An Introduction into Parallelization

Multithreading and Multiprocessing for Beginners

Jascha Schewtschenko

Institute of Cosmology and Gravitation, University of Portsmouth

May 9, 2018

Outline

Concepts and Terminology

Concepts and Terminology

CPU/Processor/Core while technically nowadays each CPU/processor hosts more than one core, we use this terms interchangeably

Node A 'standalone' unit consisting of its own CPUs, memory (& storage).

Process/Task logically discrete section of computational work - typically a program or program-like set of instructions that is executed by a processor

Thread part of the computational work of a process that is executed in parallel on an additional processor

Observed speed-up ratio between wall-clock time of serial and parallelized code

Parallel overhead Additional amount of time/resources required to run parallelized code (e.g. start-up time and memory usage of framework, data comm., synchronization)

Concepts and Terminology (cont.)

Throughput amount of (sub)tasks/data processed per time unit

Latency delay between invoking the operation and getting the response (e.g. finishing a task)

Massively Parallel Refers to the hardware that comprises a given parallel system - having many processing elements (the meaning of "many" keeps increasing)

Embarrassingly Parallel Solving many similar, but independent tasks simultaneously; little to no need for coordination between the tasks

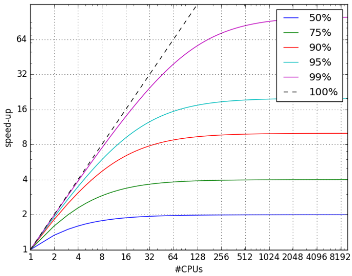
Concepts and Terminology - Amdahl's Law

- theoretical speedup in *latency* S_{latency} of execution of task with fixed workload:

Amdahl's law

$$S_{\text{latency}} = \frac{1}{(1-p) + \frac{p}{s}}$$

p is parallelizable fraction of code
s its speed-up



Concepts and Terminology - Amdahl's Law

- theoretical speedup in *latency* S_{latency} of execution of task with fixed workload:

Amdahl's law

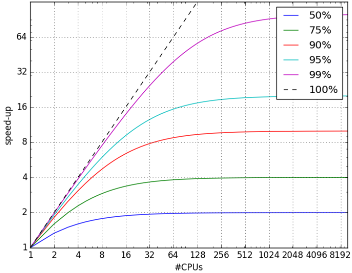
$$S_{\text{latency}} = \frac{1}{(1-p) + \frac{p}{s}}$$

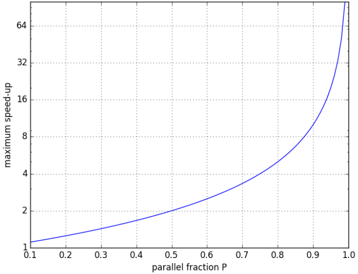
p is parallelizable fraction of code
s its speed-up

- From this follows

$$\lim_{s \to \infty} S_{\text{latency}} = \frac{1}{1-p}$$

i.e. never speeds up more than the inverse serial fraction of code





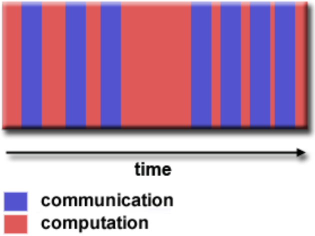
Concepts and Terminology - Granularity

- Computation / Communication Ratio

Concepts and Terminology - Granularity

- Computation / Communication Ratio

  fine-grained: frequent communication; facilitates e.g. load balancing, comes with overhead costs

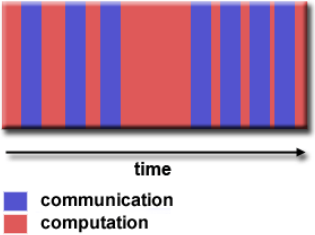


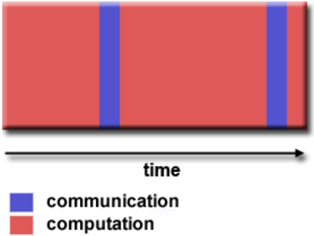
Concepts and Terminology - Granularity

- Computation / Communication Ratio

  fine-grained: frequent communication; facilitates e.g. load balancing, comes with overhead costs

  coarse-grained: less frequent comm.; lower communication costs, but potentially poorer load balancing





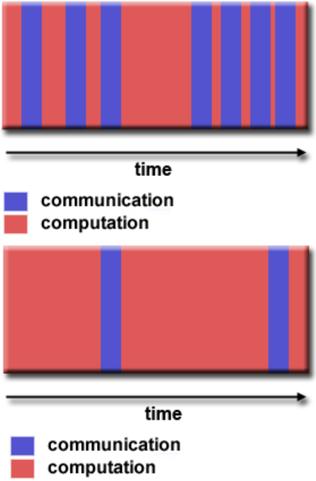
Concepts and Terminology - Granularity

- Computation / Communication Ratio

  fine-grained frequent communication; facilitates e.g. load balancing, comes with overhead costs

  coarse-grained less frequent comm.; lower communication costs, but potentially poorer load balancing

- best choice dependents on circumstances



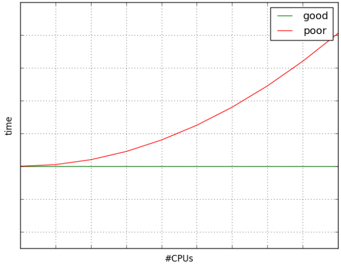
Concepts and Terminology - Scalability

Ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources:

Concepts and Terminology - Scalability

Ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources:

Weak scaling
: for running larger problem while fixing the problem size per processor

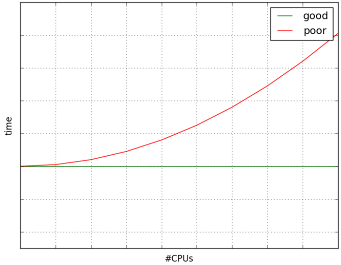


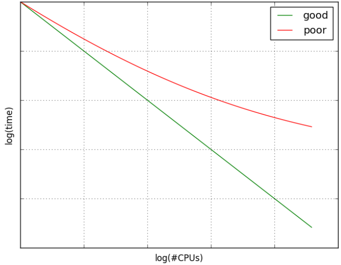
Concepts and Terminology - Scalability

Ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources:

Weak scaling for running larger problem while fixing the problem size per processor

Strong scaling for running the same problem size in less time



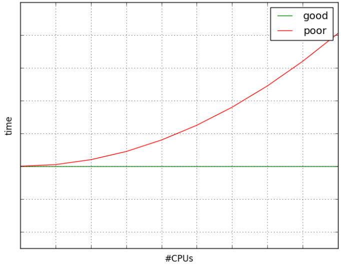


Concepts and Terminology - Scalability

Ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources:

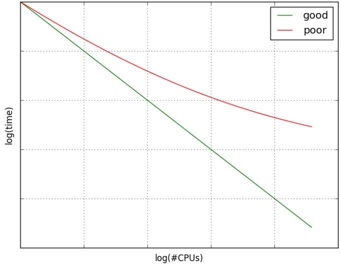
Weak scaling
: for running larger problem while fixing the problem size per processor

Strong scaling
: for running the same problem size in less time



Factors affecting scalability:

- I/O bandwidth (for RAM, storage and communication)
- imperfect/impossible load balancing
- overhead on comm. (e.g. exchange of padding around domain)
- limitations of parallel support libraries / parallel overhead



Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but total computation time and use of memory increases). Additionally, the increased complexity comes with increased development costs for:

Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but total computation time and use of memory increases). Additionally, the increased complexity comes with increased development costs for:

- Design

Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but total computation time and use of memory increases). Additionally, the increased complexity comes with increased development costs for:

- Design
- Coding

Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but total computation time and use of memory increases). Additionally, the increased complexity comes with increased development costs for:

- Design
- Coding
- Debugging

Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but total computation time and use of memory increases). Additionally, the increased complexity comes with increased development costs for:

- Design
- Coding
- Debugging
- Tuning

Concepts and Terminology - Cost of Complexity

Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but total computation time and use of memory increases). Additionally, the increased complexity comes with increased development costs for:

- Design
- Coding
- Debugging
- Tuning
- Maintenance

Concepts and Terminology - Cost of Complexity

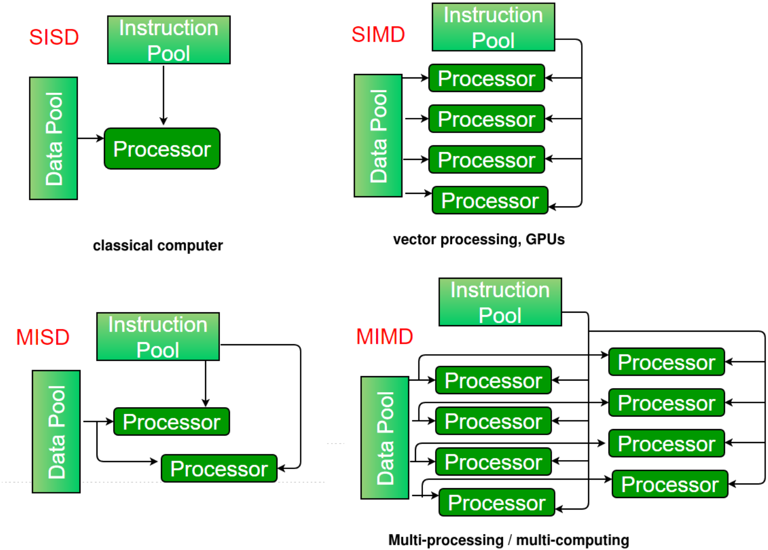
Anything short of a perfect scalability costs more resources in total (i.e. wall time may be lower but total computation time and use of memory increases). Additionally, the increased complexity comes with increased development costs for:

- Design
- Coding
- Debugging
- Tuning
- Maintenance

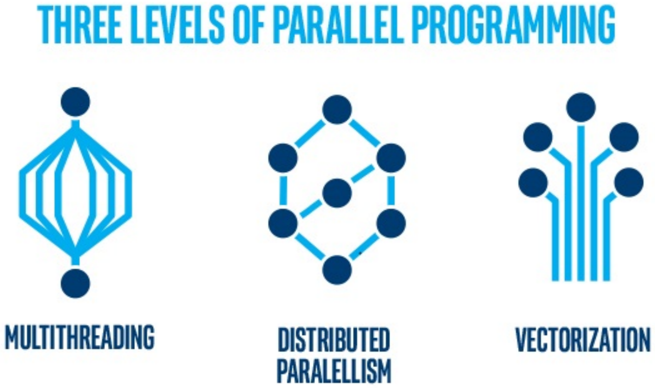
You have to find a trade-off between development time and runtime. Make sure the development of a speed-up does not cost you more time/resources than it saves you in the end!

Parallel Programming Models

Recap: Computer Architectures - Flynn's taxonomy



Parallel Programming Models - Overview



Parallel Programming Models - SIMD / Vectorization

- special case of *automatic* parallelization

Parallel Programming Models - SIMD / Vectorization

- special case of *automatic* parallelization
- special hardware allowing to run the same operation on multiple operands ('vector') at once

Parallel Programming Models - SIMD / Vectorization

- special case of *automatic* parallelization
- special hardware allowing to run the same operation on multiple operands ('vector') at once
- all modern CPUs support such operations (e.g. MMX/SSE/AVX)

Parallel Programming Models - SIMD / Vectorization

- special case of *automatic* parallelization
- special hardware allowing to run the same operation on multiple operands ('vector') at once
- all modern CPUs support such operations (e.g. MMX/SSE/AVX)
- used e.g. for loops

Parallel Programming Models - SIMD / Vectorization

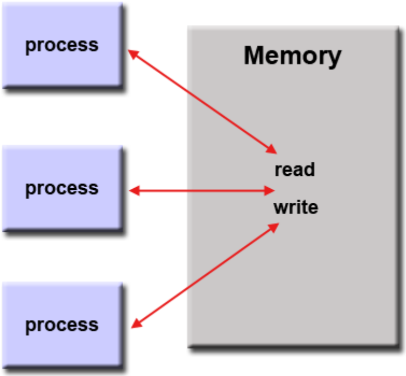
- special case of *automatic* parallelization
- special hardware allowing to run the same operation on multiple operands ('vector') at once
- all modern CPUs support such operations (e.g. MMX/SSE/AVX)
- used e.g. for loops

Scalar version	Vectorized version

<pre>int A[], B[], C[];
...
for(i=0; i<n; i++) {
  a = A[i];
  b = B[i];
  c = a+b;
  C[i] = c;
}</pre>
	<pre>int A[], B[], C[];
	...
	/* vectorized loop */
	for(i=0; i<n; i+=vf) {
	  va = A[i..i+vf[;
	  vb = B[i..i+vf[;
	  vc = padd(va, vb);
	  C[i..i+vf[= vc;
	}
	/* epilogue */
	for(; i<n; i++) {
	  /* remaining iterations */
	}</pre>

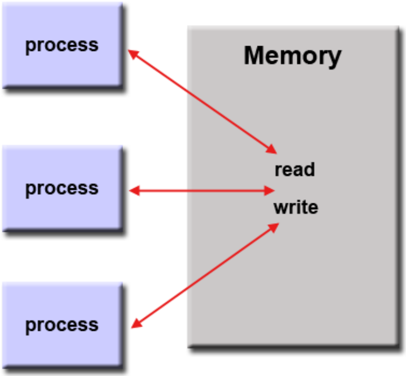
Parallel Programming Models - Shared memory without Threads

- simplest parallel programming model



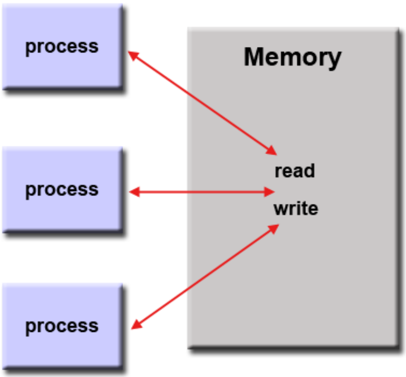
Parallel Programming Models - Shared memory without Threads

- simplest parallel programming model
- processes share common address space



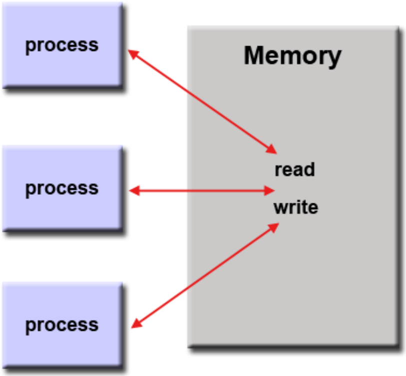
Parallel Programming Models - Shared memory without Threads

- simplest parallel programming model
- processes share common address space
- access to shared memory has to be controlled to prevent race conditions and deadlocks (see later)



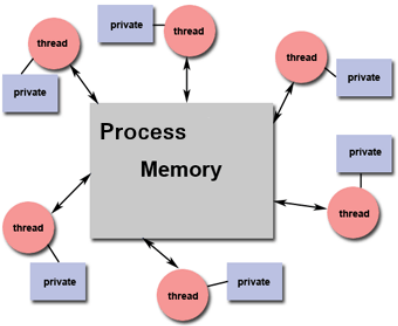
Parallel Programming Models - Shared memory without Threads

- simplest parallel programming model
- processes share common address space
- access to shared memory has to be controlled to prevent race conditions and deadlocks (see later)
- while not very common in use, e.g. POSIX standards provide API, UNIX provides shared memory segments



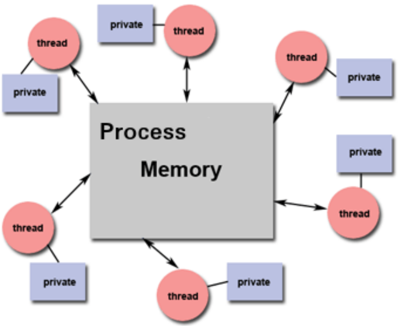
Parallel Programming Models - Shared memory with Multithreading

- In Multithreading, a single "heavy weight" process can have multiple "light weight", concurrent execution paths (threads).



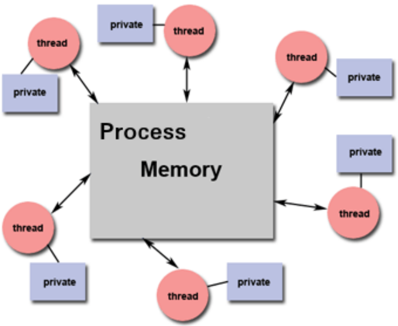
Parallel Programming Models - Shared memory with Multithreading

- In Multithreading, a single "heavy weight" process can have multiple "light weight", concurrent execution paths (threads).
- A thread's work may best be described as a subroutine within the main program.



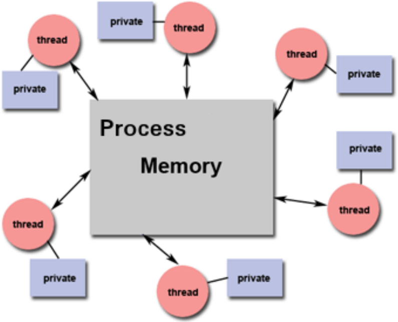
Parallel Programming Models - Shared memory with Multithreading

- In Multithreading, a single "heavy weight" process can have multiple "light weight", concurrent execution paths (threads).
- A thread's work may best be described as a subroutine within the main program.
- Any thread can execute any subroutine at the same time as other threads.



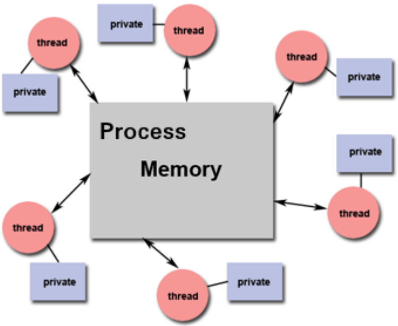
Parallel Programming Models - Shared memory with Multithreading

- In Multithreading, a single "heavy weight" process can have multiple "light weight", concurrent execution paths (threads).
- A thread's work may best be described as a subroutine within the main program.
- Any thread can execute any subroutine at the same time as other threads.
- Each thread has local data, but also, shares the entire resources of its parent process i.e. saves replicating a program's resources for each thread ("light weight").



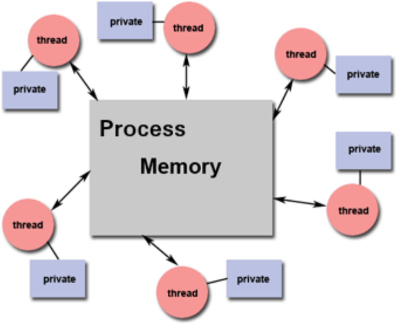
Parallel Programming Models - Shared memory with Multithreading (cont.)

- Each thread also benefits from global memory view; it shares memory space of the host process (requires **access control**!)



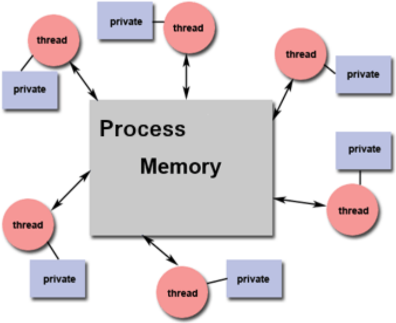
Parallel Programming Models - Shared memory with Multithreading (cont.)

- Each thread also benefits from global memory view; it shares memory space of the host process (requires **access control**!)
- Threads communicate with each other through global memory.



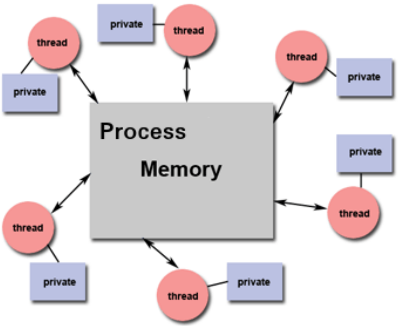
Parallel Programming Models - Shared memory with Multithreading (cont.)

- Each thread also benefits from global memory view; it shares memory space of the host process (requires **access control**!)
- Threads communicate with each other through global memory.
- Threads can be spawned and terminated at any time in the host process, but the host process remains present to provide the necessary shared resources until the application has completed.

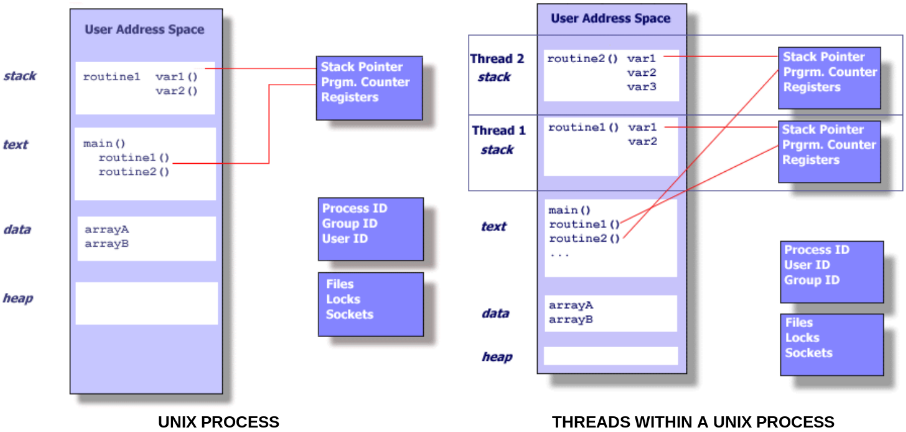


Parallel Programming Models - Shared memory with Multithreading (cont.)

- Each thread also benefits from global memory view; it shares memory space of the host process (requires **access control**!)
- Threads communicate with each other through global memory.
- Threads can be spawned and terminated at any time in the host process, but the host process remains present to provide the necessary shared resources until the application has completed.
- We will focus on two standards: **OpenMP** & **POSIX Threads**

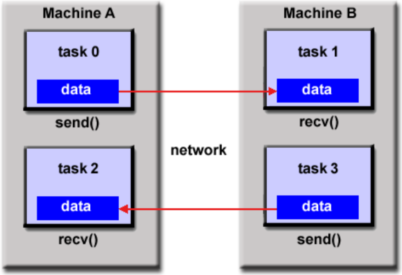


Parallel Programming Models - Shared memory with Multithreading (cont.)



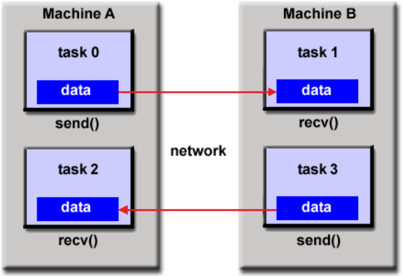
Distributed Parallelism with Message Passing

- set of processes with own local memory - reside on the same node and/or across multiple nodes.



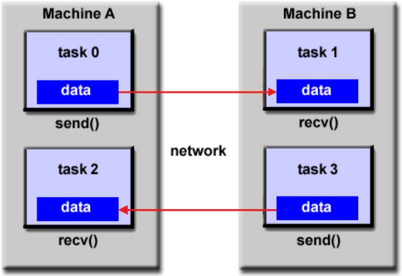
Distributed Parallelism with Message Passing

- set of processes with own local memory - reside on the same node and/or across multiple nodes.
- Data between processes is exchanged through sending and receiving messages ("Message Passing")



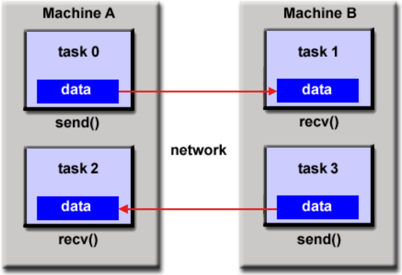
Distributed Parallelism with Message Passing

- set of processes with own local memory - reside on the same node and/or across multiple nodes.
- Data between processes is exchanged through sending and receiving messages ("Message Passing")
- The message passing requires cooperation between processes (each send must have a matching receive operation)



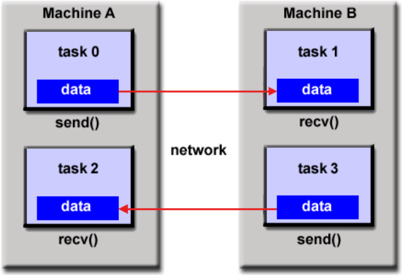
Distributed Parallelism with Message Passing

- set of processes with own local memory - reside on the same node and/or across multiple nodes.
- Data between processes is exchanged through sending and receiving messages ("Message Passing")
- The message passing requires cooperation between processes (each send must have a matching receive operation)
- The message system can also be used to synchronize processes



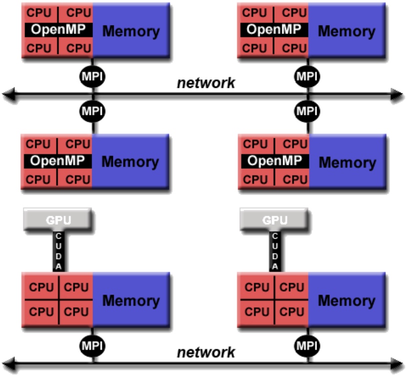
Distributed Parallelism with Message Passing

- set of processes with own local memory - reside on the same node and/or across multiple nodes.
- Data between processes is exchanged through sending and receiving messages ("Message Passing")
- The message passing requires cooperation between processes (each send must have a matching receive operation)
- The message system can also be used to synchronize processes
- *MPI* is the "de facto" standard



Hybrid Models

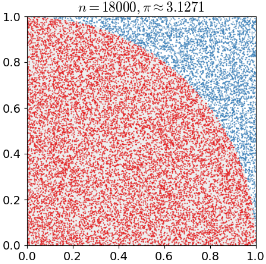
- Allows to make best use of locally shared memory or hardware, while still allowing for a good scalability across multiple nodes
- Comes with a significant increase in complexity/costs
- certain incompatibilities between libraries may exist (e.g. lack of thread-safety of MPI library)



Designing Parallel Programs

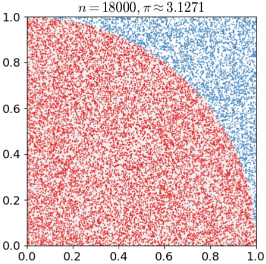
Understand your problem (!)

- some problems can be easily parallelized e.g. calculation of π by Monte-Carlo sampling



Understand your problem (!)

- some problems can be easily parallelized e.g. calculation of π by Monte-Carlo sampling

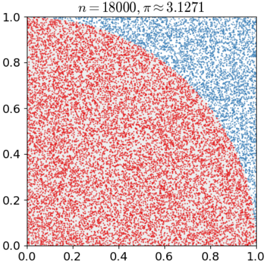


- others allow for little-to-no parallelism e.g. recursive calculation of Fibonacci series

$$F(n) = F(n-1) + F(n-2),\ F(0) = 0,\ F(1) = 1$$

Understand your problem (!)

- some problems can be easily parallelized e.g. calculation of π by Monte-Carlo sampling



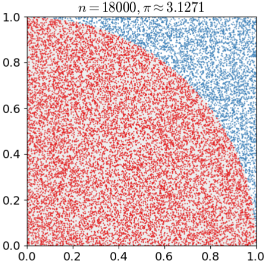
- others allow for little-to-no parallelism e.g. recursive calculation of Fibonacci series

$$F(n) = F(n-1) + F(n-2),\ F(0) = 0,\ F(1) = 1$$

- Identify inhibitors to parallelism: data-dependencies, I/O bottlenecks

Understand your problem (!)

- some problems can be easily parallelized e.g. calculation of π by Monte-Carlo sampling



- others allow for little-to-no parallelism e.g. recursive calculation of Fibonacci series

$$F(n) = F(n-1) + F(n-2),\ F(0) = 0,\ F(1) = 1$$

- Identify inhibitors to parallelism: data-dependencies, I/O bottlenecks
- Consider replacing your algorithms with equivalent ones better suited for parallelism

Understand your tools/program (!)

- pick optimal parallel programming model for your infrastructure

Understand your tools/program (!)

- pick optimal parallel programming model for your infrastructure
- make use of hardware optimization (e.g. vectorization, optimized libraries like MKL)

Understand your tools/program (!)

- pick optimal parallel programming model for your infrastructure
- make use of hardware optimization (e.g. vectorization, optimized libraries like MKL)
- identify hotspots in your program, i.e. routines where program spends lots of time in and check for improvement in parallelism ($\rightarrow$Amdahl's Law/Scaling)

Partitioning - Domain decomposition

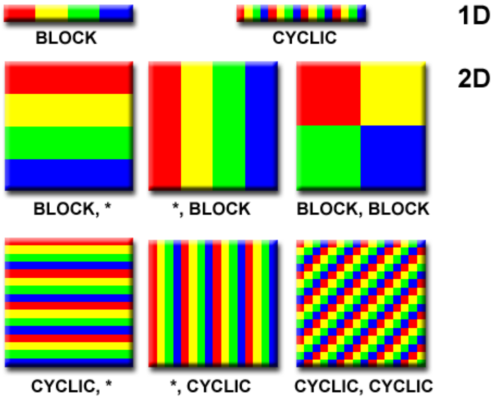
- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.

Partitioning - Domain decomposition

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.
- There are different ways to partition data:

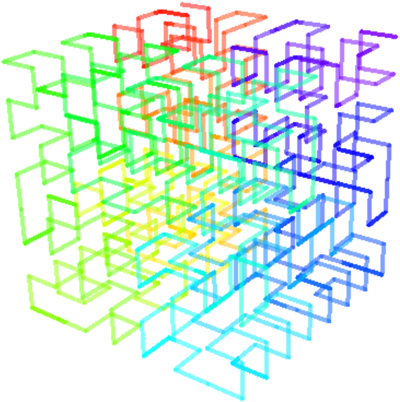
Partitioning - Domain decomposition

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.
- There are different ways to partition data:



Partitioning - Domain decomposition

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.
- There are different ways to partition data:

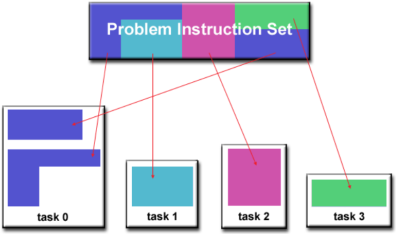


Partitioning - functional decomposition

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation.

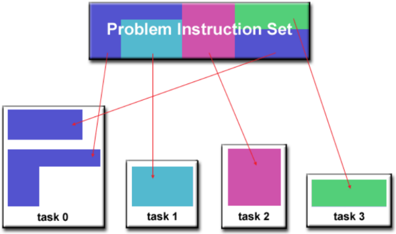
Partitioning - functional decomposition

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- Tasks/threads then specialize in doing specific parts of the overall work:



Partitioning - functional decomposition

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- Tasks/threads then specialize in doing specific parts of the overall work:



- implemented e.g. in master/slave paradigm (see exercises)

Data Dependence

- *data dependence* results from multiple use of the same memory location by different tasks.

Data Dependence

- *data dependence* results from multiple use of the same memory location by different tasks.
- Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism (cf. Fibonacci series).

Data Dependence

- *data dependence* results from multiple use of the same memory location by different tasks.
- Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism (cf. Fibonacci series).
- This can also cause a so called race condition:

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Synchronization

- synchronization is used to control the flow of processes/threads to deal with "mutually exclusive" resources (using locks/semaphores) to resolve e.g. race conditions

Synchronization

- synchronization is used to control the flow of processes/threads to deal with "mutually exclusive" resources (using locks/semaphores) to resolve e.g. race conditions
- ... or to synchronize the calculations of processes (using barriers) for communication to exchange results or to redistribute the workload

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.
- Factors to consider:

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.
- Factors to consider:
  - Communication overhead

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.
- Factors to consider:
  - Communication overhead
  - Latency vs. Bandwidth

Communication

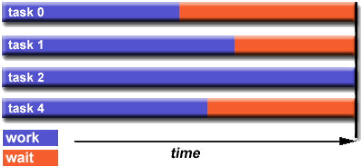
- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.
- Factors to consider:
  - Communication overhead
  - Latency vs. Bandwidth
  - Synchronous vs. asynchronous communications

Communication

- Depending on the type of problem and used decomposition approach, more or less communication may be required
- e.g. for *embarrassingly parallel* problem tasks run (mostly) independent of each other while e.g. in cosmological simulations long-range forces have to communicated between tasks working on neighbouring domains and particles may have to be exchanged if domain borders or particles move.
- Factors to consider:
  - Communication overhead
  - Latency vs. Bandwidth
  - Synchronous vs. asynchronous communications
  - Point-to-Point vs. collective communications

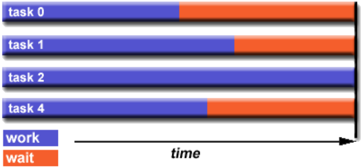
Load balancing

- Load balancing needed to ensure that processors are optimally i.e. minimizing idle times at synchronization points



Load balancing

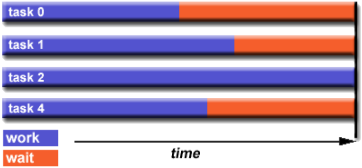
- Load balancing needed to ensure that processors are optimally i.e. minimizing idle times at synchronization points



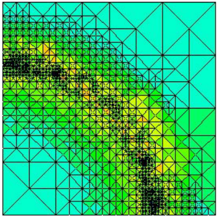
- requires well-balanced workload distribution between processors

Load balancing

- Load balancing needed to ensure that processors are optimally i.e. minimizing idle times at synchronization points



- requires well-balanced workload distribution between processors
- difficult in heterogeneous, dynamic problem sets with incomplete information about the actual workload

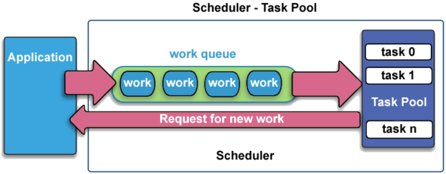


Load balancing (cont.)

- requires repeated repartitioning the problem based on estimate of workload

Load balancing (cont.)

- requires repeated repartitioning the problem based on estimate of workload
- alternatively, use asynchronous approach with scheduler-task pool with smaller workload packages



I/O

- data transfer to storage medias, network file servers

I/O

- data transfer to storage medias, network file servers
- I/O operations are generally obstacles for parallelism as they require orders of magnitude more time than memory operations.

I/O

- data transfer to storage medias, network file servers
- I/O operations are generally obstacles for parallelism as they require orders of magnitude more time than memory operations.
- Like for shared memory, may require synchronization (e.g. to avoid two processes (over)writing the same file simultaneously in a shared file system

I/O

- data transfer to storage medias, network file servers
- I/O operations are generally obstacles for parallelism as they require orders of magnitude more time than memory operations.
- Like for shared memory, may require synchronization (e.g. to avoid two processes (over)writing the same file simultaneously in a shared file system
- Parallel filesystems available (e.g. GPFS, Lustre, HDFS)

I/O

- data transfer to storage medias, network file servers
- I/O operations are generally obstacles for parallelism as they require orders of magnitude more time than memory operations.
- Like for shared memory, may require synchronization (e.g. to avoid two processes (over)writing the same file simultaneously in a shared file system
- Parallel filesystems available (e.g. GPFS, Lustre, HDFS)
- more on this tomorrow!

Performance Analysis & Tuning

- Analyzing and tuning parallel program performance can be much more challenging than for serial programs as interactions between tasks result in very complex dynamics
- Unfortunately, covering this topic in any detail would go beyond the scope of this introduction to parallel program.
- There are a number of excellent tools for this task: e.g. Intel VTune Amplifier and Intel Trace Analyzer