# CS201 W'00 Lecture Notes
## Lecture 12



## Parallel Architectures

Have seen many performance benefits from introducing parallelism **inside** processors (e.g., pipelining).

• Do instructions or parts of instructions in parallel; very small **granularity** of parallelism.

What about building systems using **multiple** (entire) processors?

• Support larger-granularity parallelism, e.g., at statement, function, or process level.

Two basic architectures:

• In **multiprocessors**, all CPUs are connected to a common **shared memory**.

– CPUs communicate via common address space for loads/stores.

• In **multicomputers**, each CPU has private memory.

– CPUs communicate via explicit message-passing interconnect.

These architectures are actually more similar than different:

• At lowest levels, even shared-memory systems must communicate messages to/from memory units.

• Can use software to simulate message passing in shared memory or vice-versa.



## Speedup

The whole point of parallel systems is to achieve **speedup** in execution, at reasonable cost.

Common measure of cost is number of processors.

Ideal **linear** speedup is factor of $p$ when $p$ processors are used.

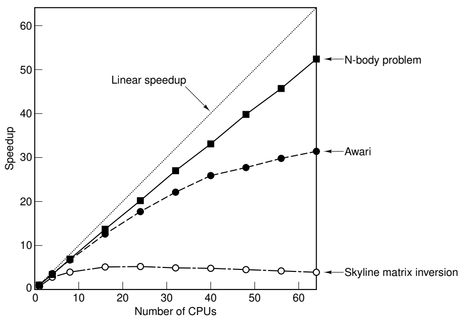


**Figure 8-8.** Real programs achieve less than the perfect speed-up indicated by the dotted line.



## Scalability

Since most parallel programs have substantial **sequential** components, Amdahl's law limits the achievable speedup.

Also, **communication** and **synchronization** costs limit speedup.

Common way to lie: give speedup in terms of improvement over parallel program with $p = 1$ rather than over sequential program.

Major challenge in parallel architecture design: maintain speedup ratio as $p$ increases.

• Systems that achieve this are **scalable**.

Good speedup is typically easier to achieve using **large-grained** parallelism:

• programs spend less time **communicating** and **synchronizing**.

• latency of underlying interconnect matters less.

## Interconnection networks

Ultimately, all parallel systems must support communication between the processors and between processors and memory units.

• Wide variety of hardware schemes for interconnects.

• Issues include topology, switching, routing.

• Measure interconnect's performance in terms of **latency** and **bandwidth**.

– Usual tradeoff between cost and performance.

– Very hard to maintain low latency as system grows.



## Shared Memory Examples

For small $p$ (2,4,8) can build **bus-based**, **symmetric** multiprocessor.

• Becoming quite common as servers (e.g., **sirius**).

• Commonly used to support process-level parallelism (CPU's almost completely independent).

For larger $p$ (100's) a single bus won't work. Instead, use a **hierarchy** of interconnects.

• In **NUMA** = **non-uniform** memory architecture, CPU's have fast access to nearby memory, and slower access to distant memory.

• Example: Sequent NUMA-Q supports up to 252 CPUs and $2^{38}$ bytes of memory, using 4-processor/4GB/1-bus boards, connected by ring interconnect.

For all shared-memory systems, big issue is **cache coherence**: how to keep per-processor **caches** in agreement with each other and with main memory.



## Multicomputer Examples

For very large $p$ (1000's), we don't know how to build effective shared-memory machines, so must go to multicomputers.

**Massively Parallel Processors** are today's supercomputers.

• Typically connect large numbers of standard CPU's using specialized high-performance interconnect network.

• Example: Intel/Sandia Option Red has 4536+ processors and runs at 1 teraflop/sec (roughly 5,000 times the floating-point performance of a 200MHz Pentium Pro).

**Clusters of Workstations (COW)** just harness existing local-area networks of workstations or PC's to do coordinated tasks.

• Doesn't have to be dedicated to parallelism (e.g., could run just at night when workstations aren't being used by people).

• Programs typically use a message passing **library** like **PVM** (Parallel Virtual Machine).