An Introduction to Parallel Programming

Marco Ferretti, Mirto Musci

Dipartimento di Informatica e Sistemistica
University of Pavia

Processor Architectures, Fall 2011

Definition

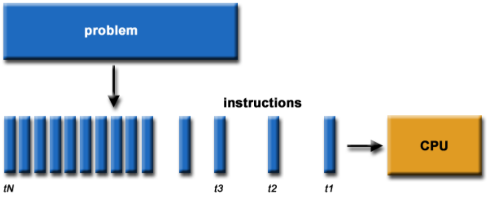
What is parallel programming?

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem

Outline - Serial Computing

Traditionally, software has been written for serial computation:

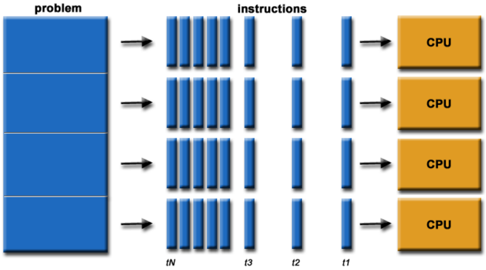
- To be run on a single CPU;
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment



Outline - Parallel Computing

Simultaneous use of multiple compute resources to solve a computational problem:

- To be run using multiple CPUs
- A problem is broken into discrete parts, solved concurrently
- Each part is broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



The Real World is Massively Parallel

Parallel computing attempts to emulate what have always been the state of affairs in the natural world: many complex, interrelated events happening at the same time, yet within a sequence

Why Parallel Computing?

- Save time and/or money
- Solve larger problems
- Provide concurrency (e.g. the Access Grid)
- Use of non-local resources (e.g. SETI@home)
- Limits to serial computing, physycal and practical
    - Transmission speeds
    - Miniaturization
    - Economic limitations

Application I

Traditional Applications

- High performance computing:
  - Atmosphere, Earth, Environment
  - Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
  - Bioscience, Biotechnology, Genetics
  - Chemistry, Molecular Sciences
  - Geology, Seismology
  - Mechanical Engineering modeling
  - Circuit Design, Computer Science, Mathematics

Application II

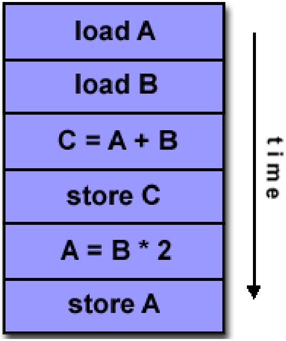
Emergent Applications

- Nowadays, industry is THE driving force:
    - Databases, data mining
    - Oil exploration
    - Web search engines, web based business services
    - Medical imaging and diagnosis
    - Pharmaceutical design
    - Financial and economic modeling
    - Advanced graphics and virtual reality
    - Collaborative work environments

Flynn's Classical Taxonomy

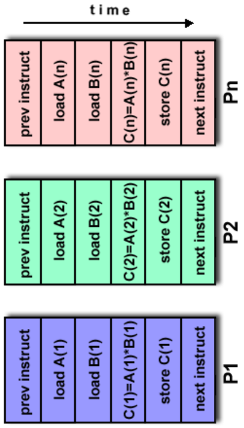
- There are different ways to classify parallel computers.
- Flynn's Taxonomy (1966) distinguishes multi-processor architecture by instruction and data:
    - SISD – Single Instruction, Single Data
    - SIMD – Single Instruction, Multiple Data
    - MISD – Multiple Instruction, Single Data
    - MIMD – Multiple Instruction, Multiple Data

Flynn's Classical Taxonomy - SISD



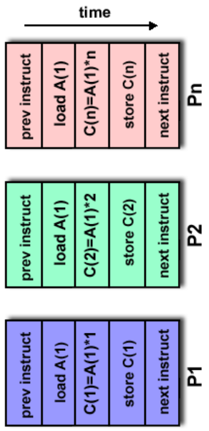
- Serial (non-parallel) computer
- Single Instruction: Only one instruction stream acted on by the CPU during a clock cycle
- Single Data: Only one data stream in input
- Deterministic execution

Flynn's Classical Taxonomy - SIMD



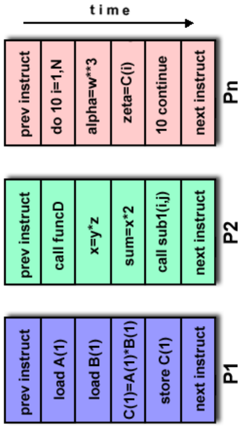
- Single Instruction: All CPUs execute the same instruction at a given clock cycle
- Multiple Data: Each CPU can operate on a different data element
- Problems characterized by high regularity, such as graphics processing
- Synchronous (lockstep) and deterministic execution
- Vector processors and GPU Computing

Flynn's Classical Taxonomy - MISD



- Multiple Instruction: Each CPU operates on the data independently
- Single Data: A single data stream for multiple CPU
- Very few practical uses for this type of classification.
- Example: Multiple cryptography algorithms attempting to crack a single coded message.

Flynn's Classical Taxonomy - MIMD



- Multiple Instruction: Every processor may be executing a different instruction stream
- Multiple Data: Every processor may be working with a different data stream
- Synchronous or asynchronous, deterministic or non-deterministic
- Examples: most supercomputers, networked clusters and "grids", multi-core PCs

Outline

Outline

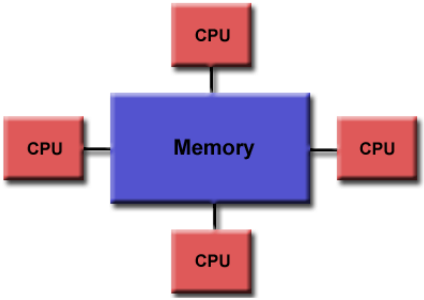
Shared Memory Architecture

- Shared memory: all processors access all memory as a global address space.
- CPUs operate independently but share memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Two main classes based upon memory access times: UMA and NUMA.

Unified Memory Access

Uniform Memory Access (UMA):

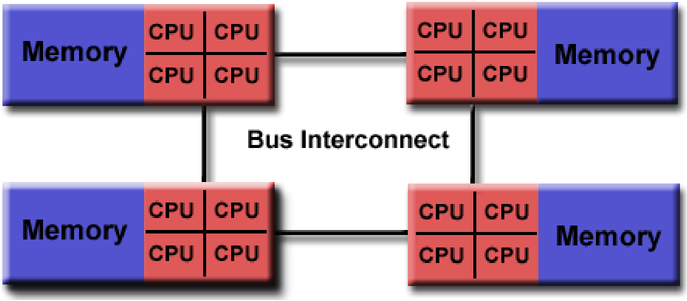
- Identical processors
- Equal access and access times to memory
- Cache coherent: if one processor updates a location in shared memory, all the other processors know about the update.



Non-Uniform Memory Access

Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower



Pros and Cons

Advantages

- Global address space is easy to program
- Data sharing is fast and uniform due to the proximity memory/CPUs

Disadvantages

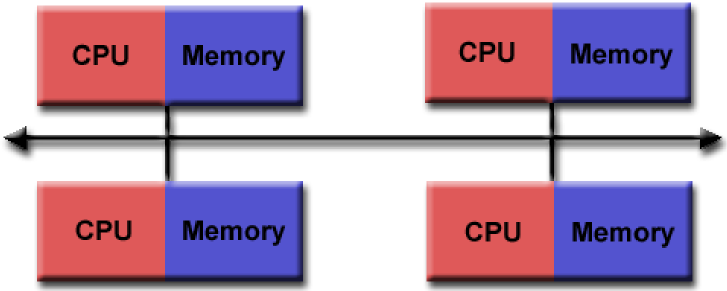
- Lack of scalability between memory and CPUs
- Programmer responsibility for synchronization

Outline

General Characteristics

- Communication network to connect inter-processor memory.
- Processors have their own local memory.
    - No concept of global address space
- CPUs operate independently.
    - Change to local memory have no effect on the memory of other processors.
    - Cache coherency does not apply.
- Data communication and synchronization are programmer's responsibility
- Connection used for data transfer varies (es. Ethernet)

Simple Diagram

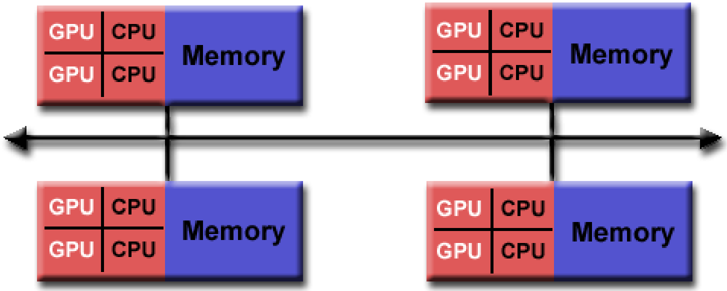


Outline

Hybrid Distributed-Shared Memory

- The largest and fastest computers in the world today employ hybrid architectures.
- The shared memory component can be a cache coherent SMP machine and/or graphics processing units (GPU).
- The distributed memory component is the networking of multiple SMP/GPU machines, which know only about their own memory.
- Network communications are required to move data from one SMP/GPU to another.

Simple Diagram



Overview

- Several parallel programming models in common use:
  - Shared Memory / Threads
  - Distributed Memory / Message Passing
  - Hybrid
  - GPGPU
  - ... and many more
- Parallel programming models exist as an abstraction above hardware and memory architectures.
- No "best" model, although there are better implementations of some models over others.

Overview II

Models are NOT specific to a particular machine or architecture.

- Shared model on a distributed memory machine: KSR1
  - Machine is physically distributed, but appear global
  - Virtual Shared Memory
- Distributed model on a shared memory machine: Origin 2000.
  - CC-NUMA architecture
  - Tasks see global address space, but use MP

Outline

Introduction

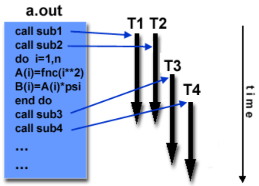
- Multiple processes or light threads with unified memory vision
- Need for careful synchronization
- Programmer is responsible for determining all parallelism.

Application Domain

- Number crunching on single, heavily parallel machines
- Desktop applications
- In Hpc or industrial application is seldom used by itself (clustering)
    - but Intel is developing Cluster OpenMP...

Threading Model

- Similar to a single program that includes a number of subroutines:
    - a.out performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run concurrently.
    - Each thread has local data, but also shares the entire resources of a.out
    - A thread's work could be seen as a subroutine
    - Threads communicate through global memory.
    - Threads come and go, but a.out remains



Implementations

Explicit approaches

- Linux: Pthreads library
  - Library based; requires parallel coding
  - C Language only
  - Very explicit parallelism
- Microsoft Windows Threading

Frameworks

- Intel TBB
- OpenMP
- Distributed shared memory OS (research topic)

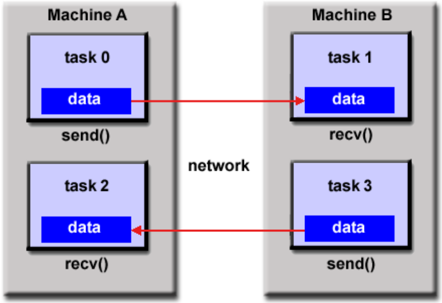
OpenMP

- It's not a library: series of compiler directives or pragmas
    - #pragma omp parallel ...
    - requires explicit compiler supports
- Features:
    - Automatic data layout and decomposition
    - Incremental parallelism: one portion of program at one time
    - Unified code for both serial and parallel applications
    - Coarse-grained and fine-grained parallelism possible
    - Scalability is limited by memory architecture

Outline

Distributed Memory / Message Passing Model

- Multiple tasks on arbitrary number of machines, each with local memory.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations



MPI: introduction

- Language independente API specification: allows processes to communicate with one another by sending and receiving messages
- De facto standard for HPC on clusters and supercomputers
- MPI was created in 1992; first standard appeared in 1994. Superseded PVM.
- Various implementations
  - OpenMPI: open source, widespread; faculties and accademic istitution; several *top500* supercomputers
  - commercial implementations from HP, Intel, and Microsoft (derived from the early MPICH)
  - custom implementations, parts in assembler or even in hardware (research topic)

MPI: features

- Very portable: language and architecture independent
- MPI hide communication complexities between distributed processes
    - virtual topology
    - synchronization
- Single process maps to single processor
- Need for an external agent (mpirun, mpiexec) to coordinate and manage task assignment and program termination
- Original (serial) application must be rewritten

MPI: functions

Library Functions

- Point-to-point communication
  - synchronous
  - asynchronous
  - buffered
  - ready forms
- Broadcast and multicast communication
- Fast and safe combination of partial results: gather and reduce
- Node synchronization: barrier

Outline

GPGPU: Introduction

- Using a GPU to perform computation in applications traditionally handled by the CPU
- Made possible by rendering pipeline modification:
  - addition of programmable stages
  - higher precision arithmetic
- Allows software developers to use stream processing on non-graphics data.
- Especially suitable for applications that exhibit:
  - Compute Intensity
  - Data Parallelism
  - Data Locality

Standards

CUDA and Stream

- Respectively NVDIA and AMD standards
- Framework for explicit GPU programming
- Language extension, library and runtime environments

- Common features:
  - Heavily threaded (ten of thousands)
  - Multi-level parallelism: Grid, Blocks, threads, weave...
  - Explicit memory copy to/from CPU
  - Linear Bidimensional memory access (texture memory)

Application domains

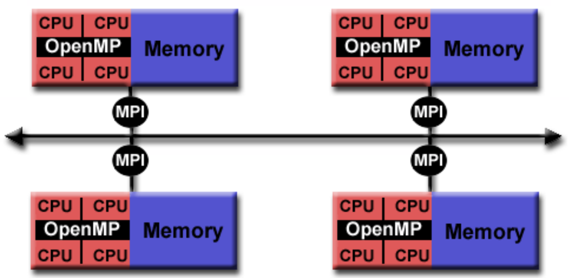
Where GPGPU has been used succesfully

- Raytracing Global illumination (not normally supported by the rendering pipeline)
- Physical based simulation and physics engines
  - Weather forecasting
  - Climate research
  - Molecular modeling, ...
- Computational finance, Medical imaging, Digital signal processing, Database operations, Cryptography and cryptanalysis, Fast Fourier transform, ...

Outline

Hybrid Model

- Combines the previously described programming models
- Well suited for cluster of SMP machines
- Example: Using MPI with GPU programming.
  - GPUs perform intensive kernels using local, on-node data
  - MPI handles communications



Implementations

- Prevalence of ad hoc solutions

New Standards

- Compiler Directives
  - OpenHMPP
  - OpenMP 3.x
- Programming Libraries: OpenCL

OpenCL

- Host-Device abstraction: handles any sort of parallel computing domain
  - CPU to CPU
  - CPU to GPU
  - CPU to DSP or any embedded device
- Need for explicit hardware support
  - NVIDIA, ATI, AMD support OpenCL
  - Unfortunately low embedded device support
- Similar to CUDA, but more generic and somewhat less programmer-friendly

Summary

- Parallel techologies are pervasive
- Lot of possible approaches
- Continue development and innovation

Future developement?

- Hybrid techologies are the key but: need for hardware support and programmers assistance
- GPU Computing will become common in desktop and embedded system

Outline

Automatic vs Manual Paralelization

- Automatic
    - Compiler analyzes code and identifies parallelism
    - Attempt to compute if actually improves performance
    - Loops are the most frequent target
- Manual - Understand the problem
    - Parallelizable Problem
        - Calculate the potential energy for several molecule's conformations. Find the minimum energy conformation.
    - A Non-Parallelizable Problem
        - The Fibonacci Series
        - All calculations are dependent

Keywords

Parallel Design Keywords

- Synchronization
- Communications
- Data Dependance
- Load Balancing
- Granularity
- Partitioning

Synchronization

Definition

The coordination of parallel tasks in real time

- Barrier
  - Each task performs its work until it stops at the barrier
  - When the last task arrives, all tasks are synchronized
  - What happens from here varies (serial work or task release)
- Lock / semaphore
  - Used to protect access to global data or a section of code.
  - Only one task at a time may “own” the lock
  - Other tasks can attempt to acquire the lock but must wait until release.

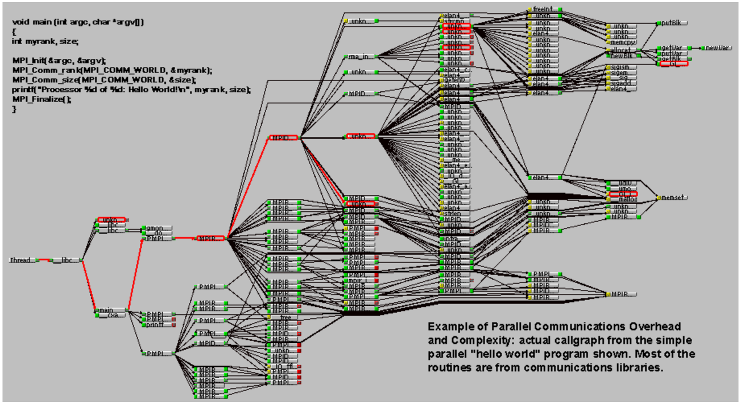
Communications

Definition

Data exchange between parallel tasks

- Who Needs Communications?
    - Embarrassingly parallel (e.g. operation on indipendent pixels)
    - Most problems are not that simple (e.g. 3D heat diffusion)
- Factors to Consider:
    - Cost of communications
    - Latency vs. Bandwidth
    - Visibility of communications
    - Synchronous vs. asynchronous communications
    - Scope of communications (point-to-point, collective)

Communications Complexity



```
void main (int argc, char *argv[])
{
int myrank, size;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("Processor %d of %d: Hello World!\n", myrank, size);
MPI_Finalize();
}
```

Example of Parallel Communications Overhead and Complexity: actual callgraph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.

Data Dependencies

Definition

A dependence exists between program statements when the order of statement execution affects the results of the program

- Results from multiple use of the same storage by different tasks.
- Primary inhibitors to parallelism.
- Loop carried dependencies are particularly important
- How to Handle Data Dependencies:
    - Distributed: communicate data at synchronization points.
    - Shared: synchronize read/write operations between tasks.

Loop Carried Dependence

Loop carried data dependence

```
for (j=1; j<N; j++) {
  vect[j] = vect [j-1] * 2;
}
```

- vect[J-1] must be computed before vect[J]: data dependency
- If Task 2 has vect[J] and task 1 has vect[J-1], computing the correct value necessitates:
  - Distributed: task 2 must obtain the value of vect[J-1] from task 1 after it finishes computation
  - Shared: task 2 must read vect[J-1] after task 1 updates it

Lood Independent Dependence

Loop independent data dependence

```
task 1          task 2
------          ------

X = 2           X = 4
  .               .
  .               .
Y = X**2        Y = X**3
```

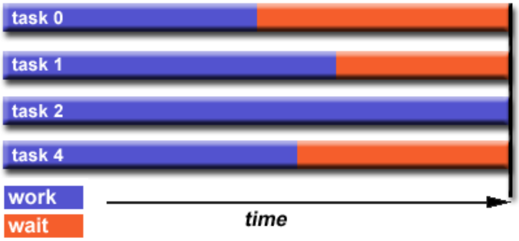
- As before, parallelism is inhibited.
- The value of Y is dependent on:
    - Distributed: if or when the value of X is communicated between the tasks.
    - Shared: which task last stores the value of X.

Load Balancing

Definition

Practice of distributing work among tasks so that all tasks are kept busy all of the time: minimization of task idle time.

- Important for performance reason
- Example: if all tasks are subject to a barrier, the slowest will determine overall performance.



Load Balancing II

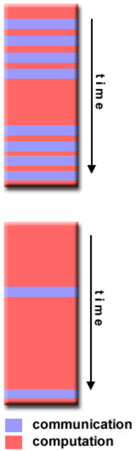
How to Achieve Load Balance:

- Equally partition the work each task receives
  - For array/matrix operations: evenly distribute the data set.
  - For loop iterations: evenly distribute the iterations.
  - Heterogeneous mix of machines: analysis tool to detect imbalances.
- Use dynamic work assignment
  - Certain problems result in load imbalances anyway
    - Sparse arrays
    - Adaptive grid methods
  - If workload is variable or unpredictable: use a scheduler. As each task finishes, it queues to get a new piece of work.
  - Design an algorithm which detects and handles load imbalances as they occur.

Granularity

Granularity: The ratio of computation to communication

- Fine: Low computation, high communication
  - High communication overhead
  - Facilitates load balancing
- Coarse: High computation, low communication
  - Potential for performance
  - Harder to load balance efficiently



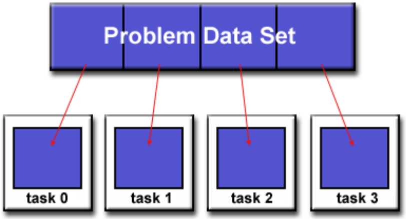
Outline

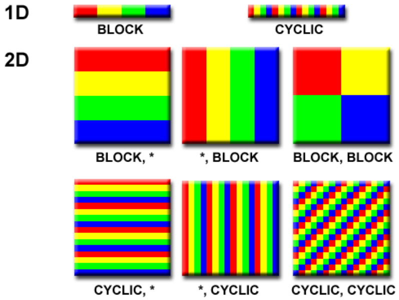
Partitioning

- One of the first steps in design is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks.
- Two basic ways:
  - Domain decomposition
  - Functional decomposition

Domain Decomposition

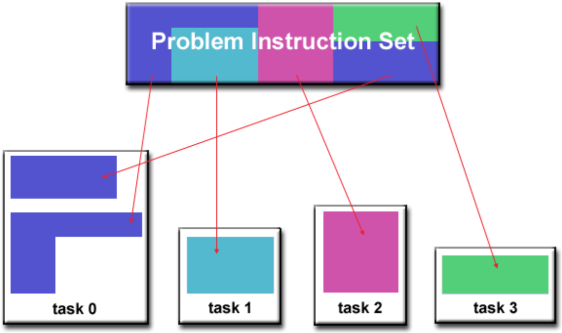
Data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.





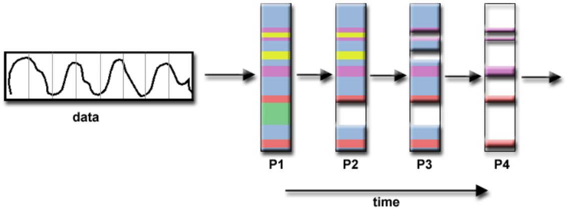
Functional Decomposition

Problem is decomposed according to the computation rather than to data. Each task then performs a portion of the overall work.



Signal Processing Example

An audio signal data set is passed through four distinct computational filters. Each filter is a separate process. The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth segment of data is in the first filter, all four tasks are busy.



Outline

Complexity

- Parallel applications could be much more complex
  - Multiple instruction streams
  - Data flowing
- The costs of complexity are measured in programmer time in every aspect of the development cycle:
  - Design
  - Coding
  - Debugging
  - Maintenance
- Good software development practices are essential when working with parallel applications

Portability

- Thanks to standardization in several APIs, portability issues are not as serious as in years past
- However, all of the usual serial portability issues apply
    - Implementations
    - Operating systems
    - Hardware architectures

Resource Requirements

- Decrease in wall clock time, often means increase in CPU time
- The amount of memory required can be greater
  - Data replication
  - Overheads
- For short parallel programs, performance can decrease

Amdahl's Law I

- Potential program speedup is defined by the fraction of code (P) that can be parallelized:

$$\text{speedup} = \frac{1}{1 - P}$$

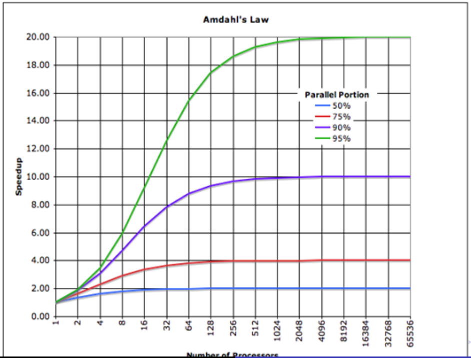
- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup). If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).

Amdahl's Law II

- Introducing the number N of processors (S is serial part):

$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

- There are limits to the scalability of parallelism.



Speed Up

$$Speedup(N) = \frac{Time_{serial}}{Time_{parallel}(N)}$$

Linear SpeedUp: Ideally, doubling the number of CPUS the execution time is halved (Speedup = N)

In reality, this happens hardly ever, because some software cannot be completely parallelized

Amdalh's law

This law gives the **theoretical speedup** a parallel application can achieve

Be:

- S the fraction of the code that cannot be parallelized (serial execution)
- P the fraction of the code that can be parallelized
- S + P = 1

$$Speedup(N) = \frac{Time_{serial}}{Time_{parallel}(N)} = \frac{(S+P)T_{serial}}{S \cdot T_{serial} + \frac{P \cdot T_{serial}}{N}} = \frac{S+P}{S+\frac{P}{N}} = \frac{1}{S+\frac{P}{N}}$$

Amdalh's law

$$\lim_{N \to \infty} \frac{1}{S + \frac{P}{N}} = \frac{1}{S}$$

$$\lim_{S \to 0} \frac{1}{S + \frac{P}{N}} = \frac{1}{S + \frac{1-S}{N}} = N$$

Observations:

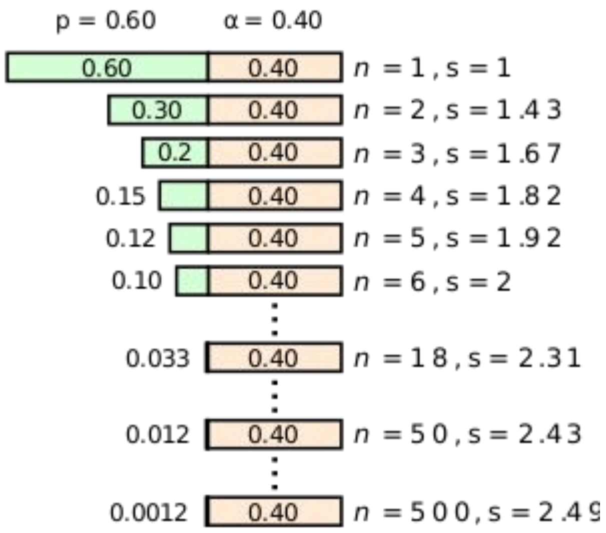
- From the first limit: the fraction of serial code **is a bound** of the scalability
- From the second limit: if there wasn't any serial code, the speed up is equal to N (**linear speed**)

Amdalh’s law: an example

If Serial Code is 10% (S = 0.10 and P = 0.90), the **highest speedup** we can get is 10, regardless the number of CPU used

$$\lim_{N \to \infty} \frac{1}{S + \frac{P}{N}} = \lim_{N \to \infty} \frac{1}{0.1 + \frac{0.9}{N}} = \frac{1}{0.1} = 10$$

Amdalh’s law: a graphical explanation



p = fraction of the code which can be executed in parallel mode

ɑ = fraction of the code which can be executed in serial mode

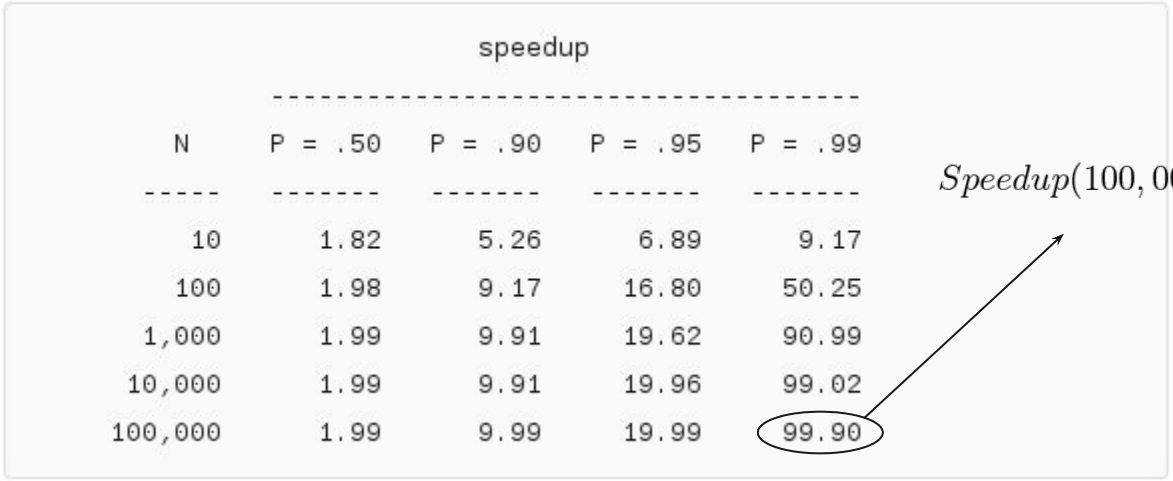
n = core number

s = SpeedUp

$$Speedup(2) = \frac{1}{S + \frac{P}{N}} = \frac{1}{0.40 + \frac{0.60}{2}} = 1.43$$

Limit of the Parallel Programming

Speedup is strongly affected by the fraction of serial code

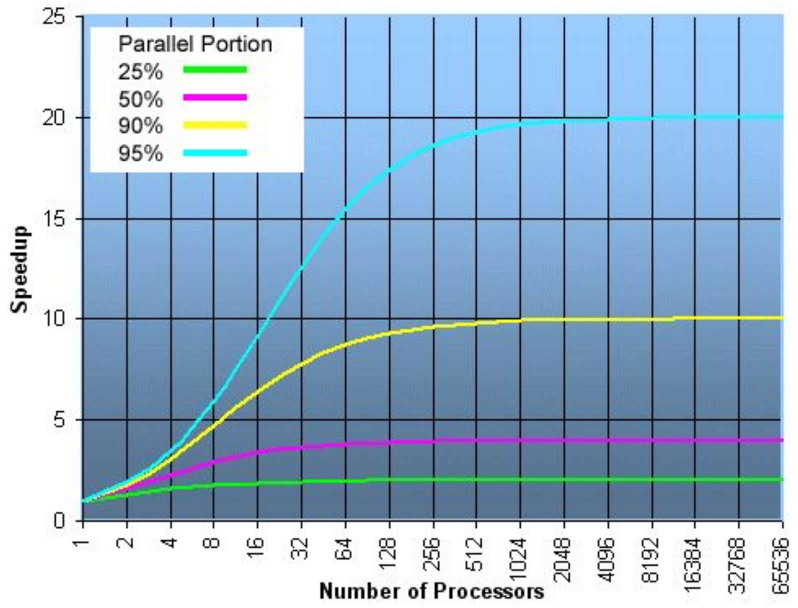


```
                        speedup
            --------------------------------------
   N        P = .50   P = .90   P = .95   P = .99
 -----      -------   -------   -------   -------
    10         1.82      5.26      6.89      9.17
   100         1.98      9.17     16.80     50.25
 1,000         1.99      9.91     19.62     90.99
10,000         1.99      9.91     19.96     99.02
100,000        1.99      9.99     19.99     99.90
```

$$Speedup(100,000) = \frac{1}{S + \frac{P}{N}} = \frac{1}{0.01 + \frac{0.99}{100{,}000}} = 99.90$$

Limit of the Parallel Programming

Speedup is strongly affected by the fraction of serial code



Superlinear speed

Some application might achieve performance even better than the linear speed, that is $S > N$

This might happen for several reasons, for example due to the CPU cache

Scalability

$$Scalability(N) = \frac{T_{parallel}(1)}{T_{parallel}(N)}$$

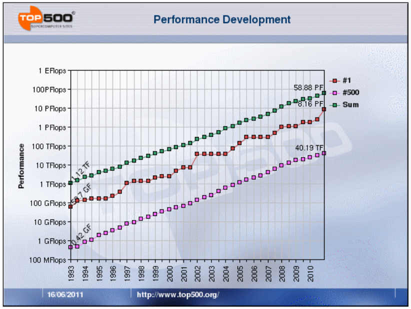
Quite similar to the SpeedUp but instead of considering the execution time of the serial implementation, it takes the execution time of the parallel implementation with a ***parallel degree equal to 1***

Scalability

- Strong Scaling (Amdahl)
    - The total problem size stays fixed as more processors are added.
    - Goal is to run the same problem size faster
    - Perfect scaling means problem is solved in 1/P time (compared to serial)
- Weak Scaling (Gustafson)
    - The problem size per processor stays fixed as more processors are added.
    - The total problem size is proportional to the number of processors used
    - Goal is to run larger problem in same amount of time
    - Perfect scaling means problem P runs in same time as single processor run

The Future:

During the past 20+ years, the trends indicated that parallelism is the future of computing.
In this same period, 1000x increase in supercomputer performance.
The race is already on for Exascale Computing!



For Further Reading I

A. Grama, A. Gupta
Introduction to Parallel Computing
Addison-Weasley, 2003.

Blaise Barney
Introduction to Parallel Computing, 2011
`https://computing.llnl.gov/tutorials/`