Introduction

Parallel Programming

2024-04-10

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

Introduction

Organization

Lecture

Exercises

Outlook

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

1. Expert
2. Advanced
3. Beginner
4. Not at all
• How familiar are you with Git?
  1. Expert
  2. Advanced
  3. Beginner
  4. Not at all
Lecture and Exercises

Organization

- Lecture: Wednesdays, 17:15–18:45
  - Foundation and background of parallel programming
  - Lecture will be recorded for later viewing
  - We will also use this time slot to clear up questions etc.
- Exercises: Wednesdays, 11:15–12:45
  - Practical exercises about parallel programming
  - We will discuss solutions and take a look at the next exercise sheet
- Exam: Written
• 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
• High Performance Computing: Modern Systems and Practices (Thomas Sterling, Matthew Anderson and Maciej Brodowicz)

• Parallel Programming: for Multicore and Cluster Systems (Thomas Rauber and Gudula Rünger) (e-book at UB)

• Parallel Programming: Concepts and Practice (Dr. Bertil Schmidt, Dr. Jorge Gonzalez-Domínguez, Christian Hundt and Moritz Schlarb) (book at UB)
Outline

Introduction

Organization

Lecture

Exercises

Outlook

Summary
- Introduction (10 April – today 😊)
  - A brief overview of some topics we will cover in the lecture
  - This is an outlook, no need to understand everything immediately

- Performance Analysis and Optimization (17 April)
  - How to measure performance correctly and identify relevant components
  - Math, code and compiler optimizations

- Hardware Architectures (24 April)
  - Differences between shared and distributed memory
  - Non-uniform memory access

- Parallel Programming (8 May)
  - How to parallelize problems
  - Potential problems and new kinds of errors
• Programming with OpenMP (22 May)
  - High-level parallelization using compiler annotations
  - Loops, tasks, synchronization etc.
• Operating System Concepts (29 May)
  - Differences between processes and threads
  - Shared memory regions, I/O, scheduling etc.
• Programming with POSIX Threads (5 June)
  - Low-level parallelization using library functions
  - Thread creation, joining, synchronization, condition variables etc.
• Programming with MPI (12 June)
  - Parallelization using the Message Passing Interface
  - Communication, I/O, collective operations etc.
• Networking and Scalability (19 June)
  • Performance metrics for network technologies and topologies
  • Scalability considerations for large systems

• Advanced MPI and Debugging (26 June)
  • Advanced concepts for message passing applications (such as RMA)
  • How to debug parallel programs using multiple threads and processes

• Parallel I/O (3 July)
  • Why parallel I/O is needed in parallel applications
  • Architecture of parallel distributed file systems

• Research Talks (10 July)
  • Research topics currently investigated in our group
• Exercises will consist of parallel programming in C
  • 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 work mostly on our cluster via SSH
  • Logging in and setting everything up will be part of the first exercise
• Introduction and setup (10 April)
  • Log in to cluster, set up software environment etc.
• Debugging (22 April)
  • Using GDB, Valgrind etc.
• Performance optimization (29 April)
  • Optimizing a serial application
• Parallelizing with OpenMP and parallelization schema (20 May)
  • Preparing a parallelization schema for the serial application
  • Parallelizing the optimized application with OpenMP
• Parallelizing with POSIX Threads (3 June)
  • Parallelizing the optimized application with POSIX Threads
• Introduction to MPI (10 June)
  • Getting familiar with the Message Passing Interface
• Parallelizing with MPI (Jacobi) (17 June)
  • Parallelizing the optimized application with MPI
Outline

Introduction
  Organization
  Lecture
  Exercises
Outlook
Summary
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)
Performance Analysis and Optimization

Outlook

- It is difficult to measure performance correctly
  - There are many factors and components to consider
  - Performance is influenced by caching, network, input/output (I/O) etc.
  - Errors can influence or even invalidate all results
- Optimization requires deep knowledge of the hardware
  - How do the different levels of caches interact?
  - Can we reach the main memory from all cores with the same speed?
  - How does our application behave with more cores?
• There are also technical issues to take into account
  • HPC applications are typically run via a batch scheduler
  • Operating system services can influence performance

• Measuring performance can be hard
  • Which components are involved and have to be measured?
  • Which performance can we expect on a given system?
Hardware Architectures

Outlook

• 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
• 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
- 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 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
Parallel Programming...

• 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]
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

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

```c
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];
    }
}
```

- 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

```
1 $ ./openmp
2 sum=3773725
3 $ ./openmp
4 sum=4012997
5 $ ./openmp
6 sum=12325088
7 $ ./openmp
8 sum=2456866
9 $ ./openmp
10 sum=11970989
11 $ ./openmp
12 sum=2818054
13 $ ./openmp
14 sum=3979092
```
Parallelization with OpenMP...

Outlook

- 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)
• 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

<table>
<thead>
<tr>
<th></th>
<th>T0</th>
<th>T1</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load 0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Inc 1</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Store 1</td>
<td></td>
<td>Inc 1</td>
<td>1</td>
</tr>
<tr>
<td>Load 1</td>
<td>Inc 2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Store 2</td>
<td></td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
• 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
• 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
• 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

<table>
<thead>
<tr>
<th>Level</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>≈ 1 ns</td>
</tr>
<tr>
<td>L2 cache</td>
<td>≈ 5 ns</td>
</tr>
<tr>
<td>L3 cache</td>
<td>≈ 10 ns</td>
</tr>
<tr>
<td>RAM</td>
<td>≈ 100 ns</td>
</tr>
<tr>
<td>InfiniBand</td>
<td>≈ 500 ns</td>
</tr>
<tr>
<td>Ethernet</td>
<td>≈ 100,000 ns</td>
</tr>
<tr>
<td>SSD</td>
<td>≈ 100,000 ns</td>
</tr>
<tr>
<td>HDD</td>
<td>≈ 10,000,000 ns</td>
</tr>
</tbody>
</table>

[Boner, 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%
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
• 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

```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];
    }
}
```
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();
```
Programming with MPI...

- 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();
```
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.
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
• 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
Parallel I/O...

- 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
• 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

Introduction

Organization

Lecture

Exercises

Outlook

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


   https://celebrating200years.noaa.gov/breakthroughs/climate_model/AtmosphericModelSchematic.png.