# Parallel Programming

## Parallel Programming

2024-11-04

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


Parallel Programming

- Review
- Introduction
- Parallelization Approaches
- Parallelization Techniques
- Synchronization
- Summary

- Why are current processors increasing the core count instead of the clock rate?
  1. Higher clock rates require changing applications
  2. Increasing the clock rate also increases heat dissipation
  3. It is cheaper because cores can be interconnected more easily
  4. Additional cores increase memory throughput and graphics performance

- Which is the most-used architecture today?
  1. SISD: Single instruction stream, single data stream
  2. SIMD: Single instruction stream, multiple data streams
  3. MISD: Multiple instruction streams, single data stream
  4. MIMD: Multiple instruction streams, multiple data streams

- Which architecture requires explicit message passing?
    1. Shared memory
    2. Distributed memory
    3. Shared distributed memory
    4. Non-uniform memory access

- Which network topology requires only a single switch?
    1. Bus
    2. Ring
    3. Star
    4. Fat tree

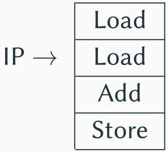
# Outline


Parallel Programming

- Review
- Introduction
- Parallelization Approaches
- Parallelization Techniques
- Synchronization
- Summary

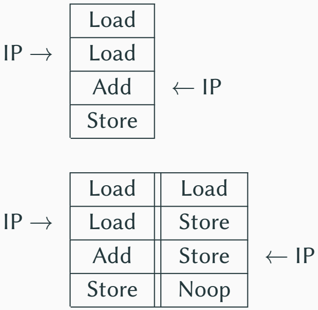
# Serial Computation

- Applications are traditionally written for serial execution
  - Statements are turned into instructions by compiler/interpreter
  - Instructions are executed serially by a single processor core
  - Only one instruction can be executed at a time
  - Instruction pointer (IP) indicates current instruction
- Performance is limited by clock rate of the single core
  - Clock rate cannot be increased further due to heat issues
  - Additional limitations due to memory and storage bandwidth



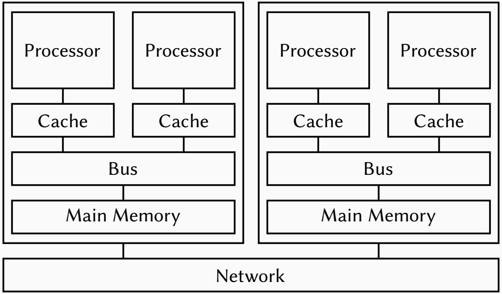
# Parallel Computation

- Parallel applications execute instructions concurrently
    - Problem has to be separated into concurrent parts
- Parallel computers have multiple processing units
    - Allows working on problems concurrently
    - Can describe different resources: ALU, FPU, core etc.
- Does not necessarily have to execute the same code
    - Different applications can run at the same time



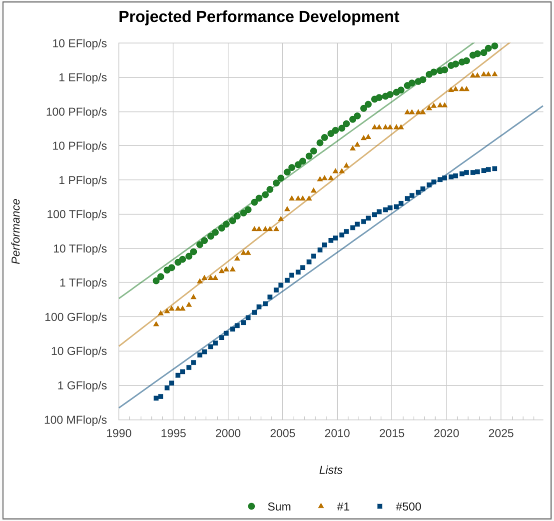
# Architectures

- Memory access model as classifier
  - Determines programming model
- Shared memory
  - Processors consist of multiple cores
  - Access to shared memory via a bus
  - Limited scalability
- Distributed memory
  - Processors only have access to own memory
  - Machines are connected via a network
  - Better scalability



# Performance Development

- TOP500 list for supercomputers
  - Collected since the 1990s
- Exponential performance growth
  - Historically, factor 300–400 every ten years
  - Increase has slowed down significantly




[TOP500.org, 2024]

- 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
  - In HPC, OpenMP and MPI are often used together

# Outline

Parallel Programming

- Review
- Introduction
- Parallelization Approaches
- Parallelization Techniques
- Synchronization
- Summary

# Overview

- Parallelization consists of several aspects
  1. Take existing algorithm and try to make it run in parallel
  2. Come up with new algorithm that supports parallelism
  3. Implement the algorithm in a way that allows parallel execution
- Application and data are distributed across resources
  - Related to SPMD and MPMD
- Different parallelization approaches
  - Automatically, semi-automatically or manually

# Overview...

- Parallelization introduces additional overhead
  - Either within in the application or the surrounding environment
  - Some form of coordination is always required
- Aim for optimal use of resources
  - Using many components in parallel increases costs
  - Optimal use is difficult to achieve, especially with overhead
- There are different kinds of overhead
  - Additional computations are required due to distribution
  - Partitioning the problem introduces more work
  - Communication and synchronization are required to coordinate
  - Transformations for coupled applications

# SPMD

- The most important architecture today is MIMD
  - SPMD and MPMD are high-level concepts that are often used on MIMD
- SPMD: Single program, multiple data streams
  - All tasks execute same application but at different points
  - Application can use threads, message passing etc.
  - Tasks use different data, for instance, using domain decomposition
  - There is typically logic to execute only parts of the application
    - For instance, coordination is performed by the first task

# SPMD. . .

- SPMD distributes data across threads/processes
  - Code is identical but can still perform different tasks
  - Often used in combination with domain decomposition
    - For instance, two-dimensional matrix is the problem domain
- Decomposition is critical for achievable performance
  - Rows might be faster than columns depending on memory layout
  - Size of sub-domains determines load of each task
- Distribution also determines communication schema
  - Communication might have to be performed at boundaries

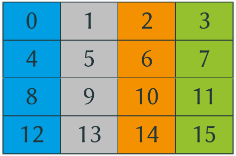
# SPMD...

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

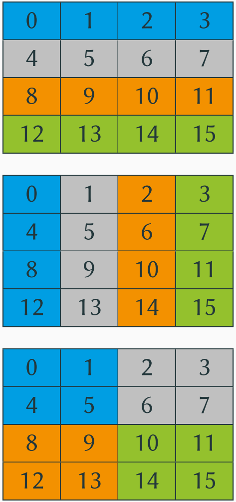
- SPMD distributes data across threads/processes
  - Code is identical but can still perform different tasks
  - Often used in combination with domain decomposition
    - For instance, two-dimensional matrix is the problem domain
- Decomposition is critical for achievable performance
  - Rows might be faster than columns depending on memory layout
  - Size of sub-domains determines load of each task
- Distribution also determines communication schema
  - Communication might have to be performed at boundaries

# SPMD. . .

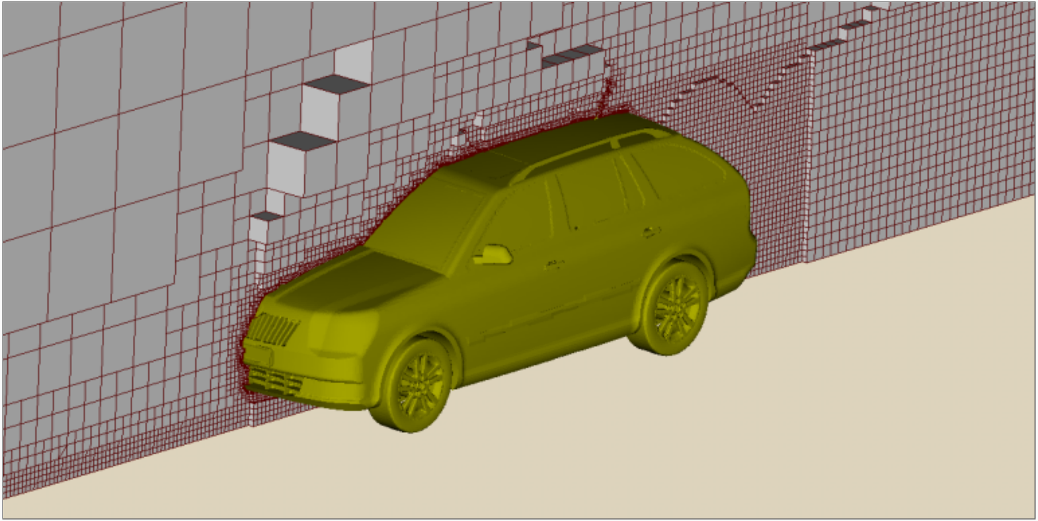
- SPMD distributes data across threads/processes
    - Code is identical but can still perform different tasks
    - Often used in combination with domain decomposition
        - For instance, two-dimensional matrix is the problem domain
- Decomposition is critical for achievable performance
    - Rows might be faster than columns depending on memory layout
    - Size of sub-domains determines load of each task
- Distribution also determines communication schema
    - Communication might have to be performed at boundaries

- SPMD distributes data across threads/processes
  - Code is identical but can still perform different tasks
  - Often used in combination with domain decomposition
    - For instance, two-dimensional matrix is the problem domain
- Decomposition is critical for achievable performance
  - Rows might be faster than columns depending on memory layout
  - Size of sub-domains determines load of each task
- Distribution also determines communication schema
  - Communication might have to be performed at boundaries



# SPMD...



[Greenshields, 2016]

# SPMD...

- Benefits
  - Relatively easy to adapt to the available hardware
    - For example, increasing the matrix size allows using more main memory
    - More tasks can be added by changing the decomposition slightly
  - Communication schemata are typically easy
    - Communication usually only happens at the sub-domain boundaries
  - Debugging is much easier since only one program is involved
- Drawbacks
  - Sometimes not appropriate for algorithm
  - Load balacing might be difficult for dynamic problems

# MPMD

- MPMD: Multiple programs, multiple data streams
    - Tasks execute different applications with different purposes
    - Application can use threads, message passing etc.
    - Tasks use different data, for instance, supplied by previous task
    - There is usually a functional decomposition
        - For instance, first task does pre-processing, last task does post-processing

# MPMD...

- MPMD distributes functionality across processes/threads
  - Different code is distributed across tasks
  - Often used in combination with functional decomposition
    - For instance, chain of operations performed on pictures
- Not as common as SPMD due to specific requirements
  - Problem has to be able to be partitioned into multiple programs
  - For instance, pre-process, calculation and finally post-process
- Good fit for chains of operations
  - Compression, transformation, encryption etc.

# MPMD...

- MPMD distributes functionality across processes/threads
  - Different code is distributed across tasks
  - Often used in combination with functional decomposition
    - For instance, chain of operations performed on pictures
- Not as common as SPMD due to specific requirements
  - Problem has to be able to be partitioned into multiple programs
  - For instance, pre-process, calculation and finally post-process
- Good fit for chains of operations
  - Compression, transformation, encryption etc.

$$x' = f(x)$$

# MPMD…

- MPMD distributes functionality across processes/threads
  - Different code is distributed across tasks
  - Often used in combination with functional decomposition
    - For instance, chain of operations performed on pictures
- Not as common as SPMD due to specific requirements
  - Problem has to be able to be partitioned into multiple programs
  - For instance, pre-process, calculation and finally post-process
- Good fit for chains of operations
  - Compression, transformation, encryption etc.

$$x' = f(x)$$

$$\downarrow$$

$$x'' = g(x')$$

# MPMD…

- MPMD distributes functionality across processes/threads
  - Different code is distributed across tasks
  - Often used in combination with functional decomposition
    - For instance, chain of operations performed on pictures
- Not as common as SPMD due to specific requirements
  - Problem has to be able to be partitioned into multiple programs
  - For instance, pre-process, calculation and finally post-process
- Good fit for chains of operations
  - Compression, transformation, encryption etc.

$$x' = f(x) \downarrow x'' = g(x') \downarrow x''' = h(x'')$$

# MPMD. . .

- Benefits
  - Appropriate for some widely available algorithms
    - Signal processing can run signal through multiple filters
- Drawbacks
  - Can be hard to tune for the available hardware
    - Requires more data to be available or functionality to be separated
  - Communication can become relatively complex
  - Debugging is complicated since multiple programs have to be watched
  - Balancing the computational load might also be complicated

# SPMD vs. MPMD

- Some applications combine the SPMD and MPMD approaches
  - Distributing data and functionality across threads/processes
- Climate models are a good example
  - Climate can be seprated into different components
    - For instance, atmosphere, ocean, ice etc.
  - Each of of the components is too big to solve serially
    - Atmospheric data is distributed across tasks
  - Couplers are used to connect the components

# Load Balancing

- Load distribution might be done statically or dynamically
  - Load balancing means that we want to keep all tasks busy
- Static distribution is relatively easy
  - Distribute data or loop iterations evenly if work is similar
    - This is often the case for numerical applications
  - Might result in load imbalance for varying computational work
    - For instance, particles migrate across domain boundaries
- Dynamic distribution requires more coordination
  - Might be done by a scheduler or a dedicated coordination task
  - Results in better load balance for varying computational work

# Quiz

- What is the best way to distribute a matrix using SPMD?
    1. Each process holds one element
    2. Each process holds one row/column
    3. Each process holds several rows/columns
    4. Each process holds a sub-matrix

# Outline


Parallel Programming

- Review
- Introduction
- Parallelization Approaches
- Parallelization Techniques
- Synchronization
- Summary

# Approaches and Technologies

- Different approaches for parallelization
  - Automatic parallelization by the compiler
  - Semi-automatic parallelization by the compiler
  - Manual parallelization by the developer
- Technologies
  - Runtimes and libraries
  - Language extensions and new languages

# Automatic Parallelization

- Automatic parallelization can be done by the compiler
  - There are approaches using Fortran etc.
  - Compiler has to analyze data dependencies and determine feasibility
  - Compiler can then distribute data/loops/etc. across resources
- Performance of existing solutions is usually not optimal
  - Sometimes parallelization cannot be performed at all
- As with all automatic approaches, limited to particular patterns

# Semi-Automatic Parallelization

- Semi-automatic parallelization is supported by the compiler
    - Developers have to identify opportunities for parallelization
    - Specifying compiler pragmas can give hints to the compiler
    - This may be combined with the automatic parallelization approach
- Most commonly used for shared memory
    - One popular example is OpenMP, which uses threads
        - Barriers, critical regions, atomic operations, reduction, tasks etc.
- There are also approaches for distributed memory
    - For instance, Chapel can distribute across multiple nodes

# Manual Parallelization

- Manual parallelization puts the burden on the developer
  - Developers have to understand the problem at hand
  - Analyze algorithm for potential parallelism
- First step: Identify hotspots
  - Parallelize those first, since they require most computation
  - If possible, use optimized software and libraries
- Identify bottlenecks
  - Bottlenecks can limit performance if scaled up
- There are also algorithms that are hard to parallelize
  - One example is the Fibonacci sequence: $f(n) = f(n-1) + f(n-2)$
  - A possible solution is using another algorithm

# Runtimes and Libraries

- Runtimes can take care of distributing tasks intelligently
  - For instance, submit many small tasks, runtime schedules them
  - This is often limited to the task level
- Libraries can support a wide range of use cases
  - MPI offers communication and more for parallel programs
  - POSIX Threads is a library-based approach for thread programming
    - Support for barriers, semaphors, mutexes, condition variables etc.

# Language Extensions and New Languages

- Language extensions retrofit existing languages with support for parallelism
  - High Performance Fortran adds `FORALL` loops and more
  - C has native support for threads starting with C11
- New languages include parallelism into the core language design
  - Go has support for channels that can be used for parallelization
  - Rust can detect data races at compile time due to its ownership concept
  - Chapel, Erlang etc. have native support for distributed applications

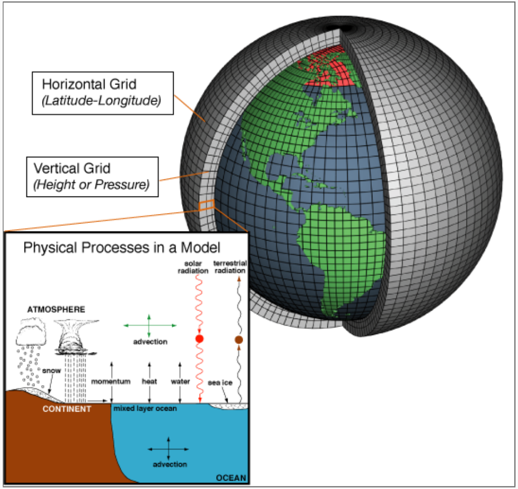
# Numerical vs. Non-Numerical Problems

- Significant differences between numerical vs. non-numerical problems
  - Numerical: Weather, climate, fluid dynamics etc.
  - Non-numerical: Search engines, databases etc.
- Grand Challenges (US National Computing Research)
  - 1980: More funding for HPC in general
    - Computational fluid dynamics, electronic structure calculations, plasma dynamics, fundamental nature of matter, symbolic computations
  - 2000: Removing mostly completed research, adding new areas
    - Climate change, biological systems, virtual product design, cancer detection and therapy, modelling hazards

# Numerical Problems

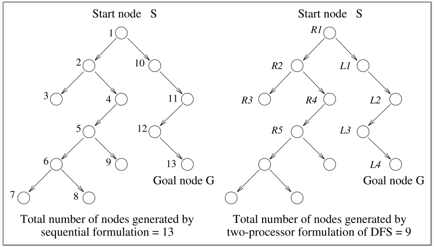
- Numerical problems are mostly iterative
  - For instance, simulations are often performed in time steps
  - Number of threads/processes is typically static
- Usually have global conditions for termination
  - In the easiest case, run for a specified number of time steps
  - Alternatively, run until a condition is met
- Data structures are often regular
  - Data can often be stored in one or more matrices
  - Dimensionality of the matrices depends on the problem
  - Communication schemata are typically regular

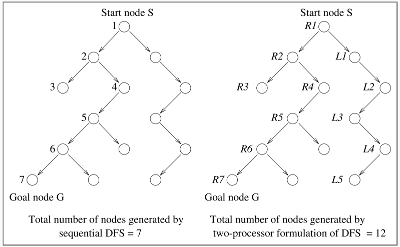
- Many phenomena are highly parallel
  - Examples include galaxies, planets, climate and weather
- Many problems are very big or complex
  - Infeasible to solve them serially
  - Weather simulation has to be finished before it actually happens ☺
- Parallel computing is well-suited
  - Data and components can be distributed




[NOAA, 2007]

- Parallelism also for non-numerical problems
  - Search algorithms, databases etc.
  - For instance, databases have to process many requests in parallel
- Some differences to numerical problems
  - Speedup for tree searches depends on location
  - Parallelism might result in redundant work




[Grama et al., 2003]

- Can this loop be parallelized automatically?
  1. Yes
  2. No, `while` loops cannot be parallelized
  3. No, there are dependency issues

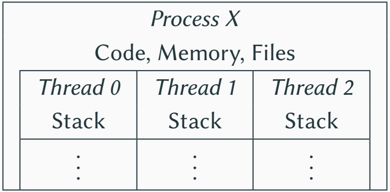
```
while (true) {
    c = calculate(c, ...);

    if (c > x) {
        break;
    }
}
```

# Outline

Parallel Programming

- Review
- Introduction
- Parallelization Approaches
- Parallelization Techniques
- Synchronization
- Summary

# Threads vs. Processes




- Processes are instances of an application
  - Applications can be started multiple times
  - Processes are isolated from each other by the operating system for security reasons
  - 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
- We will only take a look at threads for now
  - Message passing will be covered later

# Threads vs. Processes...

- 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)

# Example

- OpenMP allows parallelizing applications using compiler instructions
  - Very convenient for users since no internals have to be known
  - Reduced feature set in comparison to low-level approaches

# Example

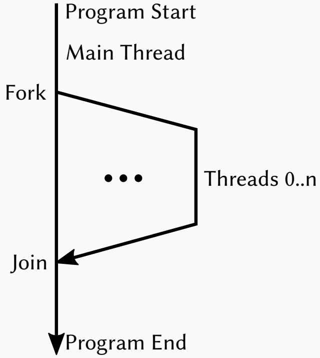
- OpenMP allows parallelizing applications using compiler instructions
  - Very convenient for users since no internals have to be known
  - Reduced feature set in comparison to low-level approaches

```
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
    for (i = 0; i < 100000000; i++) {
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```

# Example

- OpenMP allows parallelizing applications using compiler instructions
  - Very convenient for users since no internals have to be known
  - Reduced feature set in comparison to low-level approaches

```c
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
    for (i = 0; i < 100000000; i++) {
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```



# Example

- OpenMP allows parallelizing applications using compiler instructions
  - Very convenient for users since no internals have to be known
  - Reduced feature set in comparison to low-level approaches

```c
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
    for (i = 0; i < 100000000; i++) {
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```

```
$ time OMP_NUM_THREADS=1 ./openmp0
Iterations: 100000000
[...] 99% cpu 0,227 total
```

# Example

- OpenMP allows parallelizing applications using compiler instructions
  - Very convenient for users since no internals have to be known
  - Reduced feature set in comparison to low-level approaches

```c
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
    for (i = 0; i < 100000000; i++) {
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```

```
$ time OMP_NUM_THREADS=1 ./openmp0
Iterations: 100000000
[...]  99% cpu 0,227 total
```

```
$ time OMP_NUM_THREADS=2 ./openmp0
Iterations: 51147874
[...] 198% cpu 0,425 total
```

(or another number between 2 and 100,000,000)

# 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

# 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

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

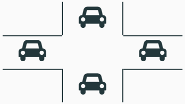
- Parallel programming has at least two new error classes
  1. Deadlocks
  2. Race conditions
- A race condition has resulted in a wrong result in our example
  - Incrementing a variable consists of three operations
    1. Loading the variable
    2. Modifying the variable
    3. Storing the variable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

# Locks

- `critical` protects an instruction or a scope with a lock
    - The locked part can only be entered by one thread at a time
    - It is possible to use `atomic` for simple instructions

- critical protects an instruction or a scope with a lock
  - The locked part can only be entered by one thread at a time
  - It is possible to use atomic for simple instructions

```c
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
    for (i = 0; i < 100000000; i++) {
        #pragma omp critical
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```

# Locks

- `critical` protects an instruction or a scope with a lock
  - The locked part can only be entered by one thread at a time
  - It is possible to use `atomic` for simple instructions

```c
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
    for (i = 0; i < 100000000; i++) {
        #pragma omp critical
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```

| Thread 0 | Thread 1 | V |
|---|---|---|
| Load 0 | | 0 |
| Inc 1 | | 0 |
| Store 1 | | 1 |
| | Load 1 | 1 |
| | Inc 2 | 1 |
| | Store 2 | 2 |
| ⋮ | ⋮ | ⋮ |

- critical protects an instruction or a scope with a lock
  - The locked part can only be entered by one thread at a time
  - It is possible to use atomic for simple instructions

```c
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
    for (i = 0; i < 100000000; i++) {
        #pragma omp critical
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```

```
$ time OMP_NUM_THREADS=1 ./openmp1
Iterations: 100000000
[...]  99% cpu 1,464 total
```

# Locks

- critical protects an instruction or a scope with a lock
  - The locked part can only be entered by one thread at a time
  - It is possible to use atomic for simple instructions

```c
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
    for (i = 0; i < 100000000; i++) {
        #pragma omp critical
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```

```
$ time OMP_NUM_THREADS=1 ./openmp1
Iterations: 100000000
[...]  99% cpu 1,464 total
```

```
$ time OMP_NUM_THREADS=2 ./openmp1
Iterations: 100000000
[...] 194% cpu 6,615 total
```

# Reduction

- An alternative solution for the problem uses a `reduction` variable
  - Each thread has a separate private copy of the variable
  - At the end of the parallel region, all variables are reduced to one result

# Reduction

- An alternative solution for the problem uses a reduction variable
  - Each thread has a separate private copy of the variable
  - At the end of the parallel region, all variables are reduced to one result

```c
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
        ↪ reduction(+:iters)
    for (i = 0; i < 100000000; i++) {
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```

# Reduction



- An alternative solution for the problem uses a reduction variable
  - Each thread has a separate private copy of the variable
  - At the end of the parallel region, all variables are reduced to one result

```c
int main (void) {
	int i, iters = 0;

	#pragma omp parallel for
	    ↪ reduction(+:iters)
	for (i = 0; i < 100000000; i++) {
		iters++;
	}
	printf("Iterations: %d\n", iters);
	return 0;
}
```

| V0 | Thread 0 | V1 | Thread 1 | V |
|---|---|---|---|---|
| 0 | Load 0 | 0 | Load 0 | |
| 0 | Inc 1 | 0 | Inc 1 | |
| 1 | Store 1 | 1 | Store 1 | |
| ⋮ | ⋮ | ⋮ | ⋮ | |
| 50M | + | 50M | = | 100M |

# Reduction

- An alternative solution for the problem uses a reduction variable
  - Each thread has a separate private copy of the variable
  - At the end of the parallel region, all variables are reduced to one result

```c
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
        ↪ reduction(+:iters)
    for (i = 0; i < 100000000; i++) {
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```

```
$ time OMP_NUM_THREADS=1 ./openmp2
Iterations: 100000000
[...]  99% cpu 0,216 total
```

# Reduction

- An alternative solution for the problem uses a reduction variable
  - Each thread has a separate private copy of the variable
  - At the end of the parallel region, all variables are reduced to one result

```c
int main (void) {
    int i, iters = 0;

    #pragma omp parallel for
        ↪ reduction(+:iters)
    for (i = 0; i < 100000000; i++) {
        iters++;
    }
    printf("Iterations: %d\n", iters);
    return 0;
}
```

```
$ time OMP_NUM_THREADS=1 ./openmp2
Iterations: 100000000
[...]  99% cpu 0,216 total
```

```
$ time OMP_NUM_THREADS=2 ./openmp2
Iterations: 100000000
[...] 197% cpu 0,106 total
```

- Why does the incorrect version get slower?
  1. Access conflicts on shared variable
  2. Increased memory traffic by threads
  3. Not slower because CPU time is measured

```
$ time OMP_NUM_THREADS=1 ./openmp0
Iterations: 100000000
[...]  99% cpu 0,227 total
```

```
$ time OMP_NUM_THREADS=2 ./openmp0
Iterations: 51147874
[...] 198% cpu 0,425 total
```

# Outline

Parallel Programming

- Review
- Introduction
- Parallelization Approaches
- Parallelization Techniques
- Synchronization
- Summary

# Summary

- Parallel applications can execute instructions concurrently
  - Parallelization is necessary to solve complex problems
- SPMD and MPMD are high-level programming concepts
  - SPMD distributes data cross tasks, while MPMD distributes functionality
- Parallelization can be done automatically, semi-automatically or manually
  - Compilers are not smart enough to do all the work for us (yet)
- Synchronization and communication are relevant on all abstraction levels
  - Real-world applications usually use hybrid approaches with MPI and OpenMP

# References


[Grama et al., 2003] Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). **Speedup Anomalies in Parallel Search Algorithms.** https://www-users.cs.umn.edu/~karypis/parbook/.

[Greenshields, 2016] Greenshields, C. (2016). **Slice through mesh for aerodynamics of a car, showing varying cell size.** https://cfd.direct/openfoam/computational-fluid-dynamics/.

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

[TOP500.org, 2024] TOP500.org (2024). **Projected Performance Development.** https://top500.org/statistics/perfdevel/.