

Memory Models in HPC: Successes and Problems

Antun Balaz

Agenda

- Introduction: HPC, performance, speedup
- Parallel computing and memory models
- Programming models and Flynn's taxonomy
- MPI vs. OpenMP
- Hybrid programming model and mismatch problems
- Thread safety
- Hybrid parallelism and programming strategies
- Transactional memory
- Memory hierarchy: GPGPUs, FPGAs, ...

What HPC stands for?

- High Performance Computing
- The term is most commonly associated with computing used for scientific research [from Wikipedia]
- It involves not only hardware, but software and people as well!
- HPC encompasses a collection of powerful:
 - hardware systems
 - software tools
 - programming languages
 - parallel programming paradigmswhich make previously unfeasible calculations possible

Only performance?

- High Throughput Computing
- High Availability Computing
- Capacity Computing
- Capability computing
- To reflect a greater focus on the productivity, rather than just the performance, of large-scale computing systems, many believe that HPC should now stand for High Productivity Computing

Performance vs. Productivity

- A definition:
 - $\text{Productivity} = \frac{\text{application performance}}{\text{application programming effort}}$
- Scientists in HPC arena have different goals in mind thus different expectations and different definitions of productivity.
- Which kind of productivity are you interested in?

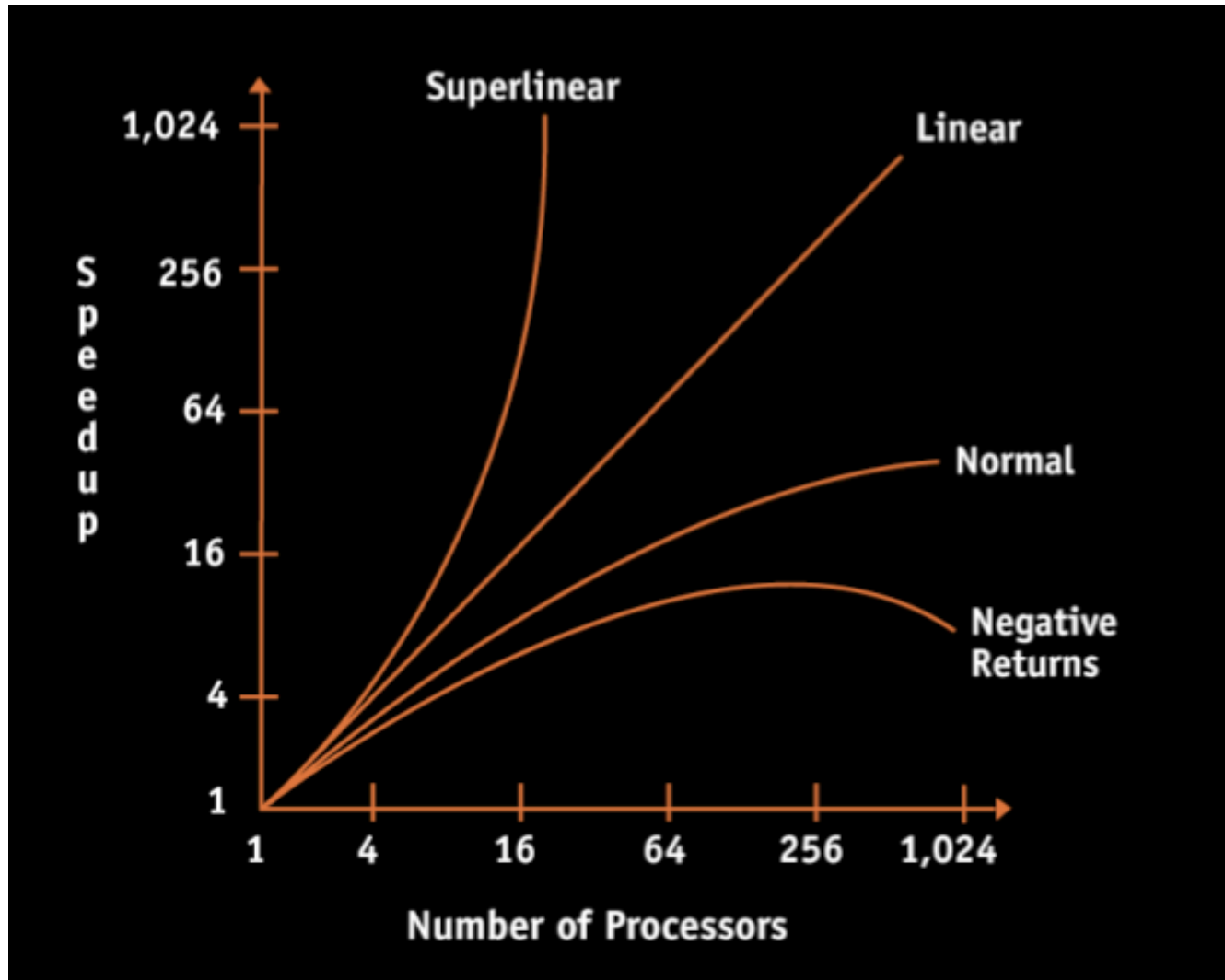
Measures of performance

- How fast can I crunch numbers on my CPU?
- How much data can I store?
- How fast can I move the data around?
 - from CPUs to memory; from CPUs to disk; from CPUs to/on different machines
 - among computers: networks
 - default (commodity): 1 Gb/s
 - custom (high speed): 10Gb/s, 20 Gb/s and now 40Gb/s or more
 - within the computer:
 - CPU – Memory: thousands of Mb/s: 10 - 100 Gb/s
 - CPU - Disks: MByte/s: 50 ~ 100 MB/s up to 1000 MB/s

Parallel performance

- The speedup of a parallel application is **Speedup(p) = Time(1) / Time(p)** where:
Time(1) = execution time for a single processor
Time(p) = execution time using p parallel processors
- If **Speedup(p) = p**, we have a perfect speedup (also called linear scaling)
- Speedup compares performance of an application with itself on one and on p processors
- More useful to compare:
 - The execution time of the best serial application on 1 processor vs.
 - The execution time of the best parallel algorithm on p processors

Speedup



Superlinear speedup?

- Can we find **superlinear** speedup, i.e.,
 $\text{Speedup}(p) > p$
- Yes, we can:
 - Choosing a bad “baseline” for $T(1)$
 - Old serial code has not been updated with optimizations
 - Shrinking the problem size per processor
 - May allow it to fit in small fast memory (cache)
 - Total time decreased because memory optimization tricks can be played.

Question

- Algorithm A and algorithm B solve in parallel the same problem
- We know that on 64 core:
 - Program A gets a speedup of 50
 - Program B gets a speedup of 4
- Which one do you choose ?
 - 1) program A
 - 2) program B
 - 3) None of the above

Answer

- None of the above, since we do not know the **overall execution time** of each of them
- What if A is sequentially 1000 time slower than B?
- We always have to use the best sequential algorithm for computing speedup (absolute speedup)
- And the best compiler to produce the executable, for both serial and parallel version of the application!

Limits to speedup

- All parallel programs contain:
 - Parallel sections
 - Serial sections
- Serial sections limit the speed-up:
 - Lack of perfect parallelism in the application or algorithm
 - Imperfect load balancing (some processors have more work)
 - Cost of communication
 - Cost of contention for resources, e.g., memory bus, I/O
 - Synchronization time
- Understanding why an application is not scaling linearly will help finding ways improving the applications performance on parallel computers.

Amdahl's law (1)

- Let S be the fraction in an application representing the work done serially
- Then, $1-S = P$ is fraction done in parallel
- What is the maximum speedup for N processors?

$$\textit{speedup} = \frac{1}{(1-P) + \frac{P}{N}} \Rightarrow \lim_{N \rightarrow \infty} \frac{1}{1-P}$$

- Even if the parallel part scales perfectly, we may be limited by the sequential portion of the code

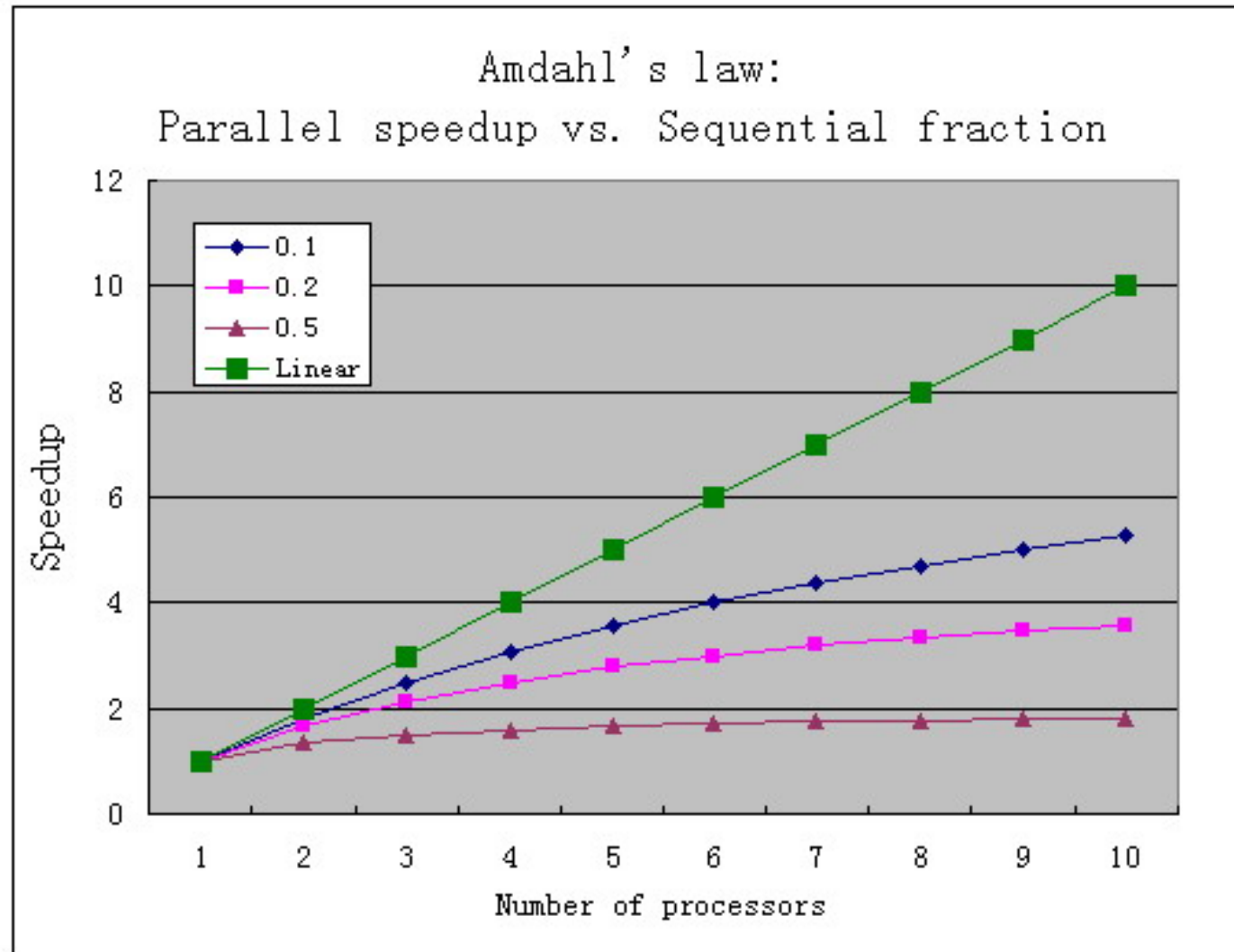
Amdahl's law (2)

- The presence of a serial part of the code is quite limiting in practice:

>	2	4	8	32	64	256	512	1024
5%	1.91	3.48	5.93	12.55	15.42	18.62	19.28	19.63
2%	1.94	3.67	6.61	16.58	22.15	29.60	31.35	32.31
1%	1.99	3.88	7.48	24.43	39.29	72.11	83.80	91.18

- Amdahl's Law is relevant only if serial fraction is independent of the problem size
- Fortunately, the proportion of the computations that are sequential (non parallel) usually decreases as the problem size increase (a.k.a. Gustafson's law)

Effective parallel performance



How to run applications faster ?

- There are 3 ways to improve performance:
 - Work Harder
 - Work Smarter
 - Get Help
- Analogy in computer science
 - Use faster hardware
 - Optimize algorithms and techniques used to solve computational tasks
 - Use multiple computers to solve a particular task
- All 3 strategies can be used simultaneously

What is parallel computing?

- Parallel computing is the simultaneous execution of the same task (split up and specially adapted) on multiple processors in order to obtain results faster
- The process of solving a problem usually can be divided into smaller tasks, which may be carried out simultaneously with some coordination

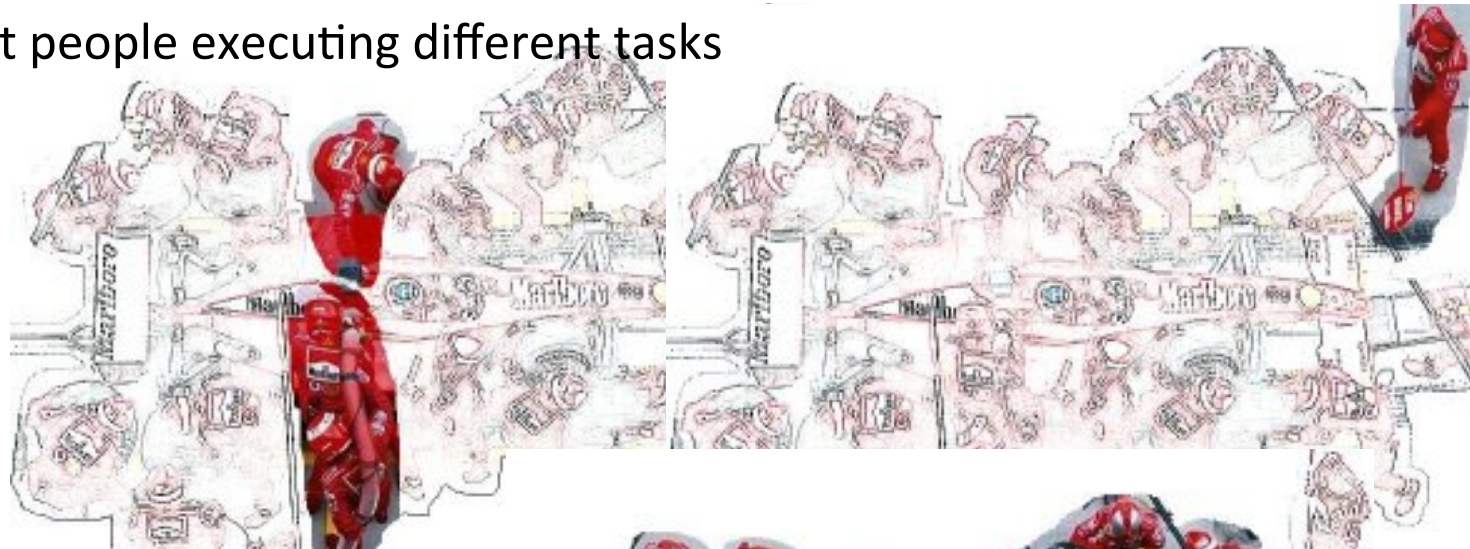
[from Wikipedia]

High performance problem:



Analysis of a parallel solution

- Functional decomposition
 - Different people executing different tasks



- Domain decomposition
 - Different people executing the same tasks

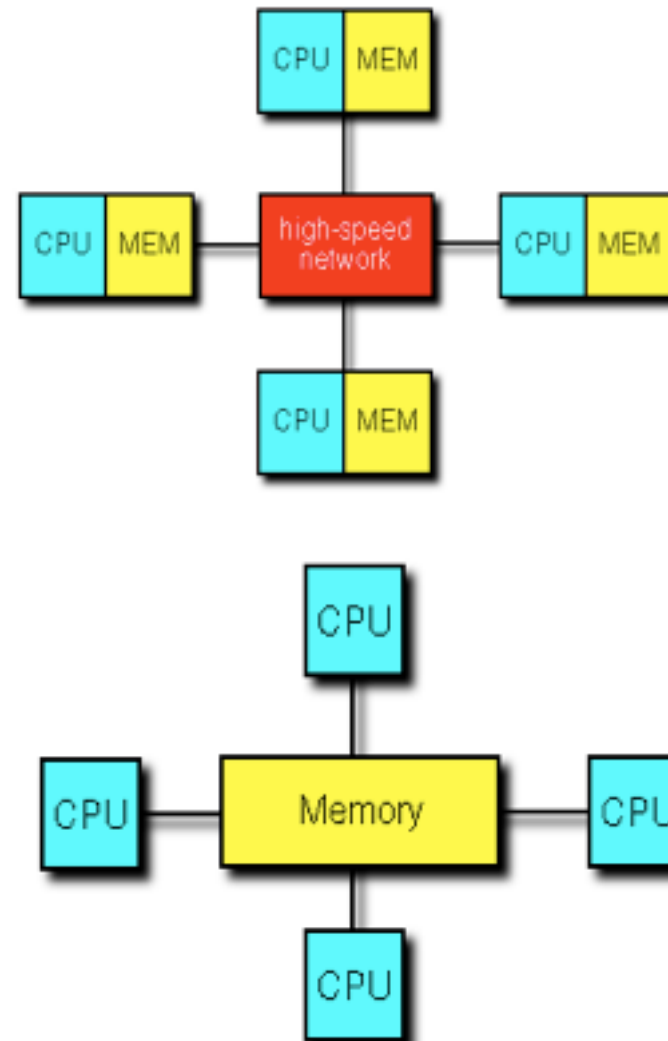


HPC parallel computers

- The simplest and most useful way to classify modern parallel computers is by their memory model.
- How CPUs view and can access the available memory?
 - Shared memory
 - Distributed memory

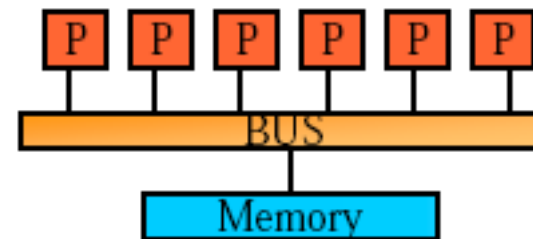
Shared vs. Distributed

- Distributed Memory:
 - Each processor has its own local memory. Must do message passing to exchange data between processors.
 - Multi-computers
- Shared Memory
 - Single address space. All processors have access to a pool of shared memory.
 - Multi-processors

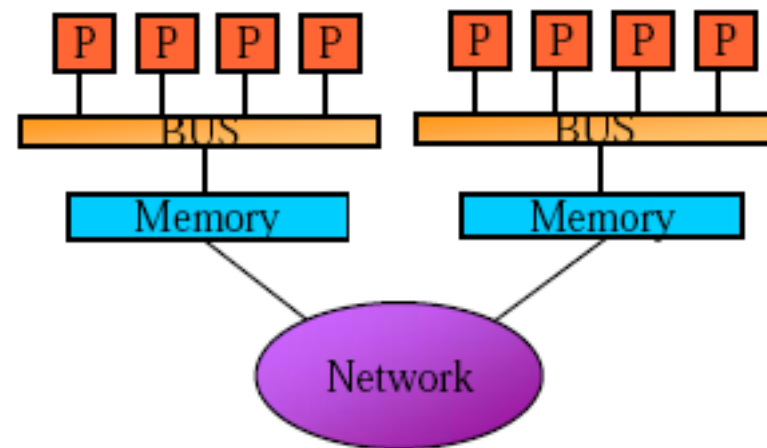


Shared Memory: UMA vs. NUMA

- **Uniform memory access (UMA):** Each processor has uniform access to memory. Also known as symmetric multiprocessors (SMP).

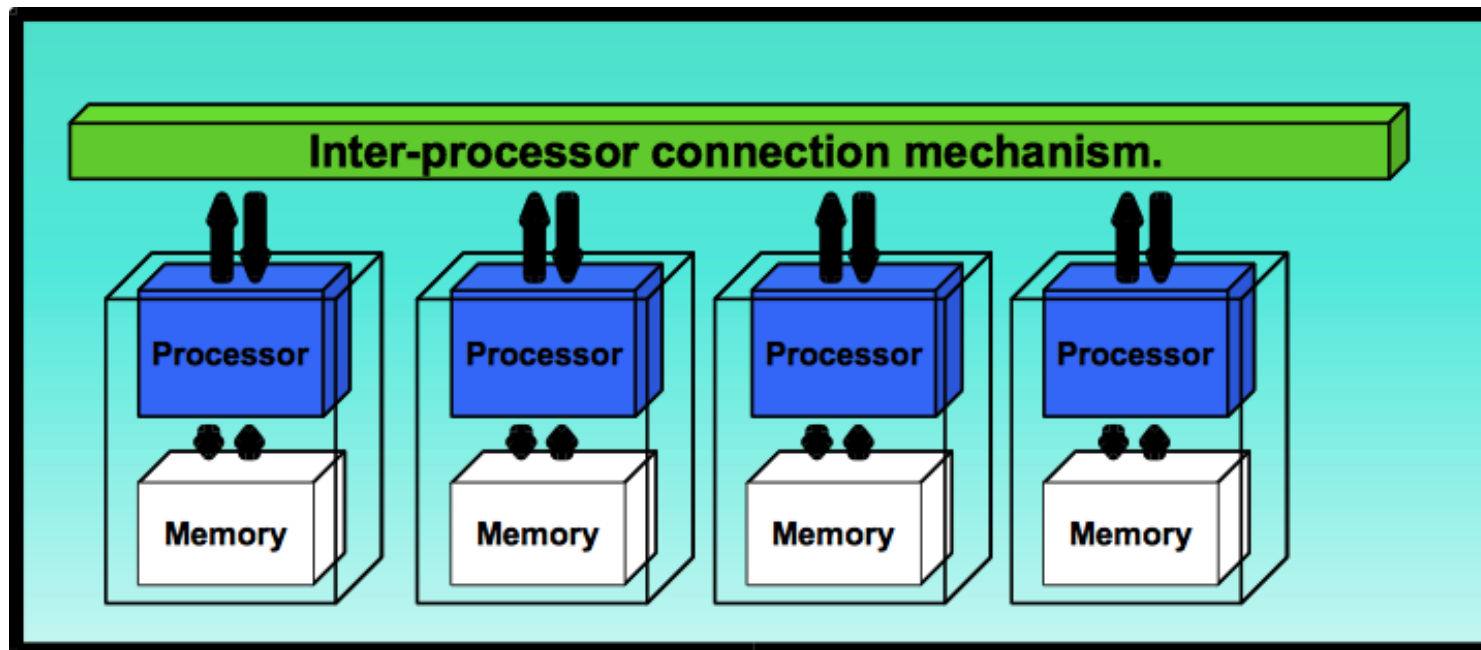


- **Non-uniform memory access (NUMA):** Time for memory access depends on location of data. Local access is faster than non-local access.



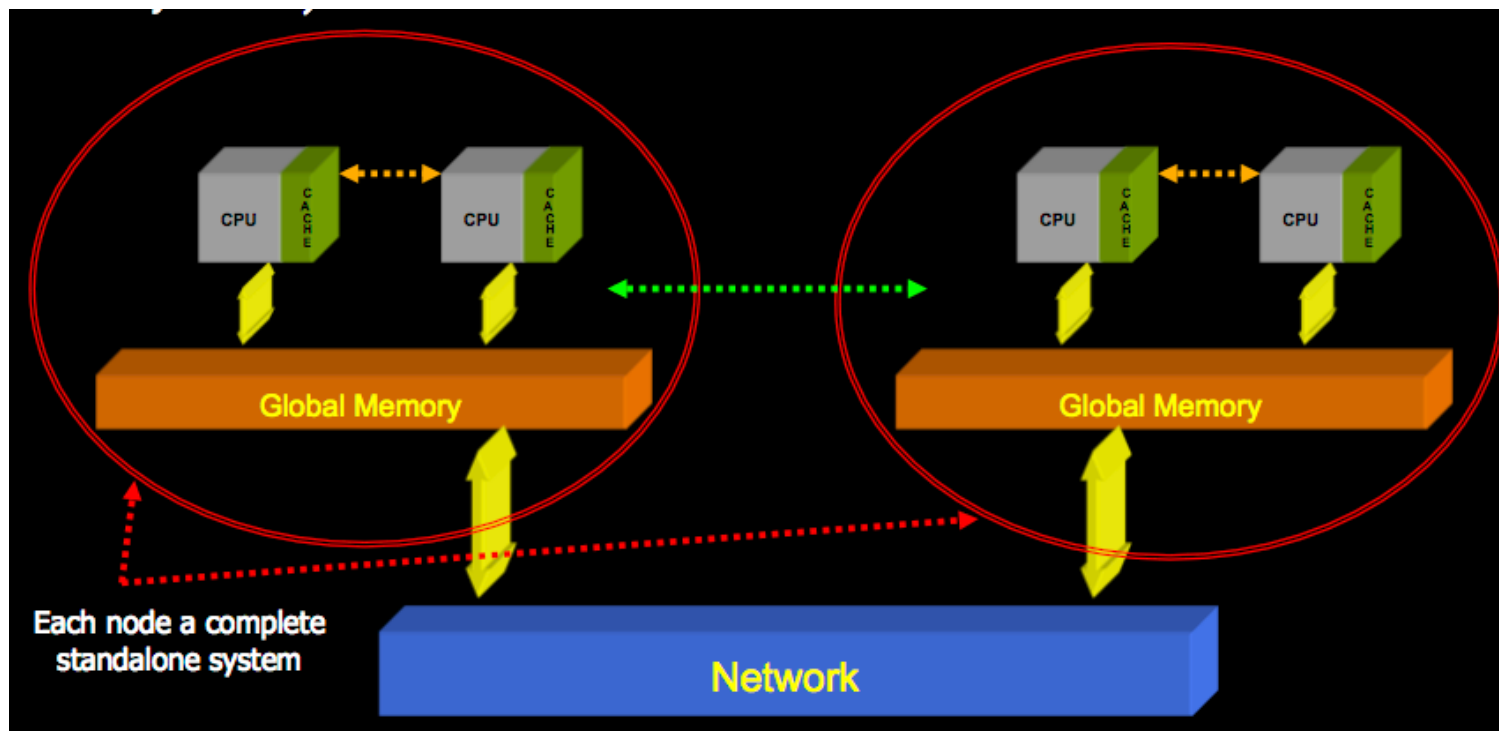
Clusters: distributed memory

- Independent machines combined into a unified system through software and networking



Hybrid architecture

- All modern clusters actually have hybrid architecture
- Many-core CPUs make each node a small SMP system



Parallel Programming Paradigms

- Memory models determine programming paradigms

Parallel machines	
Distributed memory	Shared memory
Parallel paradigms	
Message passing	Data parallel
All processes could directly access only their local memory . Explicit messages are requested to access remote memory of different processors.	Single memory view. all processes (usually threads) could directly access the whole memory .

Architecture vs. Paradigm

Clusters of Shared Memory Nodes

Shared Memory Computers

Data Parallel

Message Passing

Distributed Memory Computers

Message Passing

Architecture, Paradigm, Model

Architecture	
Distributed memory	Shared memory
Programming paradigm	
Message passing	Data parallel
Programming model	
Domain decomposition	Functional decomposition

Programming models

- Domain decomposition
 - Data divided into equal chunks and distributed to available CPUs
 - Each CPU process its own local data
 - Exchange of data if needed
- Functional decomposition
 - Problem decomposed into many sub-tasks
 - Each CPU performs one of sub-tasks
 - Similar to server/client paradigm

Flynn's taxonomy (1)

- SISD (Single instruction, single data)
- **SIMD (Single instruction, multiple data)**
 - the same instructions are carried out simultaneously on multiple data items
 - SSE is a good example
- **MISD (Multiple instruction, single data)**
- **MIMD (Multiple instruction, multiple data)**
 - different instructions on different data
- **SPSD (Single program, single data)**
- **SPMD (Single program, multiple data)**
 - not synchronized at individual operation level
 - equivalent to MIMD since each MIMD program can be made SPMD

Flynn's taxonomy (2)

- SPSD (Single program, single data)
- **SPMD (Single program, multiple data)**
- **MPSD (Multiple program, single data)**
- **MPMD (Multiple program, multiple data)**

Model	Paradigm	Flynn's taxonomy
Domain decomposition	Message Passing	SPMD
	Data Parallel - HPF	
Functional decomposition	Data Parallel - OpenMP	MPSD
	Message Passing	MPMD

Parallelism requires...

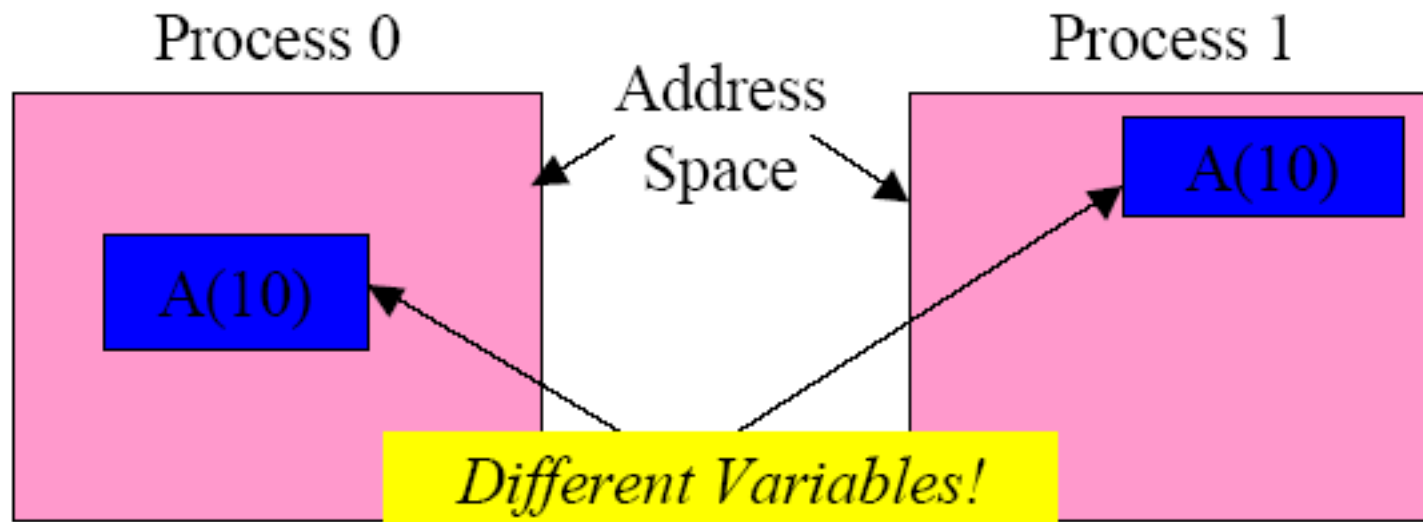
- Balancing of the load
 - Applies to computation, I/O operations, network communication
 - Relatively easy for domain decomposition, not so easy for functional decomposition
- Minimizing communication
 - Join individual communications
 - Eliminate synchronization – the slowest process dominates
- Overlap of computation and communication
 - This is essential for true parallelism!

Message Passing Interface

- Parallel programs consist of separate processes, each with its own address space
 - Programmer manages memory by placing data in a particular process
- Data sent explicitly between processes
 - Programmer manages memory movement
- Collective operations
 - On arbitrary set of processes
- Data distribution
 - Also managed by the programmer

Distributed memory

- Nothing is shared between processes



OpenMP

- Shared memory necessary
- Good for SMP nodes, but also possible on clusters via distributed shared virtual memory systems
- Parallelism achieved through the memory sharing
- Programmer responsible for proper synchronization
- Programmer also responsible for management of memory (locks)

MPI vs. OpenMP

- Pure MPI pro:
 - Portable to distributed & shared memory machines
 - Scales beyond one node
 - No data placement problem
- Pure MPI cons:
 - Difficult to develop & debug
 - High latency, low bandwidth
 - Explicit communication
 - Large granularity
 - Difficult load balancing
- Pure OpenMP pro:
 - Easy to implement parallelism
 - Low latency, high bandwidth
 - Implicit communication
 - Coarse & fine granularity
 - Dynamic load balancing
- Pure OpenMP cons:
 - Only on shared memory machines
 - Scales within one node
 - Possible data placement problem
 - No specific thread order

Standards

- MPI inter-node
 - Message passing
 - Data distribution model
 - Version 2.2 (09/2009)
 - Version 3.0 (09/2012)
 - API for C/C++ and Fortran
- OpenMP intra-node
 - Threads
 - Relaxed-consistency model
 - Version 3.1 (07/2011)
 - Version 4.0 (07/2013)
 - Compiler directives for C/C++ and Fortran

Hybrid programming model

- Hybrid MPI/OpenMP paradigm is the parallel approach for clusters of SMP architectures
- Elegant in concept and architecture: using MPI across nodes and OpenMP within nodes
- Good usage of shared memory system resource (memory, latency, and bandwidth)
- Avoids the extra communication overhead with MPI within node
- OpenMP adds fine granularity (larger message sizes) and allows increased and/or dynamic load balancing
- Some problems have two-level parallelism naturally
- Some problems could only use restricted number of MPI tasks
- Could have better scalability than both pure MPI and pure OpenMP

Mismatch problems

- Topology problem (with pure MPI)
- Unnecessary intra-node communication (with pure MPI)
- Saturation problem (with pure MPI)
- Sleeping threads (with OpenMP)
- Inter-node bandwidth problem (with hybrid)
- Additional OpenMP overhead (with hybrid)
 - Thread startup / join
 - Cache flush (data source thread)
- Overlapping communication and computation
 - application problem
 - programming problem
 - load balancing problem
- **no silver bullet – each scheme has its problems**

Thread safety

- The MPI-2 and MPI-3 standards define four different levels of thread safety, in the form of how an MPI implementation can perform communication between processes:
 - MPI_THREAD_SINGLE: there is only one thread in the application.
 - MPI_THREAD_FUNNELED: only one thread may make MPI calls.
 - MPI_THREAD_SERIALIZED: any threads may make MPI calls, but only one at a time.
 - MPI_THREAD_MULTIPLE: any thread may make MPI calls at any time.
- The level of multi-threading strongly depends on the hardware and MPI implementation.
- All MPI implementations support MPI_THREAD_SINGLE (MPI-1 too).
- Usually when people refer to an MPI implementation as thread-safe, they mean that the implementation supports the maximum level of functionality

Relationship

- In parallelizing a code with the hybrid paradigm, MPI is used for coarse-grain parallelism (i.e., principal data decomposition), while OpenMP provides fine-grain parallelism inside each MPI process.
- There are three main different mixed mode programming models depending on the way the MPI communication is being handled:
 - Master-only, where all MPI communication takes place outside of OpenMP parallel regions
 - Funnelled, where communication may occur inside parallel regions, but is restricted to a single thread
 - Multiple, where more than one thread can call MPI communication routines

Programming strategies

- Implicit
 - Use of threaded numerical libraries, which are then linked after compilation
 - No control over thread overhead
 - Very simple
- Explicit
 - Use of explicit OpenMP syntax
 - Full control
 - More complex, but also more efficient

Example: libraries for linear algebra

- Implicit (thread level)
 - BLAS, LAPACK, FFTW
 - ESSL (IBM AIX)
 - MKL (Intel)
 - ACML (AMD)
- Explicit (MPI level)
 - PBLAS
 - SCALAPACK

Transactional Memory

- Replaces waiting for locks with concurrency
- Allows non-conflicting updates to shared data
- Shown to improve scalability of short critical regions
- Promise of Transactional Memory
 - Program with coarse transactions
 - Performance like fine-grained lock
- Focus on correctness, tune for performance
 - Easier to reason about only a few transactions...
 - ... only focus on areas with true contention
- Hardware TM implementation:
 - Intel's TSX, as of Haswell-EX (disabled in E and EP models due to a bug discovered in August 2014)
 - IBM's Blue Gene/Q, zEnterprise EC12, POWER8
- Compilers: vendor-specific, gcc-4.7

Transactional Memory

Lock

```
...  
synchronized (lock) {  
    x = x + y;  
}  
...
```

Mapping from lock to protected data

- **lock** protects **x**

Transaction

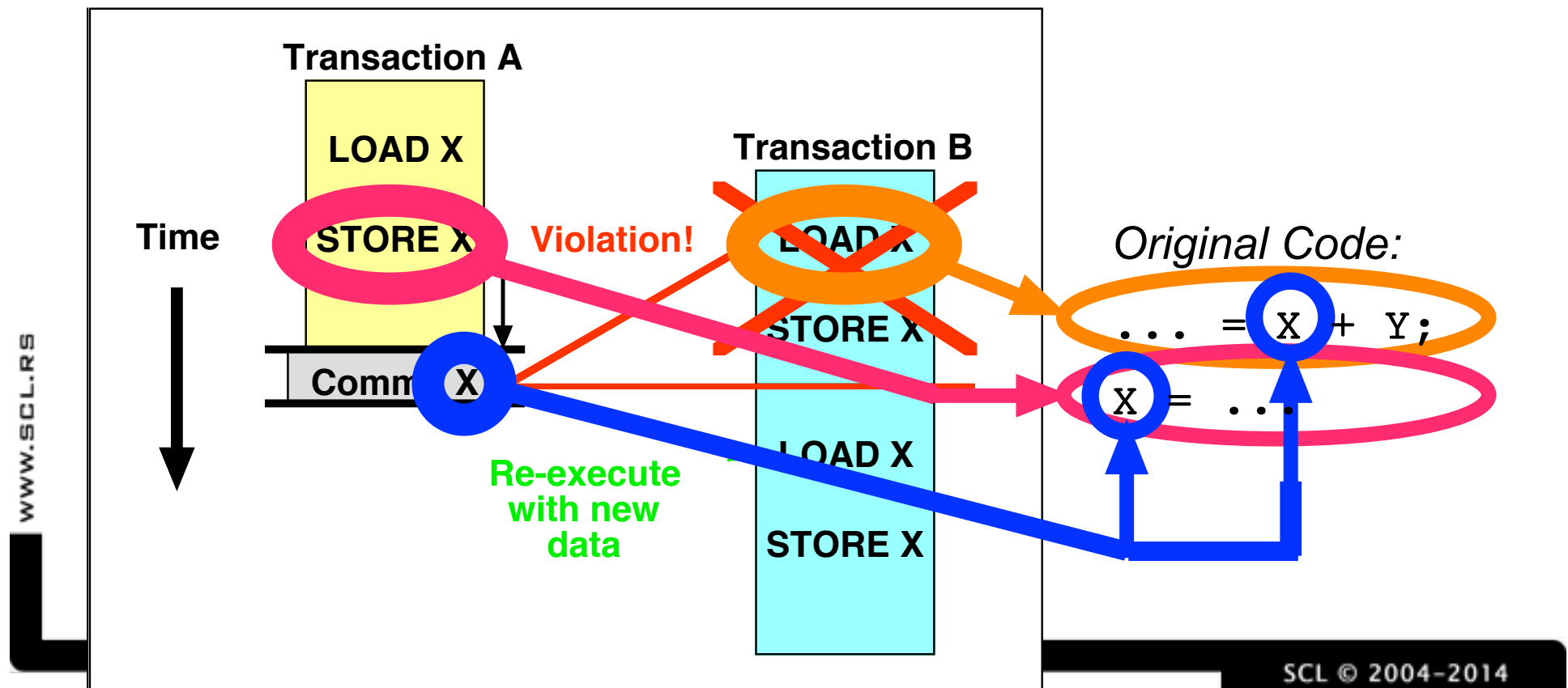
```
...  
atomic {  
    x = x + y;  
}  
...
```

Transaction protects all data

- No need to worry if another lock is necessary to protect **y**

Transactional Memory at Runtime

- What if transactions modify the same data?
 - First commit causes other transactions to abort & restart
 - Can provide programmer with useful feedback!



Memory hierarchy: GPGPUs, FPGAs, ...

- Everything becomes more complex with the advance of GPGPUs and FPGAs
 - Programming approaches
 - Libraries
 - Porting
 - Memory models
- How to efficiently handle memory in a three-layer hybrid model: MPI + OpenMP + GPGPU
- Multi-GPGPU or multi-FPGA configurations

Summary

- Memory models rapidly evolve in HPC
 - Memory hierarchy ever more complicated, from a simple RAM to cache(s)+RAM(s)+disk+GPGPU+...
- Hybrid programming a necessity
 - Hybridization involves more and more layers
- Multi-dimensional eternal search for
 - Performance
 - Scalability
 - Energy efficiency
 - ...
- The black arts now need to be taught even to undergraduates!