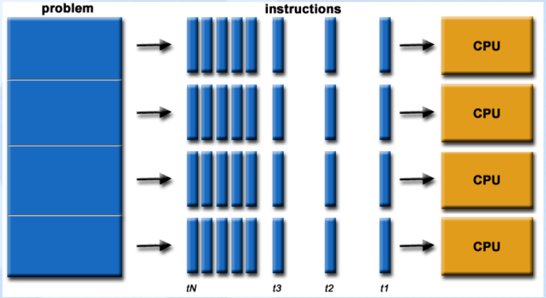
CONCURRENCY

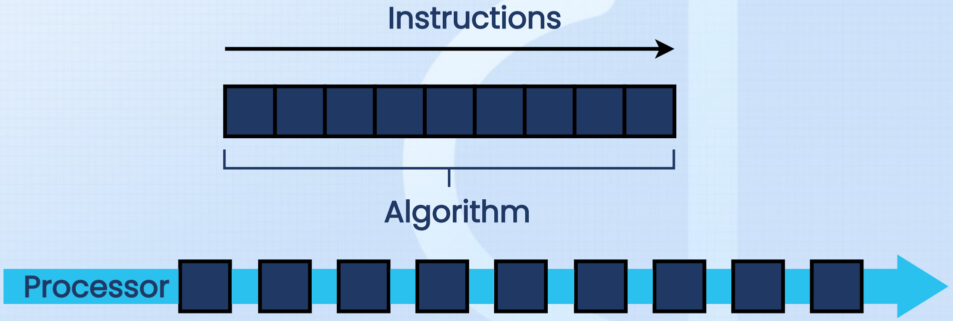
The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently.

- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control / coordination mechanism is employed



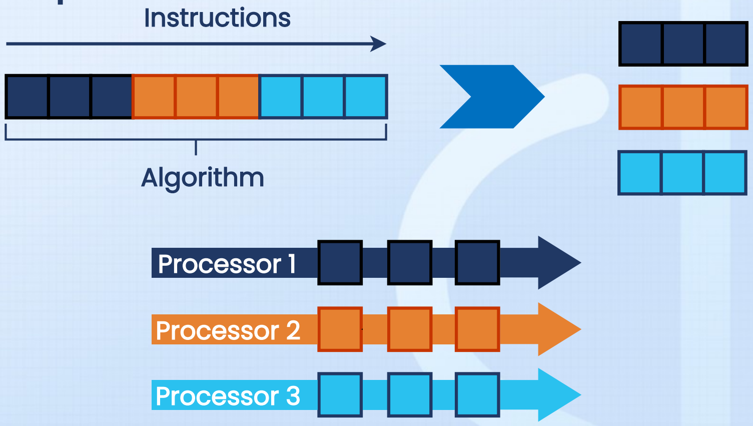
PARALLEL PROGRAMMING

Serial problem



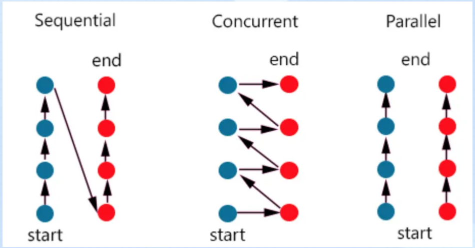
PARALLEL PROGRAMMING

Parallel problem

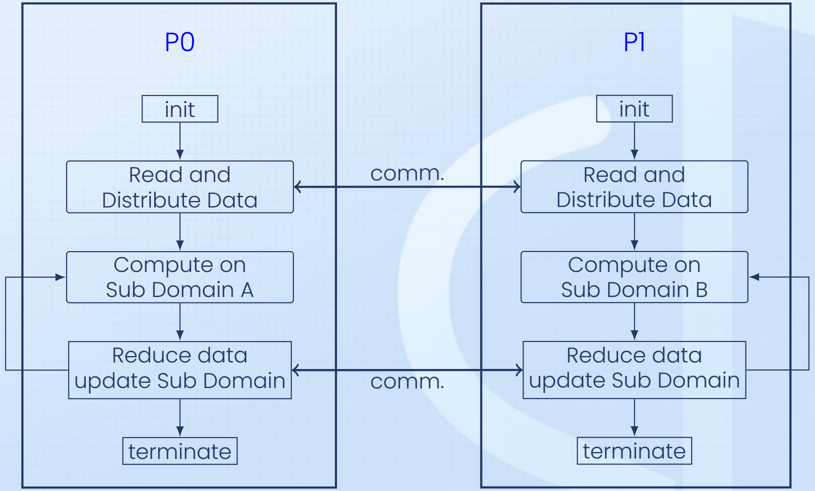


CONCURRENCY VS. PARALLELISM

- Concurrency is when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean that they will ever be running at the same instant of time. Example: Multitasking on a single-threaded machine.
- Parallelism is when two tasks literally run at the same time, e.g., on a multi-core computer.
- Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once. – Rob Pike



EXAMPLE OF PARALLEL PROGRAM

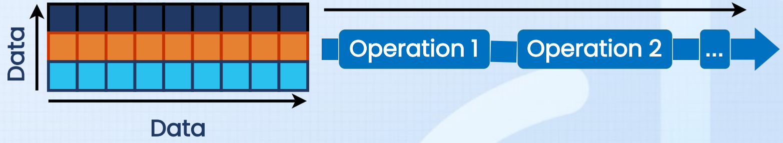


FUNDAMENTAL STEPS OF PARALLEL DESIGN

- Identify portions of the work that can be performed concurrently.
  `=> Determine which tasks or subproblems within the program can be executed independently of one another.`
- Distribute the input, output, and intermediate data.
  `=> Allocate the necessary data (inputs, outputs, ...) among the different processors to minimize duplication and prevent access conflicts.`
- Map the concurrent tasks and their data onto multiple processors.
  `=> Assign each independent portion of work - along with its corresponding data - to one or more processes running in parallel.`
- Manage accesses to shared data.
  `=> Implement synchronization mechanisms to ensure consistency and avoid race conditions when multiple processors access shared data.`
- Synchronize the processors at various stages of execution.
  `=> Coordinate the progress of parallel tasks so that specific stages start only after others have completed, ensuring program correctness and consistency.`

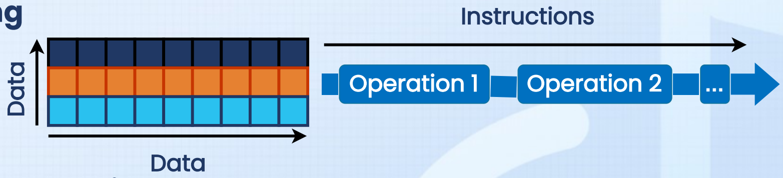
PARALLEL STRATEGIES

Setting

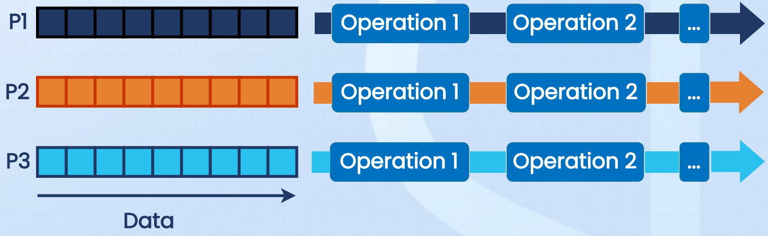


PARALLEL STRATEGIES

Setting

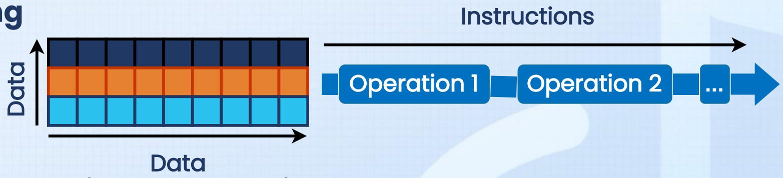


Data parallelism

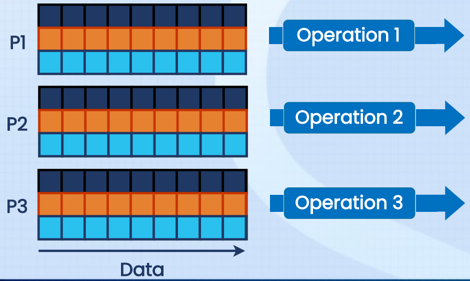


PARALLEL STRATEGIES

Setting

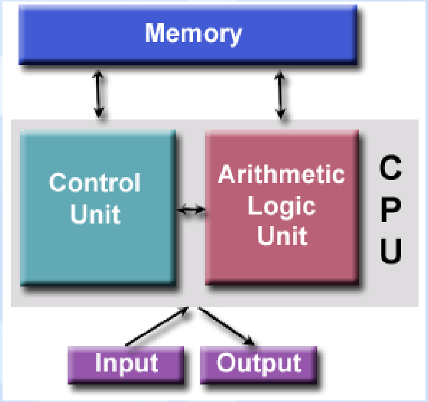


Task/Functional parallelism

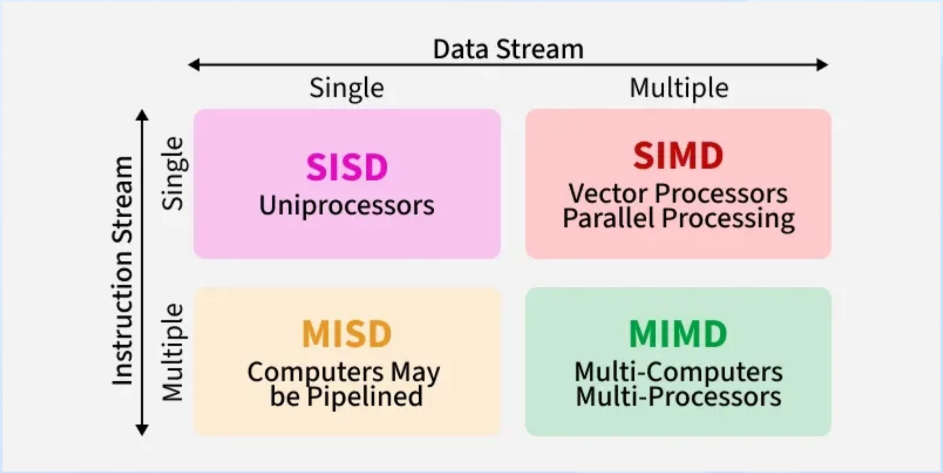


VON NEUMANN COMPUTER ARCHITECTURE

- Named after the Hungarian mathematician John von Neumann who first authored the general requirements for an electronic computer in his 1945 papers.
- Also known as the stored-program computer — both program instructions and data are kept in electronic memory. Differs from earlier computers which were programmed through “hard wiring”.
- Since then, virtually all computers have followed this basic design.
- Comprised of four main components.



FLYNN's CLASSIFICATION OF COMPUTERS

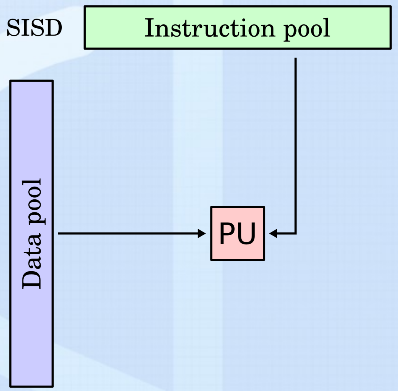


FLYNN'S CLASSIFICATION OF COMPUTERS

SISD: Single Instruction, Single Data

A serial (non-parallel) computer.

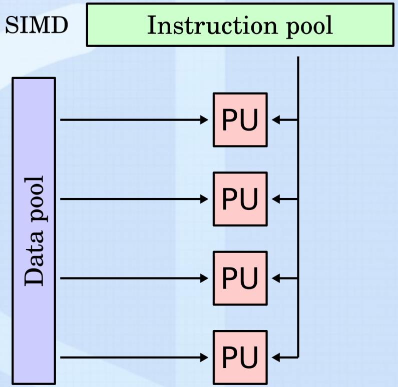
- Single Instruction: Only one instruction stream is acted on by the CPU in one clock cycle.
- Single Data: Only one data stream is used as input during any clock cycle.
- Examples: Uni-core PCs, single CPU workstations, and mainframes.



FLYNN'S CLASSIFICATION OF COMPUTERS

SIMD: Single Instruction, Multiple Data

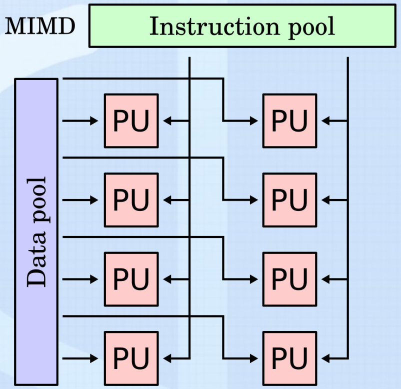
- Single Instruction: All processing units execute the same instruction at any given clock cycle.
- Multiple Data: Each processing unit can operate on a different data element.
- Best suited for problems of high degree of regularity, such as image processing.
- Examples: Connection Machine CM-2, Cray C90, NVIDIA GPUs.



FLYNN’S CLASSIFICATION OF COMPUTERS

MIMD: Multiple Instruction, Multiple Data

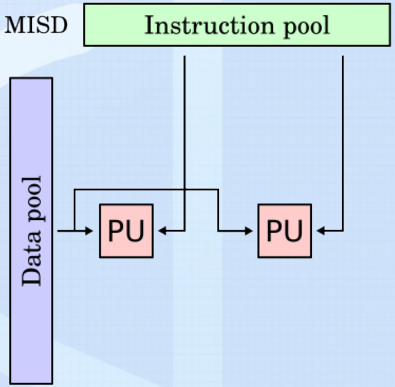
- Multiple Instruction: Every processing unit may be executing a different instruction at any given clock cycle.
- Multiple Data: Each processing unit can operate on a different data element.
- Examples: Most modern computers fall into this category.



FLYNN'S CLASSIFICATION OF COMPUTERS

MISD: Multiple Instruction, Single Data

- Multiple Instruction: Each processing unit operates on the data independently via an independent instruction stream.
- Single Data: A single data stream is fed into multiple processing units.
- Examples: Few actual examples of this category have ever existed. Typically found in redundant fault-tolerant computing systems, such as those used in NASA space shuttles.



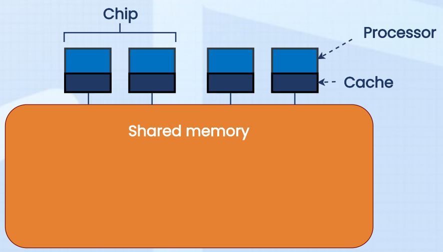
PARALLEL CLASSIFICATION

- SIMD (Single Instruction, Multiple Data):
  - Synchronized execution of the same instruction on a set of processors.
  - Used in vector processing, GPUs, and data-parallel architectures.
- MIMD (Multiple Instruction, Multiple Data):
  - Asynchronous execution of different instructions on multiple processors.
  - Used in modern multi-core CPUs, clusters, and HPC systems.

PARALLEL COMPUTER MEMORY ARCHITECTURES

Shared Memory

- Multiple processors operate independently but access a global shared memory space.
- Changes in one memory location are visible to all processors.
- Synchronization mechanisms (locks, semaphores) are required to manage concurrent access.



Advantages:

- Global address space provides a user-friendly programming perspective to memory.
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs.

Disadvantages:

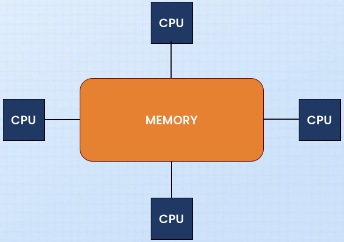
- Expensive for large systems due to memory access bottlenecks.
- Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.

PARALLEL COMPUTER MEMORY ARCHITECTURES

Shared Memory: UMA vs NUMA

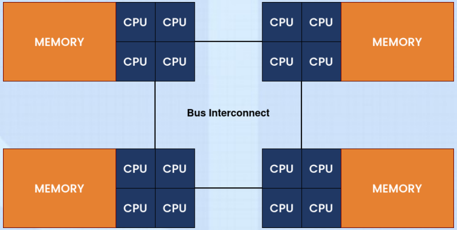
UMA (Uniform Memory Access)

- Also known as Symmetric Multi-Processors (SMP).
- Centralized shared memory.
- Accesses to global memory from all processors have the same latency.



NUMA (Non-Uniform Memory Access)

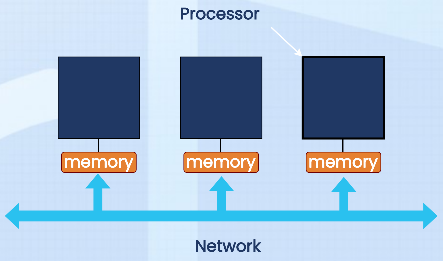
- Also known as Distributed Shared Memory (DSM).
- Memory is distributed among the nodes.
- Local accesses are much faster than remote accesses.



PARALLEL COMPUTER MEMORY ARCHITECTURES

Distributed Memory

- Processors have local (non-shared) memory.
- Requires a communication network for data exchange.
- Data from one processor must be explicitly communicated if required.
- Synchronization is the programmer's responsibility.



Advantages:

- Highly scalable for large systems.
- No cache coherence problems compared to shared memory.
- Can use commodity processors to build large clusters.

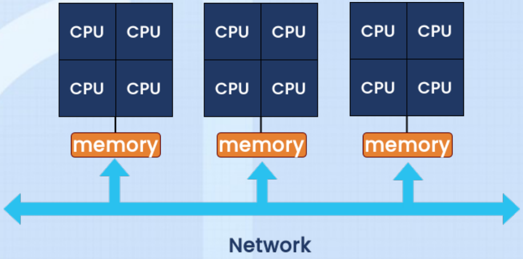
Disadvantages:

- Increased programmer responsibility for communication and synchronization.
- Non-uniform memory access times can vary, impacting performance.

PARALLEL COMPUTER MEMORY ARCHITECTURES

Hybrid Memory Model

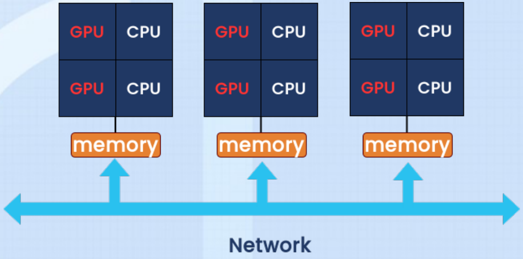
- Today's biggest machines incorporate both shared and distributed memory architectures.
- They leverage the speed and simplicity of shared memory alongside the scalability of distributed memory.
- Advantages and disadvantages are those of the individual parts!



PARALLEL COMPUTER MEMORY ARCHITECTURES

Hybrid Memory Model

- Today's biggest machines incorporate both shared and distributed memory architectures.
- They leverage the speed and simplicity of shared memory alongside the scalability of distributed memory.
- Advantages and disadvantages are those of the individual parts!



PARALLEL PROGRAMMING MODELS

Parallel Programming Models

There are several parallel programming models in common use:

- Threads Model: Pthreads, OpenMP, CUDA Threads
- Distributed Memory / Message Passing: MPI
- Hybrid: MPI + OpenMP, MPI + GPU
- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

PARALLEL PROGRAMMING MODELS

SPMD: Single Program, Multiple Data

A parallel programming model where multiple processors execute the same program but operate on different data.

- All processes run the same program, but each has a unique rank and works on different portions of data.
- Common in message passing or hybrid programming.

PARALLEL PROGRAMMING MODELS

MPMD: Multiple Program, Multiple Data

A parallel programming model where multiple processors execute different programs on different data.

- Each processor or process runs a different program, often designed to perform specific subtasks.
- Used in heterogeneous computing, where different processors handle specialized workloads.

Example: To execute an MPI program in MPMD mode, use the `--multi-prog` option in the submission script along with a configuration file (`mpmd.conf`).

```
srun --multi -prog ./ mpmd.conf
```

```
cat mpmd.conf
0
./a.out
1-3,7 ./b.out
4-6
./c.out
```

PARALLEL TOOLS

Common Parallel Programming Tools

- MPI (Message Passing Interface): Standard for distributed computing.
- OpenMP: Shared memory parallel programming.
- CUDA: GPU programming for parallel execution.
- OpenCL: Cross-platform parallel programming.
- MATLAB Parallel Computing Toolbox: High-level parallelization.

ENERGY EFFICIENCY IN PARALLEL COMPUTING

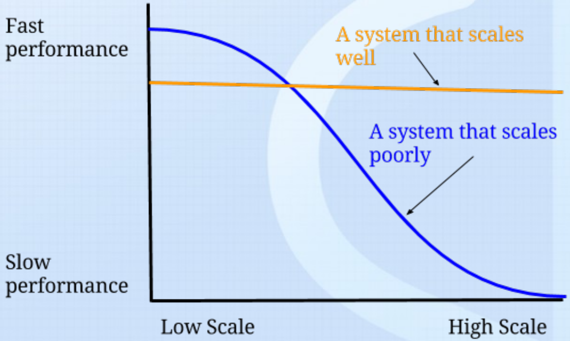
Optimizing Performance and Power

- Power Consumption: Measured in watts, affects energy efficiency.
- Dynamic Voltage and Frequency Scaling (DVFS): Adjusting CPU frequency to balance performance and energy.
- CPU Governors: Strategies for controlling CPU power consumption (e.g., `ondemand`, `powersave`).

PERFORMANCE AND SCALABILITY

Design of parallel applications:

- Performance
- Scalability



CLASSES OF PERFORMANCE METRICS

Distinct classes of performance metrics:

- Performance metrics for processors/cores
- Performance metrics for parallel applications
  - Practical metrics
  - Theoretical metrics

EXECUTION TIME

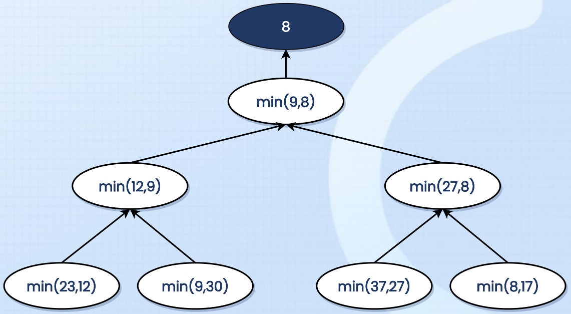
Finding minimum element among [23,12,9,30,37,27,8,17].

- Execution time:
    - Serial Time T_S $(\theta(n))$

EXECUTION TIME

Finding minimum element among [23,12,9,30,37,27,8,17].

- Execution time:
  - Serial Time T_S ($\theta(n)$)
  - Parallel Time T_P ($\theta(\log n)$)



TOTAL PARALLEL OVERHEAD

Finding minimum element among [23,12,9,30,37,27,8,17].

- Execution time:
  - Serial Time T_s $(\theta(n))$
  - Parallel Time T_p $(\theta(\log n))$

Total Parallel Overhead

- With p processes:
  - Total time = pT_p
  - Overhead = $pT_p - T_s$

Performance Metrics for Parallel Applications

- Some of the best known metrics are:
  - Speedup
  - Efficiency
- There also some laws/metrics that try to explain and assert the potential performance of a parallel application. The best known are:
  - Amdahl's law
  - Gustafson-Barsis' law

SPEEDUP

Speedup is a measure of performance.

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

- Example 1: Find out the minimum element in array

$$S_p = \frac{\theta(n)}{\theta(\log n)} = \theta\left(\frac{n}{\log n}\right)$$

- Example 2: Solve 1D transport equation

	1 CPUs	2 CPUs	4 CPUs	8 CPUs	16 CPUs
T(p)	1000	520	280	160	100
S(p)	1	1.92	3.57	6.25	10.00

EFFICIENCY

Efficiency is a measure of the usage of the computational capacity.

$$E_p = \frac{S_p}{p} = \frac{T_s}{p \times T_p}$$

- Example 1: Find out the minimum element in array

$$E_p = \frac{\theta\left(\frac{n}{\log n}\right)}{p} \quad (\text{if } p = n) => E_p = \frac{\theta\left(\frac{n}{\log n}\right)}{n} = \theta\left(\frac{1}{\log n}\right)$$

- Example 2: Solve 1D transport equation

	1 CPUs	2 CPUs	4 CPUs	8 CPUs	16 CPUs
S(p)	1	1.92	3.57	6.25	10.00
E(p)	1	0.96	0.89	0.78	0.63

AMDAHL'S LAW: STRONG SCALING

Definition: Strong scaling is defined as how the solution time varies with the number of processors for a fixed total problem size.

- Serial part: $0 \leq f \leq 1$

$$S_p = \frac{1}{f + \frac{1-f}{p}}$$

- Example: If f = 10%, what is the max. speedup achievable using 8 processors?
  - Solution:

$$S_p = \frac{1}{0.1 + \frac{1-0.1}{8}} \approx 4.7$$

- Limit:

$$\lim_{p \to \infty} = \frac{1}{0.1 + \frac{1-0.1}{p}} = 10$$

Disadvantage:

- Amdahl's law ignores the costs with communication/synchronization operations associated with the parallel version of the problem. For that reason, it can result in predictions not very realistic for certain problems.

GUSTAFSON'S LAW: WEAK SCALING

Definition: Weak scaling is defined as how the solution time varies with the number of processors for a fixed problem size per processor.

- Parallel part: $0 \leq f' \leq 1$

$$S_p = (1 - f') + f' \times p = 1 + (p - 1) \times f'$$

- Example: $f' = 90\%$, what is the scaled speedup using 64 processors?
- Solution:

$$S_p = 1 + (p - 1) \times f' = 1 + (64 - 1) \times 0.90 = 57.70$$

Disadvantage:

- By using the parallel execution time as the starting point, instead of the sequential execution time, the Gustafson-Barsis law assumes that the execution time with one processing unit is, in the worst case, p times slower than the execution with p processing units.

SCALABLE APPLICATION

Scalable application

- Strong scaling + Weak scaling

		1 CPUs	2 CPUs	4 CPUs	8 CPUs	16 CPUs
Efficiency	n = 10 000	1	0.81	0.53	0.28	0.16
	n = 20 000	1	0.94	0.80	0.59	0.42
	n = 40 000	1	0.96	0.89	0.79	0.58

SCALABLE APPLICATION

Process interactions

- The effective speed-up obtained by the parallelization depends on the amount of overhead introduced when making the algorithm parallel.
- There are mainly two key sources of overhead:
  1. Time spent in inter-process interactions (communication)
  2. Time some process may spend being idle (synchronization)

Example:

- Communication between processes can be compared to people exchanging information or objects in a coordinated system.
- Each person (process) works independently but must communicate with others to complete a shared goal.

SCALABLE APPLICATION

Load balancing

- Equally divide the work among the available resources: processors, memory, network bandwidth, I/O, ...
- This is usually a simple task for the problem decomposition model.
- It is a difficult task for the functional decomposition model.

SCALABLE APPLICATION

Minimizing communication

- When possible, reduce the communication events:
  - Group lots of small communications into large ones.
  - Eliminate synchronizations as much as possible. Each synchronization levels off the performance to that of the slowest process.

SCALABLE APPLICATION

Distributed data vs replicated data

- Replicated data distribution is useful if it helps to reduce the communication among processes at the cost of bounding scalability.
- Distributed data is the ideal data distribution but not always applicable for all data-sets.
- Usually complex applications are a mix of those techniques $\Rightarrow$ distribute large data sets; replicate small data.

SCALABLE APPLICATION

Overlap communication and computation

- When possible, code your program in such a way that processes continue to do useful work while communicating.
- This is usually a non-trivial task and is afforded in the very last phase of parallelization.
- If you succeed, you have done. Benefits are enormous.

Factors of Superlinear Speedup

$$\frac{T_s}{T_p} \geq p$$

- Nonexistent initialization, communication and/or synchronization costs
- Communication latency
- Memory capacity
- Subdivision of the problem

PERFORMANCE ANALYSIS AND TUNING

- As with debugging, analyzing and tuning parallel program performance can be much more challenging than for serial programs.
- Fortunately, there are a number of excellent tools for parallel program performance analysis and tuning.

Some tools:

- TAU: `http://www.cs.uoregon.edu/research/tau/docs.php`
- HPCToolkit: `http://hpctoolkit.org/documentation.html`
- Vampir/Vampirtrace: `http://vampir.eu`
- Valgrind: `http://valgrind.org/`
- PAPI: `http://icl.cs.utk.edu/papi/`
- mpiP: `http://mpip.sourceforge.net/`
- memP: `http://memp.sourceforge.net/`