Introduction

Parallel Storage Systems

2024-04-08

Prof. Dr. Michael Kuhn

michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de

Outline

Survey

- Have you heard our lecture on parallel programming?
  1. Yes
  2. No

Survey

- How familiar are you with C?
  1. Expert
  2. Advanced
  3. Beginner
  4. Not at all

Survey

- How familiar are you with Linux?
    1. Expert
    2. Advanced
    3. Beginner
    4. Not at all

Survey

- How familiar are you with Git?
  1. Expert
  2. Advanced
  3. Beginner
  4. Not at all

Lecture and Exercises

- Lecture: Mondays, 17:15–18:45
  - Lecture will be held in-person and recorded for later viewing
  - We will also use this time slot to clear up questions etc.
- Exercises: Fridays, 15:15–16:45
  - We will discuss solutions and take a look at the next exercise sheet
  - Attendance is mandatory and everyone has to present at least once
  - You need at least 50 % of the overall points to pass the exercises
- Exam: Oral

Communication

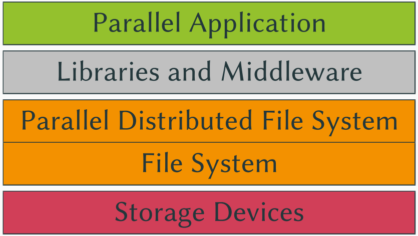
- Please sign up for the Mattermost team
  - If there are questions about the lecture or exercises, please ask them there
  - Feel free to use it for discussion and communication with your fellow students
    - You can also use it to find people for your exercise group
  - You can of course also send us e-mails:
    - `michael.kuhn@ovgu.de` (lecture and general)
    - `michael.blesel@ovgu.de` (exercises)
- Slides, exercise sheets etc. will be available on the website

Literature

- High Performance Parallel I/O (Prabhat, Quincey Koziol); October 23, 2014 by Chapman and Hall/CRC; ISBN 9781466582347
- Understanding the Linux Kernel (Daniel P. Bovet, Marco Cesati)
- Professional Linux Kernel Architecture (Wolfgang Maurer)

Outline

- Storage stack is layered
  - Many different components are involved
  - Performance problems influence all layers
- Complex interactions
  - Optimizations and workarounds on all layers
  - Information about other layers required
- Data transformation
  - Data is transported through all layers
  - Loss of high-level information



Topics

- Introduction (8 April – today ☺)
  - This is an overview of the most important parallel concepts
- Storage Devices (15 April)
  - Performance characteristics, storage arrays, reliability etc.
- File Systems (22 April)
  - General file system concepts and data structures
- Modern File Systems (29 April)
  - More advanced functionality such as copy-on-write, checksums etc.

Topics...

- Parallel Distributed File Systems (6 May)
  - Parallel and distributed concepts, performance considerations
- MPI-IO (27 May)
  - Concepts for parallel I/O, interface and functionality
- Libraries (3 June)
  - Overview of different I/O libraries
- Optimizations (10 June)
  - Basics of performance optimization, different approaches

Topics...

- Performance Analysis (17 June)
  - How to measure and assess I/O performance
- Data Reduction (24 June)
  - Overview of data reduction considerations and techniques
- Future Developments (1 July)
  - Upcoming storage hardware and software approaches
- Research Talks (8 July)
  - Research topics currently investigated in our group

Outline

Overview

- Exercises will require some programming in (preferably) a systems language
  - Trying out the concepts taught in the lecture
- You should have experience in a programming language
  - Experience in C is not necessary (but helps)
- We will mostly work on our cluster via SSH
  - Some exercises can also be done on your own computer
  - Logging in and setting everything up will be part of the first exercise

Topics

- Introduction (8 April)
  - Set up development environment and C introduction
- Debugging and Checkpoints (24 April)
  - Debugging C code and reading/writing data
- I/O Tools (8 May)
  - Using tools to analyze and optimize I/O performance

Topics...

- Dummy File System (22 May)
  - Introduction to file system interface using FUSE
- Memory File System (5 June)
  - Extend dummy file system to store data in memory
- Persistent File System (26 June)
  - Develop a design for a persistent file system
  - Extending memory file system to persist data

Outline

Motivation

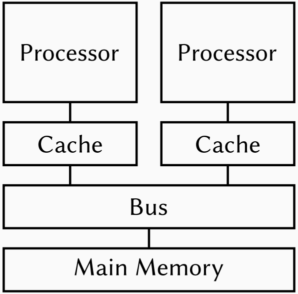
- Parallel programming is an important skill
  - Processors feature an increasing amount of cores
  - Even current phones have eight cores
- Serial applications will not be able to fully utilize a machine
  - Except for cases we call trivial parallelization
  - Sometimes possible to run multiple serial applications in parallel
- Parallelization is very important in science
  - Many problems can only be solved on supercomputers
  - High-performance computing (HPC)

Hardware Architectures

- Until ca. 2005: Performance increase via clock rate
  - Going from n GHz to 2n GHz will usually double application performance
- Since ca. 2005: Performance increase via core count
  - Clock rate cannot be increased further
  - Power consumption/heat depends on clock rate
  - Biggest supercomputers on TOP500 list have more than 10,000,000 cores
- Important classification: Memory access model
  - Shared and distributed memory
  - In reality, typically hybrid systems

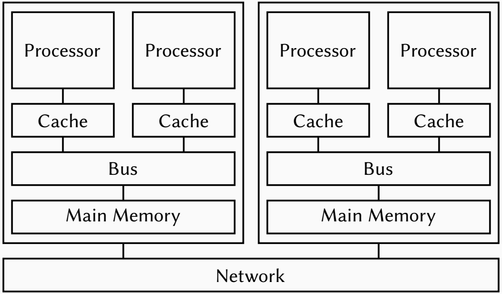
Hardware Architectures...

- All processors have access to shared memory
  - There might be speed differences due to NUMA
- Typically refers to single machines
  - Shared memory can also be virtual
- Processors consist of multiple cores
  - Each core has its own caches
  - Shared cache for the whole processor
- Access to shared memory via a bus
  - This also limits scalability of shared memory



Hardware Architectures...

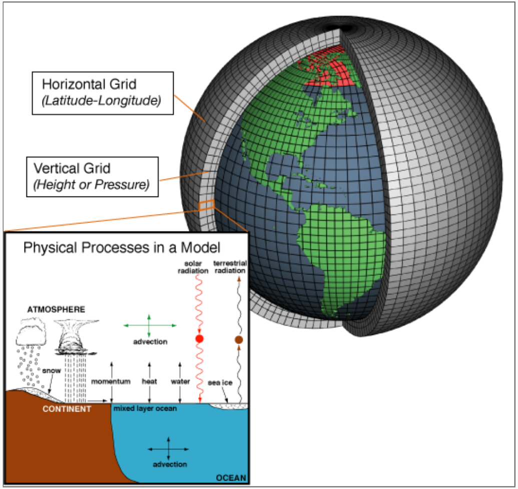
- Processors only have access to own memory
  - Typically with shared memory architecture
- Typically refers to a cluster of machines
  - Could theoretically be used inside machine
- Machines are connected via a network
  - Determines scalability and performance
  - Different network technologies and topologies



Parallel Programming

- Parallel programming is used to increase application performance
    - In HPC, OpenMP and MPI are often used together
- OpenMP is an interface for shared memory
    - Applications run as multiple threads within a single process
    - OpenMP features thread management, task scheduling, synchronization and more
- MPI (Message Passing Interface) is an interface for distributed memory
    - Applications run distributed over multiple compute nodes
    - MPI features message passing, input/output and other functions
- Both approaches are available for multiple programming languages

- Numerical problems are mostly iterative
  - Simulations often performed in time steps
- Global conditions for termination
  - Run for a specified number of time steps
- Data structures are often regular
  - Data often stored in one or more matrices
- Many phenomena are highly parallel
  - Galaxies, planets, climate and weather
- Parallel computing is well-suited
  - Data and components can be distributed



[NOAA, 2007]

Process X		
Code, Memory, Files		
Thread 0 Memory	*Thread 1* Memory	*Thread 2* Memory
⋮	⋮	⋮

- We will only take a look at threads for now
  - Message passing will be covered later
- Processes are instances of an application
  - Applications can be started multiple times
  - Processes are isolated from each other by the operating system
  - Resources like allocated memory, opened files etc. are managed per-process
- Threads are lightweight processes
  - Threads have their own stacks but share all other resources
  - Shared access to resources has to be synchronized
  - Uncoordinated access can lead to errors very easily

- Threads share a common address space
  - Communication is often done via shared variables
  - Threads are processed independently, that is, in parallel
  - If one thread crashes, the process crashes with all threads
- Processes have their own address spaces
  - Typically have to start multiple processes for distributed memory
  - Overhead is normally higher than with shared memory
  - There are also concepts for distributed shared memory
- In practice, hybrid approaches are used
  - A few processes per node (e. g., one per socket)
  - Many threads per process (e. g., one per core)

- Numerical applications often deal with matrices
  - Matrices are as big as the main memory allows
  - We want to calculate the sum of all elements
- Have to go through all rows and columns
  - Process one element after the other

(0,0)	(0,1)	...	(0,n-1)	(0,n)
...	...	...	...	...
(m,0)	(m,1)	...	(m,n-1)	(m,n)

```
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        sum += arr[i][j];
    }
}
```

Parallelization with OpenMP…

- OpenMP allows parallelization using compiler pragmas
  - Very convenient for developers, no internal knowledge necessary
  - Reduced functionality when compared to system-level approaches

```c
#pragma omp parallel for
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        sum += arr[i][j];
    }
}
```

```c
for (int i = 0; i < m/2; i++) {
    for (int j = 0; j < n; j++) {
        sum += arr[i][j];
    }
}
```

```c
for (int i = m/2; i < m; i++) {
    for (int j = 0; j < n; j++) {
        sum += arr[i][j];
    }
}
```

- First for loop is split up across multiple threads
  - Usually as many threads as there are cores
  - OpenMP can also do dynamic distributions and further scheduling
- Example: Laptop with two cores
  - First core calculates `0` to `(m/2)-1`
  - Second core calculates `m/2` to `m-1`

- This solution was very easy but also wrong ☺
  - Instead of the correct sum, we get weird values
  - Every time we run the application, the result changes

```
$ ./openmp
sum=3773725
$ ./openmp
sum=4012997
$ ./openmp
sum=12325088
$ ./openmp
sum=2456866
$ ./openmp
sum=11970989
$ ./openmp
sum=2818054
$ ./openmp
sum=3979092
```

Parallelization with OpenMP...

- This solution was very easy but also wrong ☹
    - Instead of the correct sum, we get weird values
    - Every time we run the application, the result changes
- Shared memory makes it easy to access the sum variable
    - Access has to be synchronized, otherwise errors occur
    - We have produced a so-called race condition
- There are several possibilities to solve the problem
    - Add a lock around the operation (slow)
    - Use atomic instructions (fast)

```
$ ./openmp
sum=3773725
$ ./openmp
sum=4012997
$ ./openmp
sum=12325088
$ ./openmp
sum=2456866
$ ./openmp
sum=11970989
$ ./openmp
sum=2818054
$ ./openmp
sum=3979092
```

Debugging

- Parallel programming has at least two new error classes
  1. Deadlocks
  2. Race conditions
- A race condition has resulted in a wrong result in our example
  - Incrementing a variable consists of three operations
    1. Loading the variable
    2. Modifying the variable
    3. Storing the variable
  - Operations have to be performed atomically

- Parallel programming has at least two new error classes
  1. Deadlocks
  2. Race conditions
- A race condition has resulted in a wrong result in our example
  - Incrementing a variable consists of three operations
    1. Loading the variable
    2. Modifying the variable
    3. Storing the variable
  - Operations have to be performed atomically

T0	T1	V
Load 0		0
Inc 1		0
Store 1		1
	Load 1	1
	Inc 2	1
	Store 2	2

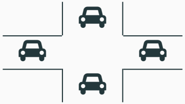
- Parallel programming has at least two new error classes
  1. Deadlocks
  2. Race conditions
- A race condition has resulted in a wrong result in our example
  - Incrementing a variable consists of three operations
    1. Loading the variable
    2. Modifying the variable
    3. Storing the variable
  - Operations have to be performed atomically

T0	T1	V
Load 0		0
Inc 1		0
Store 1		1
	Load 1	1
	Inc 2	1
	Store 2	2

T0	T1	V
Load 0		0
Inc 1	Load 0	0
Store 1	Inc 1	1
	Store 1	1

Debugging…

- Deadlocks cause parallel applications to stop progressing
  - Can have different causes, most often due to locking
  - May not be reproducible if there is time-dependent behavior
- Error condition can be difficult to find
  - Trying to lock an already acquired lock results in a deadlock
  - Erroneous communication patterns (everyone waits for the right neighbor)
- Error effect is typically easy to spot
  - Spinlocks or livelocks can look like computation, though



Debugging…

- Race conditions can lead to differing results
  - Debugging often hides race conditions
- Error condition is often very hard to find
  - Can be observed at runtime or be found by static analysis
  - Modern programming languages like Rust can detect data races
- Error effect is sometimes not observable
  - Slight variations in the results are not obvious
  - The correct result cannot be determined for complex applications
  - Repeating a calculation can be too costly

- Scalability of shared memory systems is limited
    - Current processors feature up to 64 cores with 128 threads
    - Typically two, at most four processors per node
- Computation is only one part of parallel applications
    - They need to store data in main memory and persist it to storage
    - Amount of main memory and storage per node is also limited
- To solve the biggest problems, we need distributed memory systems
    - These typically consist of a cluster of shared memory systems
    - Multiple nodes are connected via a so-called interconnect

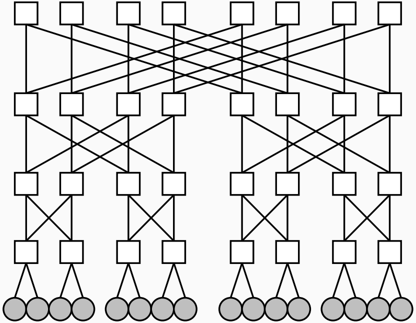
Networking Aspects...

- Processors require data fast
  - 3 GHz equals three operations per nanosecond
  - Even accessing the main memory is too slow
  - Multiple cache levels hide main memory latency
- Network and I/O extremely slow in comparison
  - Waiting for an HDD ruins performance
  - SSDs have alleviated the problem a bit

Level	Latency
L1 cache	$\approx$ 1 ns
L2 cache	$\approx$ 5 ns
L3 cache	$\approx$ 10 ns
RAM	$\approx$ 100 ns
InfiniBand	$\approx$ 500 ns
Ethernet	$\approx$ 100,000 ns
SSD	$\approx$ 100,000 ns
HDD	$\approx$ 10,000,000 ns

[Bonér, 2012] [Huang et al., 2014]

- Network topologies can get quite complex
    - Easy: All nodes are connected to a single switch
- Larger systems use hierarchical topologies
    - A fat tree has different throughputs depending on the tree level
- Fat trees can also have blocking factor (2:1)
    - Nodes in enclosure can communicate at 100 %
    - Enclosures in rack can communicate at 50 %
    - Racks can communicate at 25 %



[A5b, 2010]

Networking Aspects…

- Current network technologies feature high throughputs
  - InfiniBand can do up to 600 GBit/s
  - Ethernet can do up to 400 GBit/s
  - There are more technologies like Intel's Omni-Path
- Sophisticated approaches required to reach these high speeds
  - Kernel bypass to save context switches
  - Zero copy to avoid exhausting bus speeds

Programming with MPI

- Parallel applications can be run across multiple nodes
  - Typically as separate processes, requires message passing
  - MPI is the de-facto standard
- MPI offers operations for communication and more
  - Process groups and synchronization
  - Sending, receiving, reduction etc.
  - Point-to-point, collective or one-sided communication
- MPI also supports parallel I/O
  - Concurrent access to shared files

- Parallel application now runs as two independent processes
  - Processes can only see their own results, no shared memory
  - There is no risk of overwriting other values as in the OpenMP example
  - However, results have to be communicated between processes somehow

```
for (int i = 0; i < m/2; i++) {
    for (int j = 0; j < n; j++) {
        sum += arr[i][j];
    }
}
```

```
for (int i = m/2; i < m; i++) {
    for (int j = 0; j < n; j++) {
        sum += arr[i][j];
    }
}
```

- MPI allows us to perform efficient reduction operations
    - A predefined reduction operation is the sum

```c
MPI_Init(NULL, NULL);
for (int i = 0; i < m/2; i++) {
    for (int j = 0; j < n; j++) {
        sum += arr[i][j];
    }
}
MPI_Allreduce(&sum, &allsum, 1,
              MPI_INT, MPI_SUM,
              MPI_COMM_WORLD);
MPI_Finalize();
```

```c
MPI_Init(NULL, NULL);
for (int i = m/2; i < m; i++) {
    for (int j = 0; j < n; j++) {
        sum += arr[i][j];
    }
}
MPI_Allreduce(&sum, &allsum, 1,
              MPI_INT, MPI_SUM,
              MPI_COMM_WORLD);
MPI_Finalize();
```

- Application code is typically still contained in one file
    - MPI allows us to write a generic version of the application
    - We can determine our rank and the number of processes

```c
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
for (int i = (m/size) * rank; i < (m/size) * (rank + 1); i++) {
    for (int j = 0; j < n; j++) {
        sum += arr[i][j];
    }
}
MPI_Allreduce(&sum, &allsum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
MPI_Finalize();
```

Scalability

- When writing parallel applications, we must consider scalability
  - Scalability describes how an application behaves with increasing parallelism
- HPC systems are usually very expensive and should be used accordingly
  - Procurement costs can reach up to € 250,000,000
- To determine scalability, we have to analyze performance
  - HPC systems are complex, performance yield is often not optimal
  - Many different components interact with each other
    - Processors, caches, main memory, network, storage system etc.

Scalability…

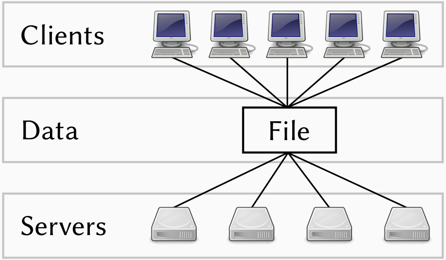
- In addition to procurement costs, operating is also quite expensive
  - 1. Frontier (USA): 1.2 EFLOPS at 22.7 MW ≈ € 52,700,000/a (in Germany)
  - 5. LUMI (Finland): 380 PFLOPS at 7.1 MW ≈ € 16,500,000/a (in Germany)
  - 74. Levante (Germany): 10 PFLOPS at 2 MW ≈ € 4,600,000/a
- Communication and I/O are often responsible for performance problems
  - High latency, which causes excessive waiting times for processors
  - Communication and I/O typically happen synchronously

Scalability…

- The performance improvement we get is called speedup
    - In the best case, the speedup is equal to the number of threads
    - In reality, the speedup is usually lower due to overhead
- Speedup can sometimes be higher than the number of threads
    - This is called a superlinear speedup and usually points at a problem
    - For example, each thread's data suddenly fits into the cache
        - This means that the measured problem became too small
        - Larger problems will not fit and therefore have a lower speedup

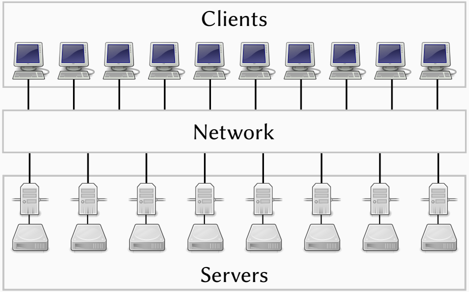
- Applications typically need input data and produce output data
  - I/O is an important aspect and can be relevant for overall performance
  - Without I/O, the results of a scientific application would be lost
- Applications often run for multiple days or weeks
  - To cope with crashes, it is necessary to write checkpoints
  - Jobs are often only allowed to run for a few hours at a time
- As mentioned before, storage devices have high latencies
  - Waiting for I/O usually impacts performance negatively
  - File systems try to cache data aggressively to hide latency

- Access via parallel distributed file systems
  - Allow concurrent access from clients
  - Distribute data across servers
- Clients can access a shared file
  - Everyone can read input and write results
  - Necessary for parallel applications
- Servers share the load
  - Files are split up and distributed
  - Use capacity and throughput of many servers



Parallel I/O...

- Computation and storage usually separated
  - Can be optimized for respective workloads
  - No interference of other components
- Clients run parallel applications
  - Small local storage for OS and caching
  - Access to the file system via the network
  - No direct access to file system's devices
- Servers store data and metadata
  - Typically servers with many HDDs and SSDs



Outline

Summary

- Parallel programming is an important skill
  - Current computers always have multiple cores or processors
- Parallelization is used to improve performance
  - It is necessary to understand the hardware and keep scalability in mind
- Shared memory and distributed memory are the two main architectures
  - Threads can be used for shared memory systems
  - Message passing is often used for distributed memory systems
- Parallel applications can have deadlocks and race conditions
  - These errors can be hard to find and non-deterministic
- Parallel I/O is an important part of parallel applications
  - I/O is necessary to read input data and store results

References

[A5b, 2010] A5b (2010). **Fat tree topology of networks.** https://commons.wikimedia.org/wiki/File:Fat-tree1.svg.

[Bonér, 2012] Bonér, J. (2012). **Latency Numbers Every Programmer Should Know.** https://gist.github.com/jboner/2841832.

[Huang et al., 2014] Huang, J., Schwan, K., and Qureshi, M. K. (2014). **NVRAM-aware Logging in Transaction Systems.** *Proc. VLDB Endow.*, 8(4):389–400.

[NOAA, 2007] NOAA (2007). **Schematic for Global Atmospheric Model.** https://celebrating200years.noaa.gov/breakthroughs/climate_model/AtmosphericModelSchematic.png.