite(), pread() functions
  - pread() and pwrite() are thread safe
- Don't use asynchronous I/O (aio)
  - Performance varies between GLIBC implementation
- Don't use fwrite() etc.
  - Sometimes 64 Bit issues
  - Buffering unclear / might vary (even with setvbuf())
- Build a wrapper which ensures proper writing
  - Read man pages

MPI-I/O Concepts

- MPI-I/O: (coordinated) parallel I/O for MPI
  - Similar to POSIX-I/O
- File pointer:
  - Individual vs. Shared File Pointer
- File view
  - Filters uninteresting file areas for the processes
  - Allows noncontiguous accesses
- Collective I/O
  - Multiple clients participate in a collaborative I/O
  - Enables collective optimizations
- Non-blocking I/O
  - Overlap computation with I/O

Accessing a File with MPI-I/O

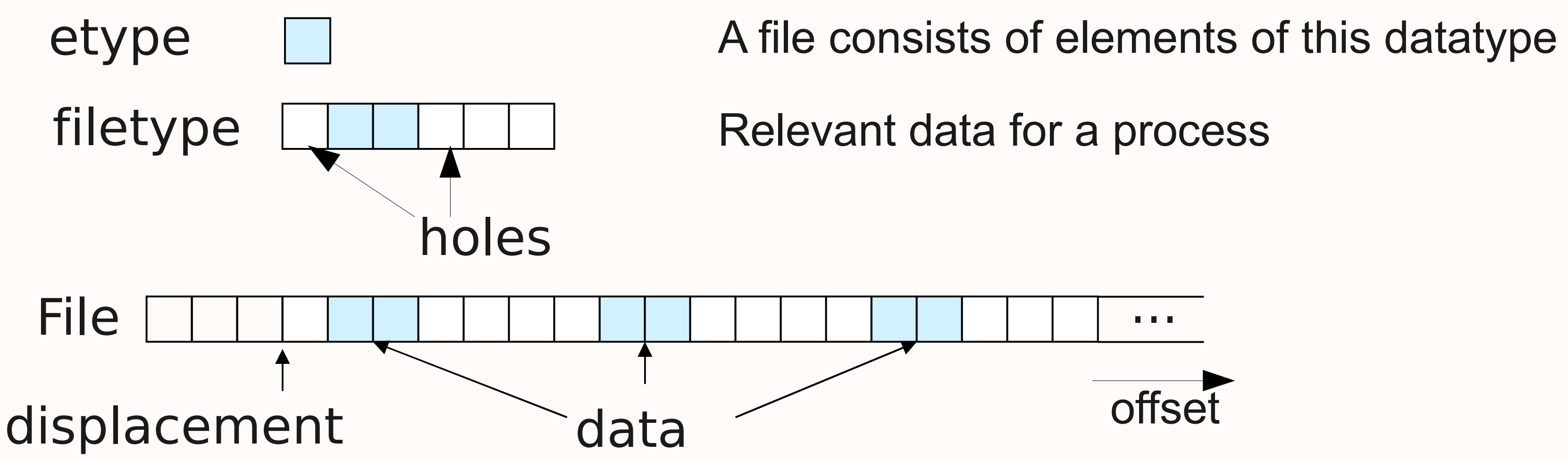
```
MPI_Offset my_offset, my_current_offset;

MPI_File fh;

MPI_Status status;

int buffer[100];

MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_RDWR
    | MPI_MODE_CREATE , MPI_INFO_NULL, & fh);

MPI_File_write(fh, buffer, 100, MPI_INT, &status);

MPI_Get_count(&status, MPI_INT, &count);

printf("process %3d wrote %d ints\n", my_rank, count);

MPI_File_close(&fh);
```

Shared File Pointer

- Processes share only one file pointer
- Read/Write from global file offset
- Useful for log-files, appends to the file
- Drawbacks:
  - Serialization of operation => Bottleneck
  - Not always implemented
- Advice: Avoid shared file pointers

File Views

- Each process can set an individual file view
  - MPI_File_set_view(MPI_File fh, MPI_Offset displacement, MPI_Datatype etype, MPI_Datatype filetype, char *datarep, MPI_Info info)
  - MPI_File_set_view() is a collective operation!

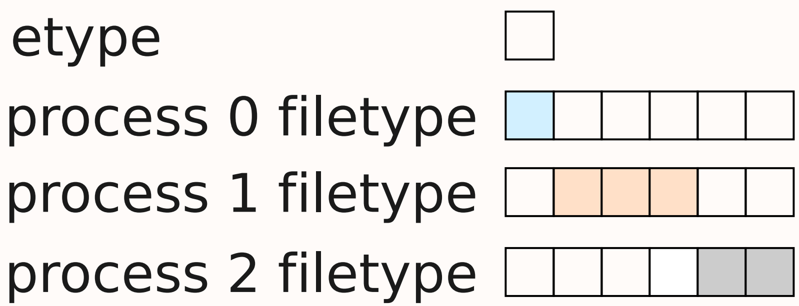


File Views (2)

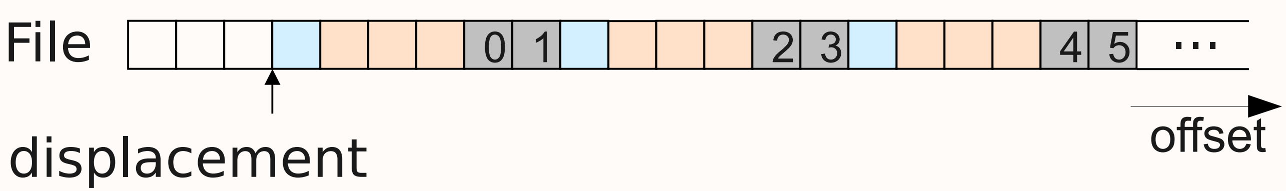
- I/O operations return only accessible data
- Positioning happens in multiple of etype
  - Example: MPI_File_seek(fh, 10, MPI_SEEK_SET)
    - File pointer is set to the 10 element which can be accessed
- Displacement is used to skip the file header
- For portability a data representation can be set
  - “native” : data is stored in the file as it is in memory
    - Not portable
  - “internal” : MPI implementation chooses format
    - Portable as long as the same MPI implementation is used
  - “external32” : I/O is converted to a defined format

File Views Example

- Example: assume three processes
  - File data is distributed among processes

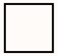


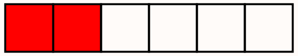
  - Now they use MPI_File_set_view(filetype...)
  - File data is distributed in the following way:



Offsets are given for process 2 in the boxes

File View Code Example

etype = MPI_INT 

filetype = 

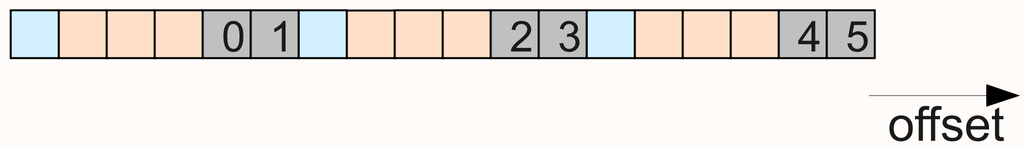
```
/* Create a datatype consisting of two integers */
MPI_Type_contiguous(2,MPI_INT,&contig);

/* add four holes */
lower_boundary=0;
extent=6*sizeof(int); /* extend size is a total of 6 int */
MPI_Type_create_resized(contig,lower_boundary,extent,
  &filetype);

/* finalize the new datatype */
MPI_Type_commit(&filetyp);

/* set the file view according to the new datatype */
MPI_File_set_view(filehandle,displacement,etype,filetype,
  “native“,MPI_INFO_NULL);
```

Collective I/O

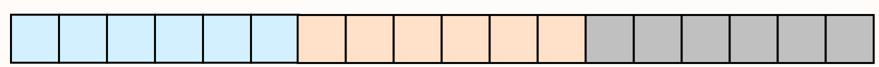
- MPI_File_[read|write]_all()
  - All clients of a communicator perform I/O
- MPI_File_open() sets participating communicator
- Usually file is partitioned by setting a file view
- Clients try to aggregate small accesses
  - Large requests to I/O subsystem
  - Clients exchange required data by communication
  - *Additional communication might slow down I/O!*

Collective I/O Example

- Assume three processes access their data

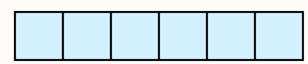


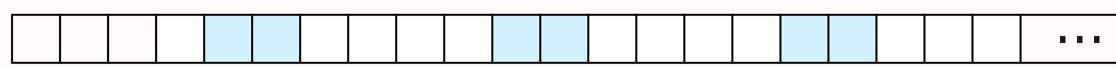
- Processes read individual potions of the file:



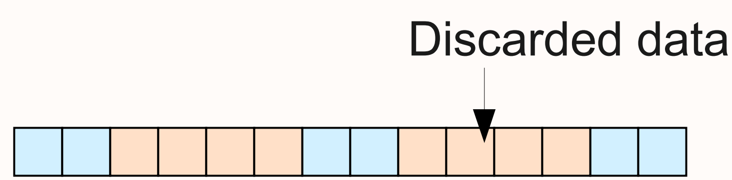
- Then communicate data to target processes
  - Process continues until all data is processed
    - I/O granularity depends on the size of the collective buffer
    - At most each client fills its collective buffer (e.g. 8 MByte)

Data-Sieving

- Applicable to non-contigous access
- Read/Write a contiguous block of data
  - Throw away uninteresting data
  - Reduces number of I/O calls
- Application requests data: 
- File view:

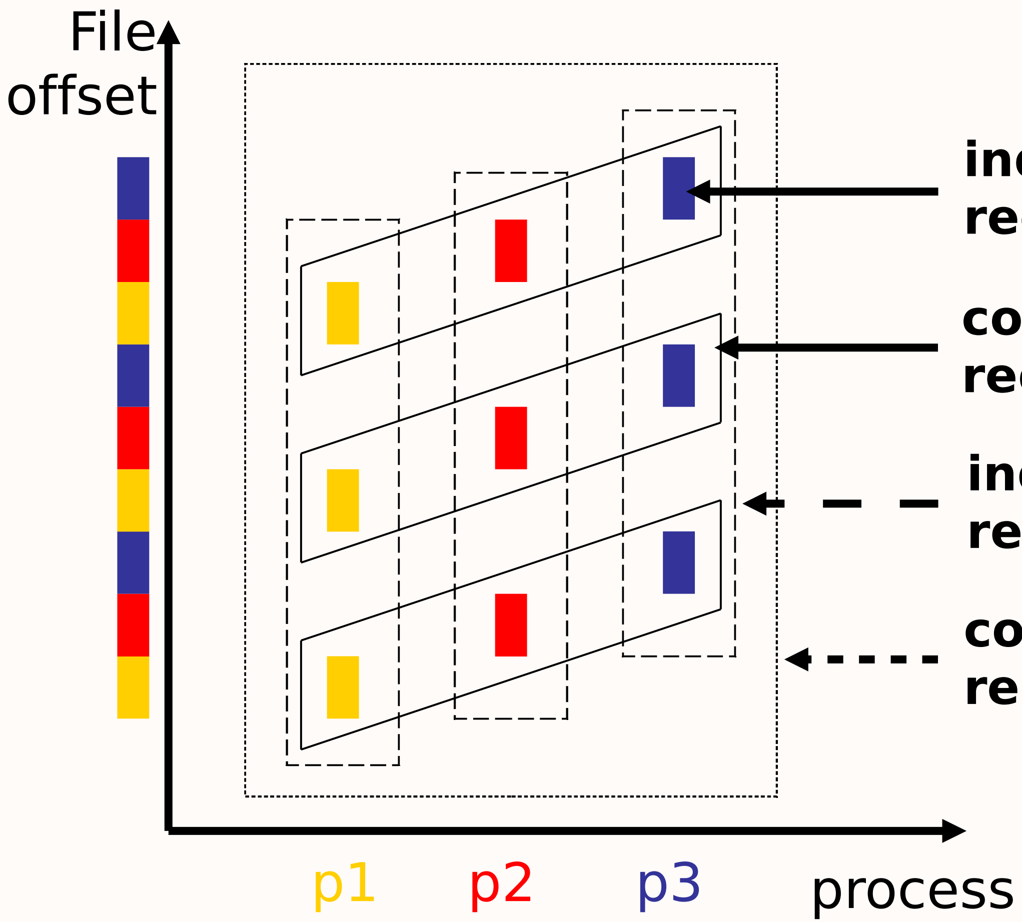


- Accessed file data:



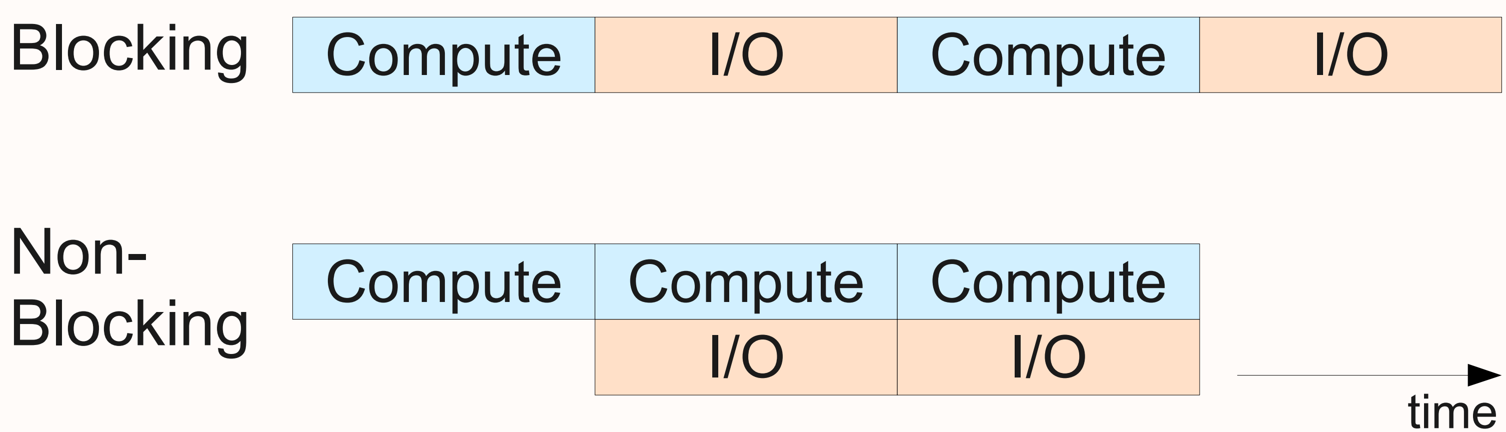
Level of Access

- Non-contiguous and collective can be combined



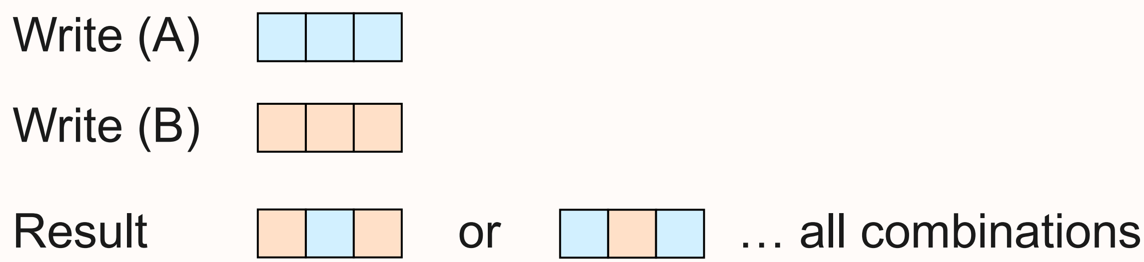
Non-blocking I/O

- All blocking I/O operations have a non-blocking variant
  - Name convention MPI_File_I... [read|write|...]
- Allows to overlap computation & I/O
- In the best case I/O access time is hidden

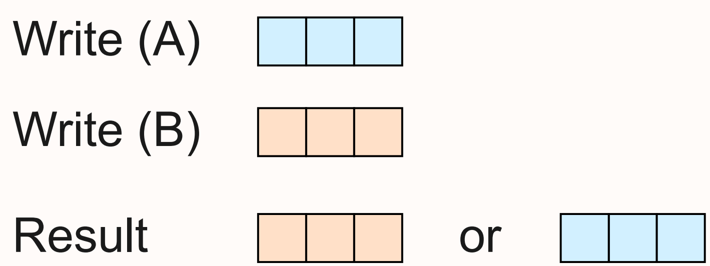


MPI Consistency Semantics

- Concurrent modifications of the same block may destroy data:



- Atomic mode (Sequentially consistency)
  - MPI_File_set_atomicity(MPI_File fh, int flag)



- Flushing file content (e.g. for checkpoints)
  - MPI_File_sync(MPI_File fh)

High-Level I/O Libraries

- A file is more than just an array of bytes!
  - Abstract description of datatypes and I/O
  - Self describing data formats
  - Architecture independent (portability)
- Examples: NetCDF-4 and HDF5
- Recent development: ADIOS
  - API explicitly expresses non-blocking I/O
  - Adjust I/O method by changing XML
    - MPI (collective, individual), asynchronous I/O, NULL
    - Without recompiling!
  - BP Dataformat relaxes HDF5 dataformat for performance

What is HDF5

- *A versatile **data model** that can represent very complex data objects and a wide variety of metadata.*
- *A completely **portable file format** with no limit on the number or size of data objects in the collection.*
- *A **software library** that runs on a range of computational platforms, from laptops to massively parallel systems, and implements a high-level API with C, C++, Fortran 90, and Java interfaces.*
- *A rich set of **integrated performance features** that allow for access time and storage space optimizations.*
- ***Tools and applications** for managing, manipulating, viewing, and analyzing the data in the collection.*

Citation from HDFgroup.org

HDF5

- More sophisticated than NetCDF
  - Relaxes shortcomings
  - Datatype concept similar to MPI, but references to data
- Uses its own portable file format
  - Basically its a file system inside a file
  - *HDF5 group*: directory with metadata contains datasets and other groups
  - *HDF5 dataset*: a multidimensional array of data elements with additional metadata
  - NetCDF-4 allows to store data in the HDF5 format!
- Parallel processing is possible, it can use MPI-IO

Other Features

- Filters can be added per dataset
  - Compression, integrity protection via MD5, …
- File formats allows different file layouts
  - contiguous, compact, chunked
  - Depending on the workload the right one can be chosen
- References between datasets
- Attributes describe content of a dataset
- Tools:
  - h5perf tool measures performance of a N-D matrix
  - h5dump prints metadata and data of a file
  - h5repack copy & modify internal storage format

HDF5 – Creating a Dataset

```
hid_t    dataset, datatype, dataspace;   /* declare identifiers */
/*  Create dataspace: Describe the size of the array and  create the data space for
fixed size dataset.  */
dimsf[0] = NX;  dimsf[1] = NY;
dataspace = H5Screate_simple(RANK, dimsf, NULL);

/*  Define datatype for the data in the file. Store little endian integer numbers. */
datatype = H5Tcopy(H5T_NATIVE_INT);
status = H5Tset_order(datatype, H5T_ORDER_LE);

/*  Create a new dataset within the file using defined dataspace and datatype and
default dataset creation properties. NOTE: H5T_NATIVE_INT can be used as
datatype if conversion to little endian is not needed. */
dataset = H5Dcreate(file, DATASETNAME, datatype, dataspace, H5P_DEFAULT);
```

[Example taken from http://www.hdfgroup.org/]

h5dump example

```
HDF5 "h5ex_t_cpxcmpdatt.h5" {
GROUP "/" {
  DATASET "Ambient_Temperature" {
    DATATYPE  H5T_IEEE_F64LE
    DATASPACE  SIMPLE { ( 32, 32 ) / ( 32, 32 ) }
    DATA {
    (0,0): 66.8, 66.9, 67, 67.1, 67.2, 67.3, 67.4, 67.5, 67.6, 67.7, 67.8,
    (0,22): 69, 69.1, 69.2, 69.3, 69.4, 69.5, 69.6, 69.7, 69.8, 69.9,
     ...
    (31,22): 72.1, 72.2, 72.3, 72.4, 72.5, 72.6, 72.7, 72.8, 72.9, 73
  }}

  GROUP "Land_Vehicles" {}

  DATASET "DS1" {
    ATTRIBUTE "A1" {
      DATATYPE  H5T_COMPOUND {
         H5T_VLEN { H5T_COMPOUND {
         H5T_STD_I32LE "Serial number";
         H5T_STRING {
           STRSIZE H5T_VARIABLE;STRPAD H5T_STR_NULLTERM;CSET
H5T_CSET_ASCII;CTYPE H5T_C_S1;
         } "Location";
         H5T_IEEE_F64LE "Temperature (F)";
         H5T_IEEE_F64LE "Pressure (inHg)";
        }} "Sensors";
         H5T_STRING {
           STRSIZE H5T_VARIABLE...
         } "Name";
         H5T_ENUM {
           H5T_STD_I32LE;  "Red" 0; "Green"  1; "Blue" 2;
         } "Color";
         H5T_ARRAY { [3] H5T_IEEE_F64LE } "Location";
         H5T_REFERENCE "Group";
         H5T_REFERENCE "Surveyed areas";
      }
      DATASPACE  SIMPLE { ( 2 ) / ( 2 ) }
      DATA {
      ...
      (1): { ({
               3244,
               "Roof",
               83.82,
               29.92
            }),
         "Automobile",
         Red,
         [ 326734, 221568, 432.36 ],
         GROUP 1400 /Land_Vehicles ,
         DATASET /Ambient_Temperature {(8,26)-(11,28)}
       } } } }}}
```

[Example taken from http://www.hdfgroup.org/]

I/O Programming Recommendation

- Design a On-Disk-Format
  - Think about later access patterns / use cases
    - Must allow large blocks to be accessed
  - Add a self describing header (version etc.)
- Develop a domain specific wrapper library
  - Abstract from application tasks e.g. start_checkpoint()
  - Don't use any I/O library directly in your code!
    - Maybe a more suitable library appears
  - Use HDF5 and MPI-I/O if appropriate
  - Add optimizations like write-behind / read-ahead later

Outlook

- SSDs will be deployed
  - Reduces the penalty of random I/O
  - Writing sequential blocks of data out remains important
- HDF5 will improve (grants from US government)
- No changes to I/O in MPI-3

- Long term perspective:
  - MPI will be aware of system topology
  - (Better) Automatic tuning of MPI towards application

Excercise

- Parallelization of a Matrix Vector multiplication
  - Each process multiplies a part of the matrix
  - Matrix/Result is read/written in MPI by two different ways:
    - The first process reads/writes the whole Matrix and distributes/gathers data
    - Each process uses a file view to read/write portions of its data
      - The user can choose to use individual or collective calls
  - The data format should be variable between:
    - Your own simple file format
    - HDF5
    - Therefore, use functions to abstract from the real input/output
  - Identify performance factors and bottlenecks for your cluster
    - How could you estimate the time for I/O and computation?
- You can program a Matrix Matrix multiplication instead ;-)

Literature

- MPI:
  http://www.mcs.anl.gov/research/projects/mpi/mpi-s
- HDF5: http://www.hdfgroup.org/
- ADIOS: http://adiosapi.org/
- Thomas Ludwig, Lecture HEAS0809 and HR10
- Brent Welch, NSC08, online presentation*:*
  Scalable Performance of the Panasas Parallel File System