Programming Parallel Computers:

Programming SIMD Machines:

Data Parallel Languages:

HPF, HPC, HPC++.

Constructs such as C = A + B (where A, B, C are actually matrices)

Programming MIMD Machines:

- *Message passing paradigm*: Programs cooperate via messages. Regular languages are augmented by functions for sending, receiving messages.

- *Shared memory programming*: Regular languages augmented with support for synchronization and locking shared data.

Programming Parallel Computers:

- ❑ Parallelism arising from multiplicity of data; single instruction operating on multiple data items (e.g.. adding two matrices). Each of the multiple processors executes the same instruction despatched by a central controller which is also responsible for synchronization after each instruction (*Single instruction, multiple data or SIMD*).

- ❑ Often, it may be difficult to extract parallelism based on multiplicity of data. Why not just let the processors work independently on their own subproblems and exchange results by communicating when necessary. The processors may be executing different programs altogether operating on different data (*Multiple Instructions, Multiple Data or MIMD*).

- ❑ A halfway concept called *Single Program, Multiple Data (or SPMD)* provides fast synchronization of SIMD and flexibility of MIMD.

Interconnection Networks for Parallel Computers:

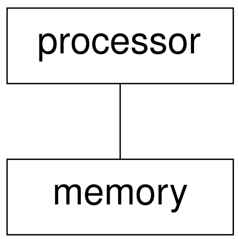
- ❑ Rings (1-D mesh): Kendall Square Research KSR1, KSR2.
- ❑ Mesh (2-D mesh): Intel Delta
- ❑ Mesh (3-D mesh / Torus / Donut): Cray T3D, T3E.

Advantages: Many physical applications map naturally to meshes.

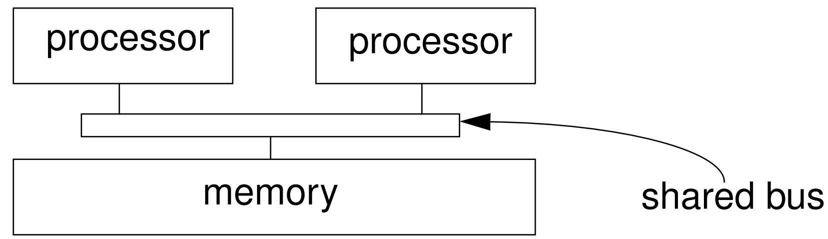
- ❑ Hypercubes: nCUBE, Cosmic Cube.
- ❑ Fat Trees: Thinking Machines CM5.

Advantages: High communication bandwidth, Proximity of processors.

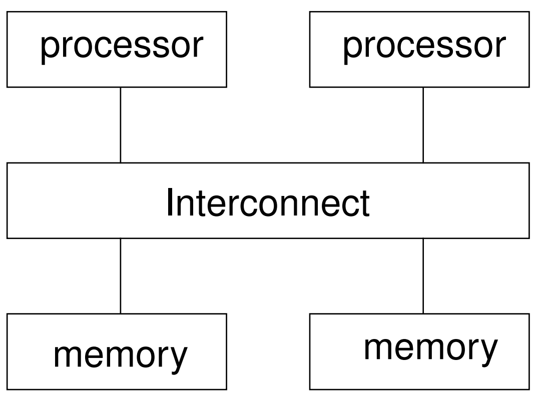
Evolution of Parallel Computing Hardware:



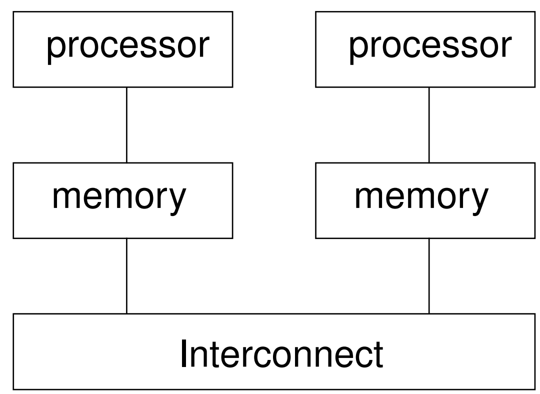
(a) Uniprocessor



(b) Shared-bus multiprocessor



(c) Shared-memory multiprocessor

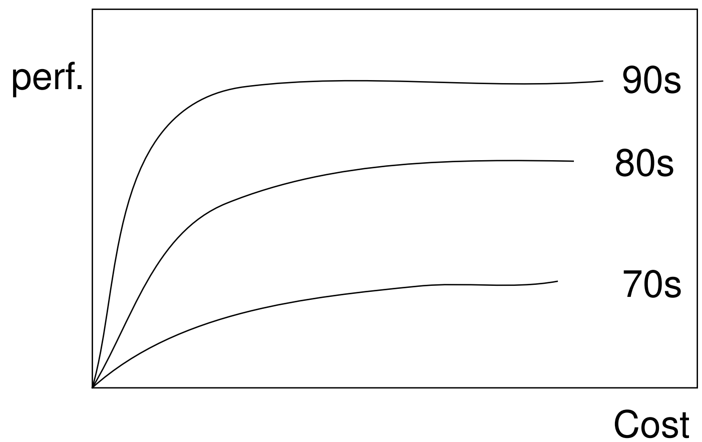


(d) Distributed memory multiprocessor.

A Case for Parallel Processing:

- Perhaps the only way of getting very high performance.

- A cheaper way of getting mid-range performance
(ten one GFLOPS processors are much cheaper than one ten GLOPS Processor).



- Improved memory system performance because of better bandwidth (typically) to memory.

What have we learned so far?

- ❏ Memory system performance is CRITICAL!
- ❏ Caches help performance, but complicate programming significantly.
- ❏ Clock speeds are saturating (only expected to go up 2 X in the next 10 years!! Compare this to a 25 fold increase in the past 10 years).
- ❏ Processors can be made to go faster using techniques such as pipelining, superscalar execution and VLIW processors.
- ❏ There are however problems with each of these approaches that limits the speed achievable by a single processor (or what appears to be a single processor).

How do we get more speed?!

How do you make a processor go even faster?

Execute multiple instructions in one processor cycle.

❏ *Superscalar execution*: Processor has multiple add, multiply units. The processor looks at the next (say) 256 instructions, picks out a possible add and a multiply that can be executed simultaneously and schedules them during the same cycle.

Problem: Scheduling is done by hardware. As number of superscalar units increases, the complexity of this hardware increases significantly. Also, many applications just do not have the kind of concurrency.

❏ *Very Long Instruction Word (VLIW) Processors*: The scheduling here is done by the compiler that picks out a fixed number of instructions that are scheduled simultaneously on multiple execution units.

Problem: Compilers do not always work. Code must be recompiled if we change the number of functional units.

Evolution of Processor Architectures:

- ❑ *t-15* years (microprocessors): Simple scalar processors, some instructions take one cycle, others take more.

- ❑ *t-10* years: The RISC revolution starts. Identify instructions that are most frequently executed and optimize the processor for these instructions. Other (complex) instructions are implemented as subroutine calls.

- ❑ Pipelined execution: instructions are executed in the cycle: fetch-decode-execute. One can pipeline this as follows:

I1:fetch	decode	execute			
I2:	fetch	decode	execute		
I3:		fetch	decode	execute	
I4:			fetch	decode	execute

Once the pipeline has been filled, it is possible to execute one instruction in each cycle.

Problem: what happens on a goto (or jump) instruction?

Cache Performance:

Example 1: Dot product.

No data reuse; latency hiding is to the extent of memory banking.

Example 2: Matrix multiplication.

Algorithm 1: View matrix multiplication as n^2 dot products.

No (or incidental) data reuse, memory system performance latency bound.

Algorithm 2: View matrix multiplication in terms of products of little submatrices.

Excellent data reuse. High cache hit ratios. This is the best benchmark for peak performance of most current microprocessors.

Placement Policies:

Direct mapped caches: Each memory location has a unique location in the cache. Say the cache is one-tenth the size of the main memory, then location m in the main memory is mapped to location (m modulo 10) in the cache.

Problems: Adding two vectors where the vectors happen to have a conflicting map.

Set Associative Cache: A k-way set associative cache has k possible locations in cache for each memory location. Consider a 2-way set associative cache. The total cache is one-tenth the size of the main memory as before. In this case, view the cache as two blocks, each, a fifth of the size of the main memory. Word m in the memory can be found either at (m modulo 5) in the first block, or (m modulo 5) in the second block.

Problems: more sophisticated cache hardware.

Principles of Cache Performance:

Hit Ratio: The fraction of the total number of memory requests that are serviced from the cache. Hit ratio h_r can be as high as 98% for some applications.

if t_{cache} is time to service data request from cache, and t_{main} is time to service data request from main memory, then effective access time of the memory system for the application is

$t_{effective} = h_r * t_{cache} + (1-hr) * t_{main}$.

Say $h_r = 0.98$, $t_{main} = 50$ ns, and $t_{cache} = 10$ ns,

$t_{effective} = 10.8$ ns !!

Principles of Cache Performance:

Temporal Locality:

If a processor accesses a piece of data, it is likely that it will use the data again in a small time window.

(Data reuse solves bandwidth problem)

Spatial Locality:

If a processor makes a reference to a piece of data in the memory, it is likely that subsequent references would be to data in the neighborhood.

(Solves latency problems, hint: think banking)

Memory Delays:

Latency:

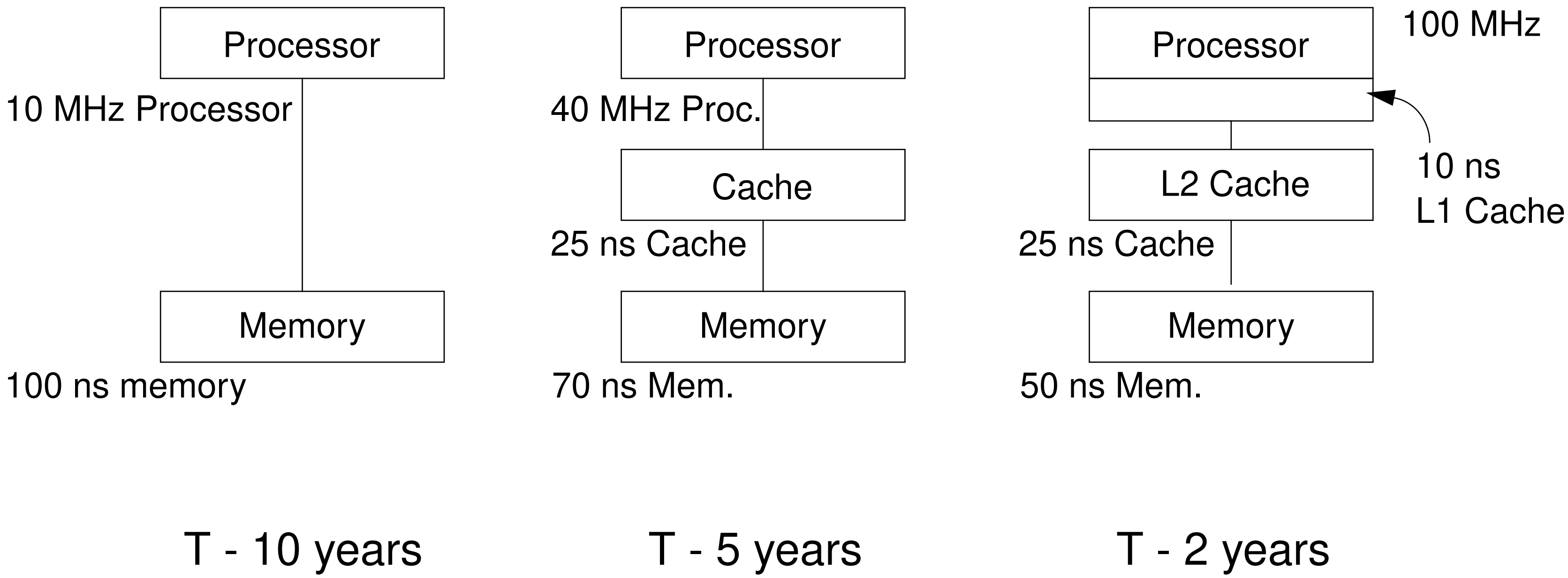
Time from request for the data at the memory to when the data is actually available.

Bandwidth:

Consider two memory modules each 8 bits wide and latency l. Each gets a memory request at time t. At time $t+l$, we have $8 + 8 = 16$ bits available. We have therefore doubled the amount of data that we can pump out of the memory system. The volume of data that can be pumped out of the memory system in a second is referred to as the bandwidth.

Bandwidth of a memory system is limited by number of memory banks, data-carrying capacity of the wire connecting the memory to the processor (memory bus bandwidth).

Evolution of Computer Architectures (Memory System):



CS590C: Introduction to Computational Science

Lecture 2: Evolution of Computer Hardware.

This lecture is intended to:

- ❏ Review basics of computer architectures (both serial and parallel)
- ❏ Familiarize you with commonly used terms
- ❏ Discuss performance related issues
- ❏ Briefly discuss algorithmic issues as they relate to performance tuning.