SFWRENG 4F03
Parallel Computing

Winter 2020

03 Parallel Hardware

Dr Asghar Bokhari

Faculty of Engineering, McMaster University

January 30, 2020

Note Taker Required

- Please contact SAS, if interested.
- More info at:
  https://sas.mcmaster.ca/volunteer-notetaking/

The von Neuman Architecture

- Writing efficient Parallel programs requires some understanding of the underlying hardware and software
- Parallel hardware and software grown out of serial hardware and software
- Parallel systems consist of a number of single processors
- There is a possibility of parallelism within single processors
- RISC processors commonly used in parallel architectures - review some of the related details
- Should know the memory and cache arrangements for single processors before extending the ideas to multiple processors

Basic Computer System

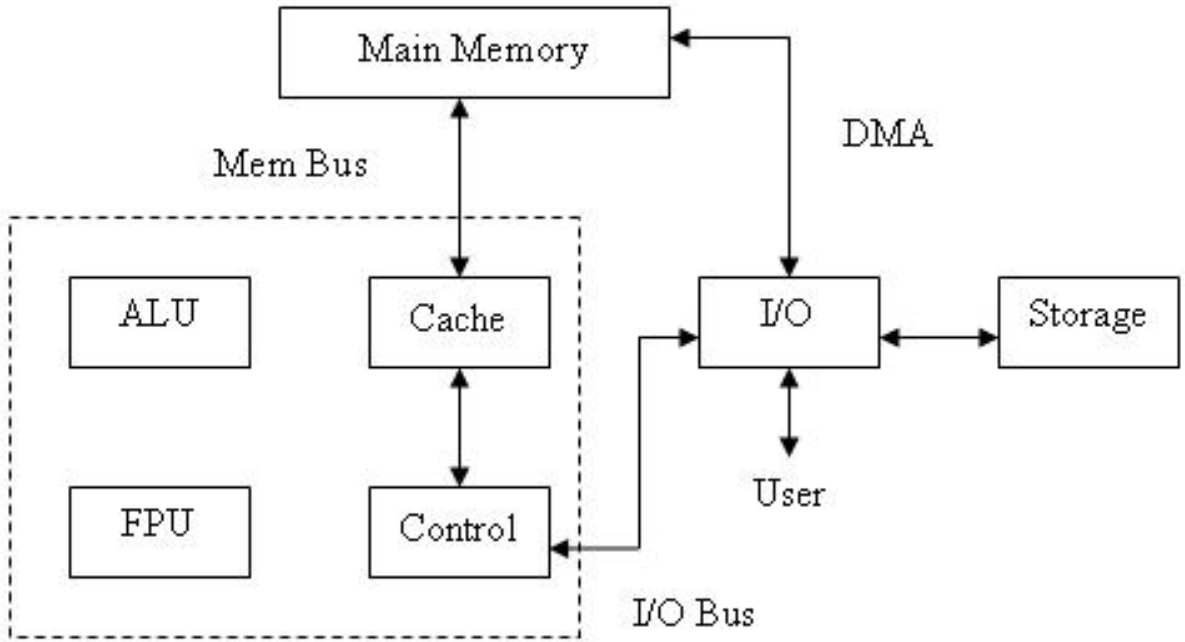


Figure: Basic Computer System

Stored-program Machine

- Basic idea of von Neuman architecture: Store instructions as well as data in the memory and let CPU execute instructions one by one.

- Central Processing Unit (CPU)
  - CPU consists of millions of electronic circuits.
  - Processes instructions.
  - In today's PCs the CPU is contained in a single silicon chip called a microprocessor chip
    Examples of microprocessor chips makers
    - Intel - Pentium IV
    - Motorola 68070
    - Cyrix
    - AMD Athlon 64
    - ARM

CPU

- CPU consists of three primary units:
  - the control unit.
  - the arithmetic/logic unit
  - a set of registers (temporary storage)
    (Some describe the CPU as 4 units by viewing the arithmetic/logic unit as two segments.)

CPU

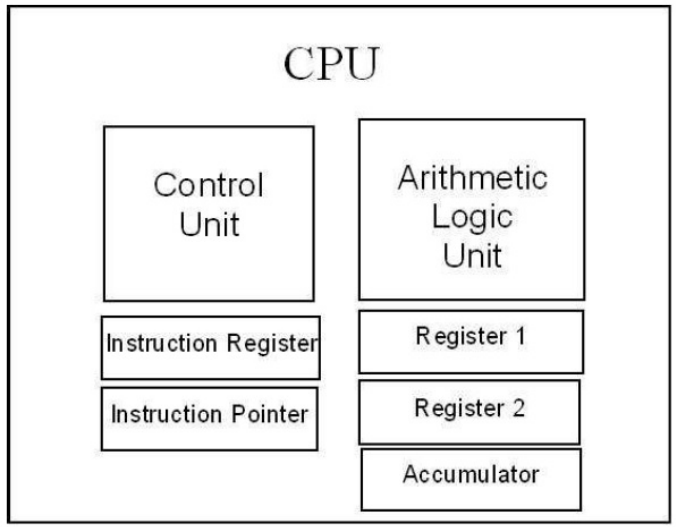


Figure: CPU

CPU

- Role:
  - to control and to execute instructions for logic and arithmetic operations
  - to manipulate data
  - to manage memory
  - to manage the input/output devices
- Control Unit
  - Manages all resources of a computer
  - An instruction consists of steps required to do a primitive task like, add, read, store
  - An instruction set lists all primitive tasks that a CPU can perform
  - Instructions are generally built into control unit as microcodes

CPU

- Arithmetic / Logic Unit (ALU):
  - performs simple (integer) arithmetic operations (addition, subtraction, etc.)
  - performs simple logical operations (such as the comparison of two numbers then this can result in the selection of subsequent operations)
  - The circuits within the arithmetic/logic unit are created from elementary components called gates with one or more inputs and one output valued 0 or 1
- Registers:
  - High speed memory normally used to hold only a few bytes of data that are currently being processed by CPU
  - Located within the CPU
  - Size of registers - word size (16, 32, 64)

CPU

- Some special registers:
  - instruction pointer: which indicates the location (address) of the next instruction to be executed
  - instruction register: which contains the instruction currently being executed
  - accumulator: register assigned to store the result of operations performed by the ALU

Basic Working of CPU

- Machine Cycle (Fetch, Decode, Execute)
  - Series of operations performed by CPU to complete an instruction
  - Each byte in memory has a unique address
  - Starting address in the instruction pointer register
  - Fetch: the first operation in machine cycle –retrieves from memory an instruction, (based on the current instruction pointer value)
  - decode: determines tasks defined by the instruction and interpreted based on the computers instruction set
  - execute: the instruction may include getting data from memory and copying it into a register arithmetic or logical operations copying contents of accumulator into memory (store)
  - Finally, update the instruction pointer to the address of next instruction

Bottleneck

- Separation of memory and CPU in von Neuman architecture
- Interconnect determines tha rate at which data and instructions canbe accessed
- Interconnect a bottleneck when a large quantity of instructions and data required by modern CPUs
- CPUs are able to handle instructions more than 100 times faster than they can fetch items from main memory.

Dominant Architectures

- CISC - Complex instruction set computer
  - Examples Motorola 68000, Intel x86, VAX 8600
  - Typical set may consist of 120 to 350 instructions with variable formats
  - 8 to 24 general purpose registers
  - Has a large number of memory reference operations based on more than a dozen addressing modes
  - Was the best choice in 70s
  - Memory was scarce and programmers had to save time and space by programming in assembly language
  - High level primitives were easy to work with
  - Powerful instructions could perform multiple tasks without the programmer specifying all small steps
  - Less memory required by lesser instructions

Dominant Architectures

- CISC - Complex instruction set computer .....
  - Loaded faster as less instructions to load from memory that was slow
  - Programs were easier to debug
  - Optimizing compilers did not use complex instructions
  - It was realized that only 25% of the instructions were used 95% of the time
  - Why waste valuable chip area for rarely used instructions?
  - Not suitable for instruction level parallelism.

Dominant Architectures

- RISC - Reduced instruction set compute
  - Examples: Motorola M 88100, Sun SPARC, Intel i860, AMD 29000, MIPS, PowerPC
  - Typical RISC instruction set contains less than 100 instructions
  - Has a fixed instruction format (32 bit)
  - Only three to five simple addressing modes
  - Memory access by load/store instructions only
  - A large register file (at least 32 registers)
  - Most instructions execute in one clock cycle with hard wired control
  - Although started with simple instructions, more complex instructions added and now complexity similar to CISC
  - Complexity also moved from insrtuction set to software. A good optimising compiler required for fast code generation

Clock Rate and CPI

- Digital computers driven by a clock with a constant cycle time or period τ
- Inverse of cycle time is the clock rate $f = 1/\tau$ (MHz or GHz)
- Size of a program is determined by the count of machine instructions executed in the program
- Time required to execute one instruction - CPI (cycles per instruction)
- Different instructions may take different number of clock cycles to execute
- We can calculate an average CPI over all instruction types - requires a large amount of code traced over long period of time
- Generally CPI assumed to be average value w.r.t a given instruction set and a given program mix

MIPS Rate

- The processor speed is often measured in millions of instructions per second (MIPS), ususally called the MIPS rate.

MIPS rate $= \frac{I_c}{T \times 10^6}$ Where I_c is the number of instructios executed in time T

$$T = I_c \times CPI \times \tau$$

So MIPS rate is given by:

$$= \frac{I_c}{I_c \times CPI \times \tau}$$

$$= \frac{f}{CPI \times 10^6}$$

MIPS Rate

If C is the total number of clock cycles to execute the program

$$CPI = C/I_c$$

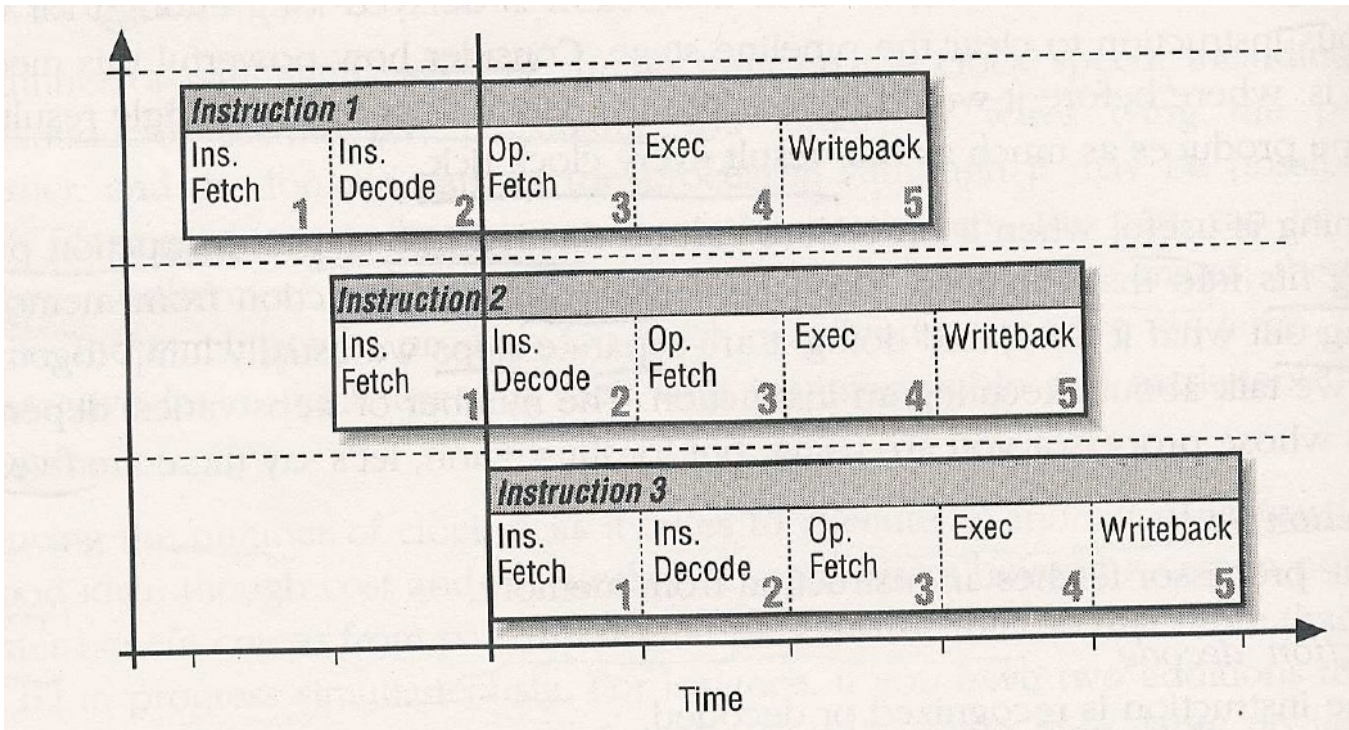
Substituting in above eqn. MIPS rate is:

$$= \frac{f \times I_c}{C \times 10^6}$$

- MIPS rate does not take into account the I/O speed and architecture of processor.

Instruction Level Parallelism

- Scaler processor: only one instruction is issued per cycle
- Performance can be improved by **Pipelining** - overlapping of instructions



[1]

[1] K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Pipelining

- A potential speed up equals n times the speed without pipeline, where n is the stages in the pipeline
- After the first five cycles, one instuction is executed each clock cycle
- 3 instructions in 7 cycles - $CPI = 7/3 = 2.3$ instead of 5
- However if the number of instructions is increased to say 10000, the total cycles $= 10005$
- and $CPI = 10005/10000 \approx 1$ representing a speed up of 5 times
- Ideally a linear pipeline of k stages can handle n instructions in $k + (n - 1)$ cycles where k cycles are required to complete the very first instruction.
- Total time required is : $T_k = [k + (n - 1)]\tau$ where τ is the clock period.

Pipelining

- An equivalent non-pipelined processor will need $T_1 = nk\tau$ time to execute the same instructions
- Speedup factor S_k for this ideal pipeline, is given by:

$$S_k = \frac{nk\tau}{[k + (n-1)]\tau} = \frac{nk}{k + (n-1)}$$

- Pipeline reduces latency not the execution time

Problems:

- Data Hazards: An instruction j is data dependent on instruction i if:
  - Instruction j uses the result produced by instruction i or
  - Instruction j uses data produced by instruction k and instruction k is data dependent on instruction i

Pipelining

```
sub $2, $1, $3 //Register 2 contains
       //difference of registers 1-3
and $12, $2, $5 // $2 depends on sub
or  $13, $6, $2 // $2 depends on sub
add $14, $2, $2 // both operands depend on sub
sw  $15, 100($2)// Base $2  depends on sub
```

The four instructions after sub depend on the correct result in register $2. If it contained 10 before sub and 5 after, these instructions should use 5 not 10. Can avoid situation by forwarding results from pipeline registers or by inserting 2 independent instructions or noops between sub and add or both

Hazards

- Data Hazards:
  - RAW (read after write): j tries to read a source before i writes to it - j gets old value
  - WAW (write after write): j writes an operand before i does so - wrong order
  - WAR (write after read) j tries to write a location before i reads it
- Control Hazards
  - Caused by branch instructions
  - Processor does not know whether the branch will be taken or not until the instruction is decoded
  - If branch is taken, pipeline must be flushed as it contains incorrect result.

Example

```
if (A == B)
{ S_1;
  S_2;
  S_3;
}
else
 S_4;
```

Hazards

- Struxtural Hazards:
  - Hardware cannot support requirements of instructions in the pipeline.
  - Example: 4 instructions need floating point units, only 2 available.

Avoiding Hazards

- Forward results to ALU from pipeline registers
- Dependancy detection and rearrangement of instructions sometime adding no-ops
  For control hazards:
- Stall until branch instruction is complete - too slow
- Assume branch not taken - may reduce cost by half if branches are not taken 50% of time
- Delayed branch - add an instruction after branch that will be executed whether or not the branch is taken.
- Branch prediction

Speed Up

- Pipelining increases instruction throughput of a CPU not the time for execution on an instruction
- Under ideal conditions, time per instruction on pipelined processor is:
  = (Time per instruction on un-pipelined machine)/(Number of stages in the pipeline)
- Considering different overheads
  = (Average time taken by an instruction un-pipelined)/(Average time taken by an instruction pipelined)
  where average instruction time
  = Time for one clock cycle * Average CPI

Effect of Stalls

- Ideal CPI on a pipeline processor is almost always = 1,
- The above equation can be written as:
  speedup = (CPI unpipelined)/(CPI Pipelined) *
  (Clock cycle time unpipelined)/(Clock cycle time pipelined)
- To take care of the effect of stalls
  CPI pipelined = (Ideal CPI + Pipeline stall clock cycles per instruction)
- Ignoring the cycle time overhead of pipelining and assuming the stages perfectly balanced, cycle time of two processor is equal. This leads to:
  speedup = (CPI unpipelined)/(1 + pipeline stall cycles per instruction)
- If all instructions are assumed to take the same cycle time equal to number of pipeline stages called pipeline depth
  speedup = (Pipeline depth)/(1 + pipeline stall cycles per instruction)

Further Developments

Superpipelining: MIPS R4000, 4300

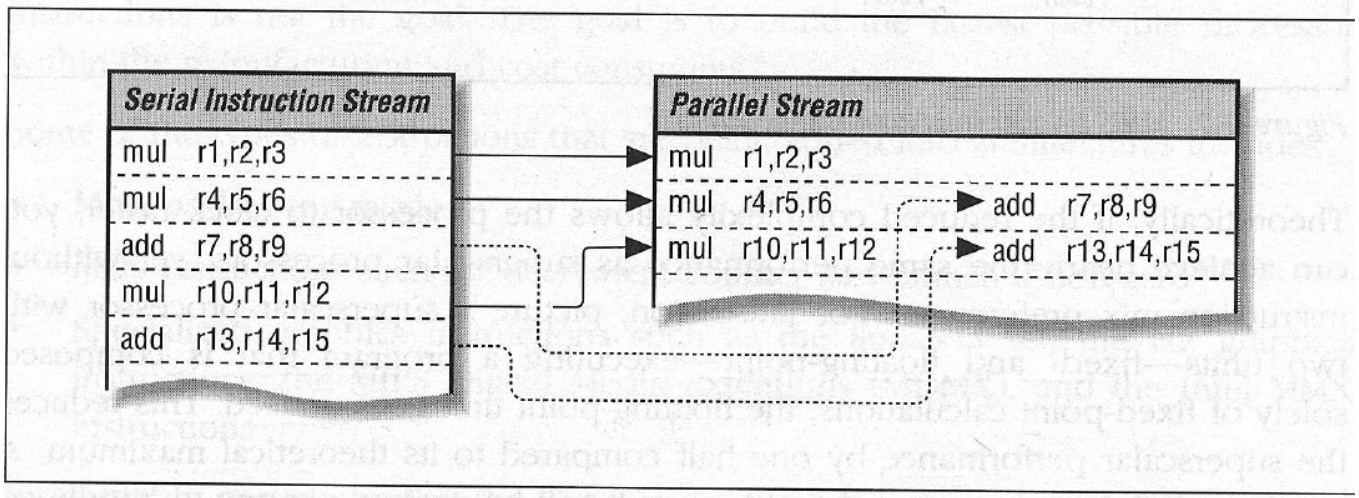
- If speed up is proportional to pipeline depth, some architectures tried pipeline depths
- greater than five such as 8 in case of above two processors.
- Increased penalty for stalls
- Need good compilers

Further Developments

Superpscaler Processors

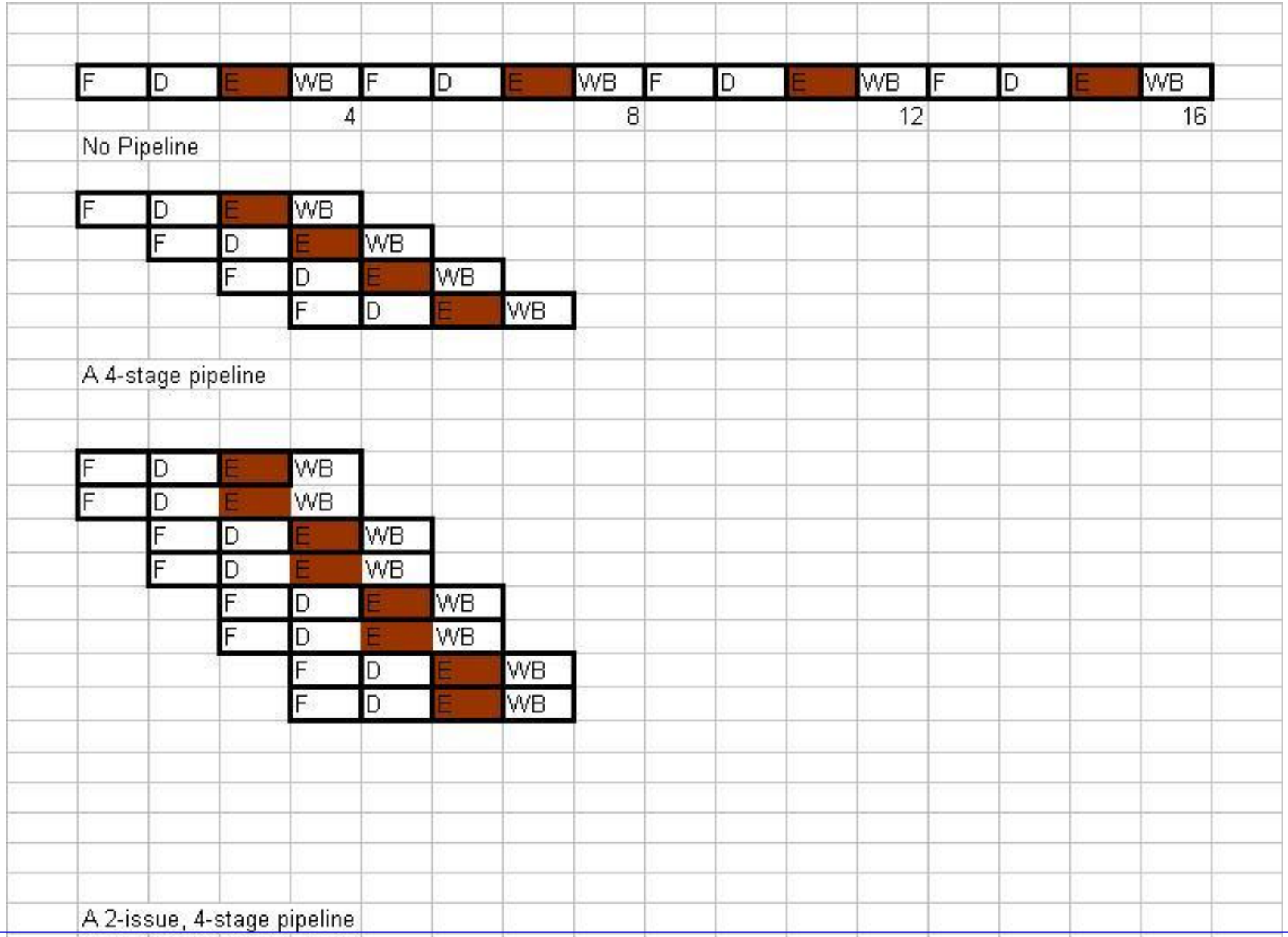
- Goal is to issue multiple instructions in each clock cycle
- Duplicate compute elements on the space available due to increase chip density
- Another term used is *multiple issue processor*
- The number and variety of operations that run in parallel depends on both the program and the processor
- Program has to have enough usable parallelism and
- processor must have appropriate number of functional units and be able to keep them busy

Further Developments



Decomposing a serial stream

Comparasion



Comparison

- Consider the ideal execution of N independent instructions.
- The time (in cycles) required by the scaler processor is:

$$T_s = k + N - 1$$

- The time required by an m issue superscaler is:

$$T_m = k + \frac{N - m}{m}$$

where k is the time required to execute the first m instructions through the pipelines simultaneously and the remaining $N - m$ instructions are executed m per cycle.

- The ideal speedup for an m-issue processor is:

$$S_m = \frac{T_s}{T_m} = \frac{N + k - 1}{N/m + k - 1} = \frac{m(N + k - 1)}{N + m(k - 1)}$$

Next Steps

- Two-way superscalar processors were achieving about 1.6-1.8 instructions per cycle.
- Next step would naturally be four-way or eight-way, right?
- The basic problem is to find out contiguous sets of m instructions that can be executed in parallel without stalls.
- Need to keep four (eight) units busy. Hard to find four (eight) sets of instructions in a row to execute in parallel.
- Solution:
  - *Out of order execution* and *speculative computation.*
  - If the processor could not find enough instructions in the sequential stream, it looks ahead for instructions that don't rely on previous ones. It then computes them in advance to take up unused resources.

Example

```
ld r1, r2(r0) load into r1 from memory
        ........ instructions of various kinds
        fdiv r4, r5, r6  r4= r5/r6
```

- Assumptions:
  - we are executing ld instruction
  - r5, r6 already loaded by previous instructions
  - No instruction between ld and fdiv need the FP divide unit
- Why not start using the divide unit to compute fdiv now?
- But, what if that instruction would never get executed?

Further Developments

Speculative Execution

- Must separate the idea of *computing* a value from *executing* an instruction.
- to make use of unused resources, processor computes the value of instructions that it thinks it may need, particularly if they are a slow operation. The results are then stored in an internal, hidden, register.
- If a branch occurs, and the instruction isn't needed, the results are discarded.
- The number and variety of operations that run in parallel depends on both the program and the processor
- Program has to have enough usable parallelism and
- processor must have appropriate number of functional units and be able to keep them busy

Overview of Memory Hierarchy

RAM

- Von Neuman architecture separated the memory from processor
- Infinitely fast processor still needs to load/store data and instructions from/to memory
- Need to have matching performance between different players
- Remember the bottleneck!
- Two main types of RAM:
  - DRAM- need refreshing (charge leak, loss of charge due to read), cheaper, capacitor based, slow
  - SRAM based on technology similar to that of CPU is able to keep up with CPU speed (approx 1 ns)
  - SRAM- no refreshing, expensive, gate based, fast, generates more heat, large sizes may need coling

Memory

- Example:
  - Cray-1 super computer used SRAM as main memory that was able to keep up with 12.5ns clock cycle - required liquid cooling!
  - Not practical to manufacture inexpensive systems using SRAM
- Solution:
  - Hierarchy of memories
  - Registers, SRAM caches, DRAM main memory, virtual memory using storage media
- Memory Access time: Amount of time it takes to read or write a memory location
- Memory cycle time: time interval between two successive memory accesses.Chip needs time to recover from earlier access

Registers and Caches

- Registers:
  - Top layer of the hierarchy (speed usually 1 cpu cycle)
  - Hold operands during excution of instructions
  - Why not load all data into registers ? cost, space on CPU chip
- Caches:
  - small ammounts of SRAM that store a subset of the main memory
  - prosessors are now so fast that off-chip SRAM not fast enough (overhead in moving data over wires)
  - multi level cache approach, one or two levels of caches implemented as part of the prossesor

The Basics of Cashing

- **How does cache improve performance?**
  - Can all memory locations that a program needs to access be loaded into cache?
  - A subset of memory is copied into cache. when CPU needs to access adresses in this subset, it can access cache leading to performance improvement
  - How to decide which adresses to load in to cache?
  - How it is done ?
- **Spatial and temporal locality:**
  - Programs often make use of instructions and data that are near to other instructions and data both in space and time.
  - Cache memory is divided into a number of equal size blocks sometimes known as lines
  - Each line can hold a number of words and usually contain contents of a number of sequential memory locations

The Basics of Cashing

- Spatial and temporal locality continued
  - When the processor needs to access a particular memory location it first checks the cache to determine if the contents of that address exist in cache
  - If not it is termed as a **cache miss**
  - If it finds the contents in cache it is caleld a **cache hit**
  - In case of a hit, data is returned with a minimal delay; in case of a miss a whole line is retrived from main memory starting from the address that caused the miss
  - Chances are that the additional information will be required by the program soon and then it will be availible in cache.

Updating Memory

- When data are stored into cache there must be some mechanism to update it in the main memory
- **write-back cache:**
  - Data stays in cache untill the cache line is replaced at which point it is written back to memory
- **write through cache:**
  - Data are simultaniously written into cache and memory; in this case when a slot is required to be replaced the previous contents are simply discarded

Unit Stride

- When a program requires data from succesive locations whose adresses can be obtiained by incrementing the previous adress by 1, it is called a **unit stride**; data from all locations of a line in cache are used resulting efficent utilization of cache
- When a program accesses data using a non-unit stride, performance suffers as data are loaded into cache that are not used

Cache Mappings

- Process of pairing memory locations with cache lines is called mapping
- A particular cache line shared with different memory locations (Cache much smaller than RAM!)
- Tags used for indicating memory locations mapped into cache lines and when it was last used
- Three different ways to organize caches
- **Direct mapped**
  - Each memory location mapped to exactly one location in the cache
  - cache line = memory block address mod Number of lines in the cache
  - Example: 8 lines cache - mem location 22 will map to line 6, and line 16 to 0

Cache Mappings

- **Fully associative**
  - Any memory location can be mapped into a cache line
  - When cpu need data, all entries in the cache must be searched
  - Hardware comparators used to facilitate the search
  - Too costly
- **Set associative**
  - Fixed number of locations (atleast two) where a block of memory can be placed
  - Each block is directly mapped into the set of cache locations; then all the blocks in the set needs to be searched to find a match

Caches and Programs

- Programmers don't determine which instruction/data in the cache
- Knowing that it follows the principle of spacial and temporal locality helps
- C stores a two dimentional array in "row-major" order
- The first pair of for loops (in the next slide) will run faster

Caches and Programs

```
double A[MAX][MAX], x[MAX], y[MAX];
/* Initialize A and x. Assign y = 0*/
.....
/* First pair of loops*/
for (i=0; i<MAX; i++)
    for (j=0; j<MAX; j++)
        y[i] += A[i][j]*x[j];

.....
/* Second pair of loops*/
for j=0; j<MAX; j++)
    for (i=0; i<MAX; i++)
        y[i] += A[i][j]*x[j];
```

Virtual Memory

- All of the instructions and data of a large program may not fit into physical memeory
- Multitasking needs to load several programs into physical memory.
- VM allows one to use a large address space even when those addresses are not physically available in actual memory
- The entire address space is organized into pages and if enough physical memory is not available, extra pages are stored on hard disk
- Main memory acts like a cache for the secondary storage.

Virtual Memory

- Pages that are not required immediately are put in **swap space**
- Pages belonging to different programs may have the same page number in virtual space
- Page table used to translate virtual address to physical address.
- **Translation-lookaside buffer (TLB)** used for fast translation
- TLB hit, TLB miss and page faults.

Parallel Hardware

- Multiple issue and pipelining are parallel harware but programmer has no control.
- We now start discussing the hadware that is visible to a programmer
- He/She can modify the source code to exploit the hardware

Flynn's Taxonomy

- Based on instructions and data streams called for by instructions
- SISD (single instruction single data) - this is the uniprocessor
- SIMD (single instruction multiple data) - for example, vector processor
- MISD (multiple instructions single data) - No commercial system of this type has been built as yet
- MIMD (multi instructions multiple data) - each processor fetches its own instructions and operates on its own data
  - Shared memory multiprocessors UMA, NUMA, COMA, ccNUMA
  - Distributed memory multicomputers

SIMD Systems

- Operate on multiple data streams by applying the same instruction to multiple data items
- An abstract SIMD machine can be thought of as having a single control unit and multiple ALUs
- Instruction is broadcast from the control unit to all ALUs
- Each ALU applies the instruction to current data it has or is idle (if no data)
- Consider the addition of two vectors:

```
for (i=0; i<n; i++)
    x[i] += y[i];
```

- Assuming n ALUs, ith ALU will add y[i] to x[i] and store result in x[i]

SIMD Systems

- If number of ALUs is $m < n$ addition can be performed in blocks of m elements at a time
- Parallelism is achieved in SIMD sytems by dividing data among the available processsors
- All processors apply the same instruction to their subsets of data **Data Parallelism**
- Very efficient on large data parallel problems
- May not perform well on othe types of problems
- Example: Vector Processor

Vector Processor

- Key characteristic: operation on arrays or vectors of data instead of data elements
- A vector machine consists of a vector processor attached to a scaler processor
- Data are loaded into memory by the host scaler processor
- Instructions are also decoded by the control unit of the scaler processor
- If an instruction contains a scaler operation, it is carried out using functional pipelines of the scaler processor
- If a vector operation is called for, it is sent the vector control unit
- The vector control unit coordinates the vector data flow from memory to vector functional pipelines
- A number of vector pipelines may be built into a vector processor

Example

```
 VADD.W   V1, V2, V3  where
 V1, V2, V3 are all 64 element
 vectors of 64-bit floating
 point numbers.
 A single vector instruction
 performs all the work - fewer
 instruction fetches in general,
 fewer branch instructions
 and so fewer mispredicted
 branches.
This equivalent to:
DO I = 1, 64
   V3 = V1(I) + V2(I)
ENDDO
```

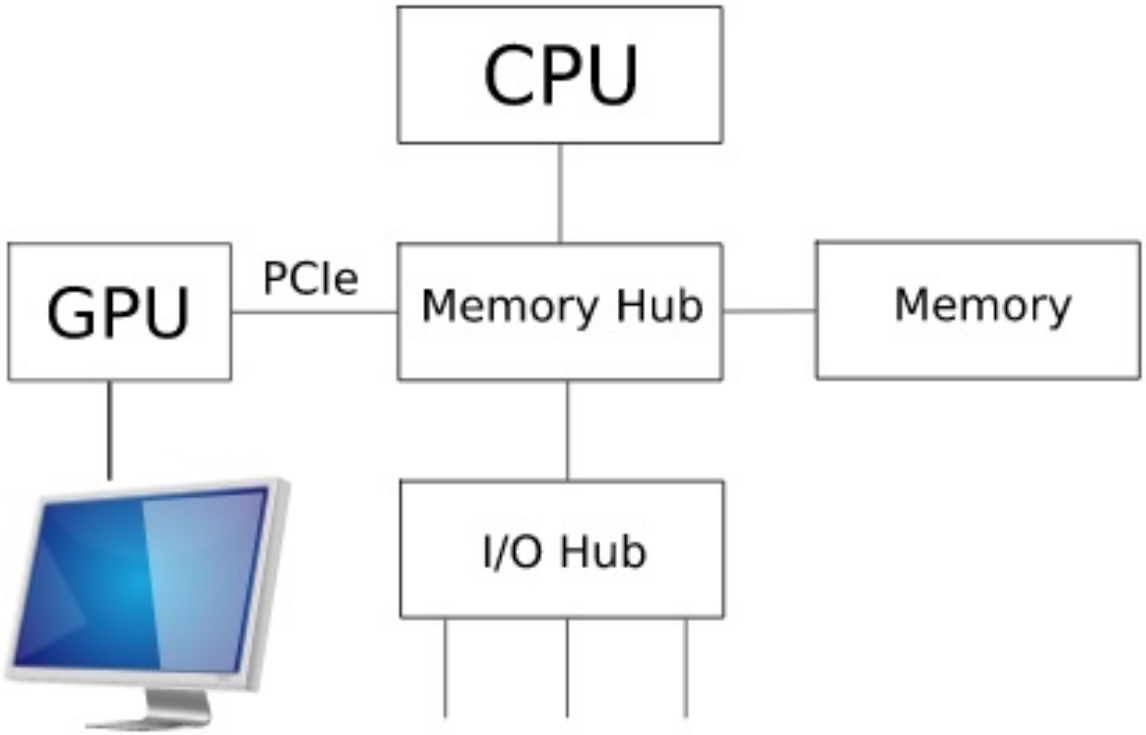
Two Models

- Register-to-register Architecture:
  - Ueses vector registers to hold vector operands and results
  - Length of each vector register is fixed, such as sixty-four 64-bit components in a Cray series supercomputer
  - Multiple memory pipelines are used to load operands from memory to registers
- Memory-to-memory Archtecture
  - Vector operands are directly retrieved from memory to the functional units and the results directly stored in memory
  - Cyber 205 is an example of this type
  - Suitable for more specific problems only
- More recently graphics processing units (GPUs) are making use of aspects of SIMD computing

GPU- Graphics Processing Unit

- A specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.
- Used in embedded systems, mobile phones, personal computers, workstations, and game consoles.
- Modern GPUs are very efficient at manipulating computer graphics and image processing.
- In a personal computer, a GPU can be present on a video card or embedded on the motherboard.

Position of GPU in a PC



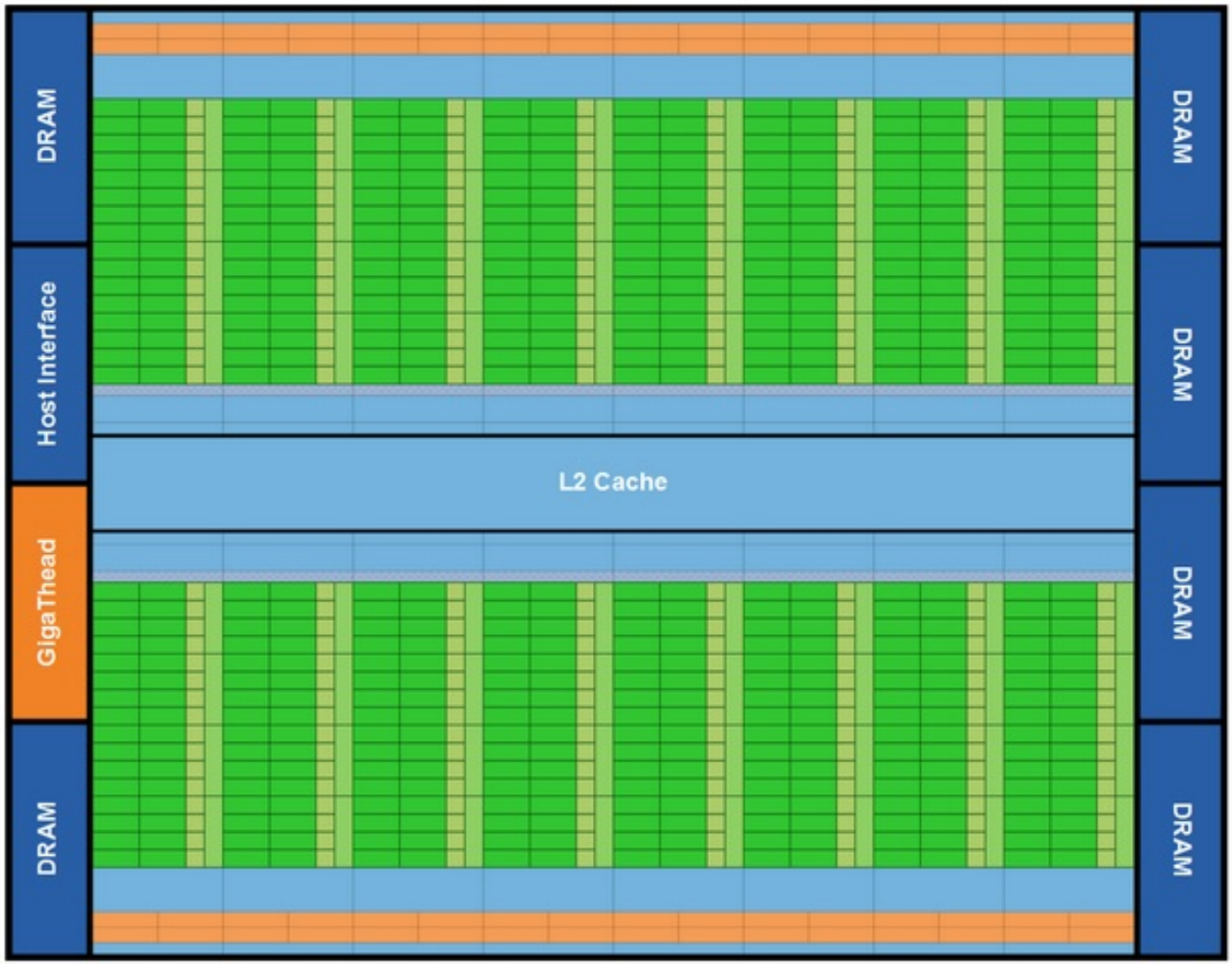
GPGPU

- The highly parallel structure of GPUs makes them more efficient than general-purpose central processing units (CPUs) for algorithms that process large blocks of data in parallel.
- They are generally suited to high-throughput type computations that exhibit data-parallelism to exploit the wide vector width SIMD architecture of the GPU
- GPUs have ignited a worldwide AI boom. They've become a key part of modern supercomputing.
- In AI, GPUs have become key to a technology called deep learning
- CPUs race through a series of tasks requiring lots of interactivity.
- GPUs break complex problems into thousands or millions of separate tasks and work them out at once.

GPGPU- NVIDIA Fermi

- Fermi - the codename for a graphics processing unit (GPU), first released by Nvidia to retail in April 2010
  - Successor: Kepler
  - Predecessor: Tesla 2.0
  - 16 Streaming Multiprocessor(SM)
  - GigaThread global scheduler: distributes thread blocks to SM thread schedulers and manages the context switches between threads during execution

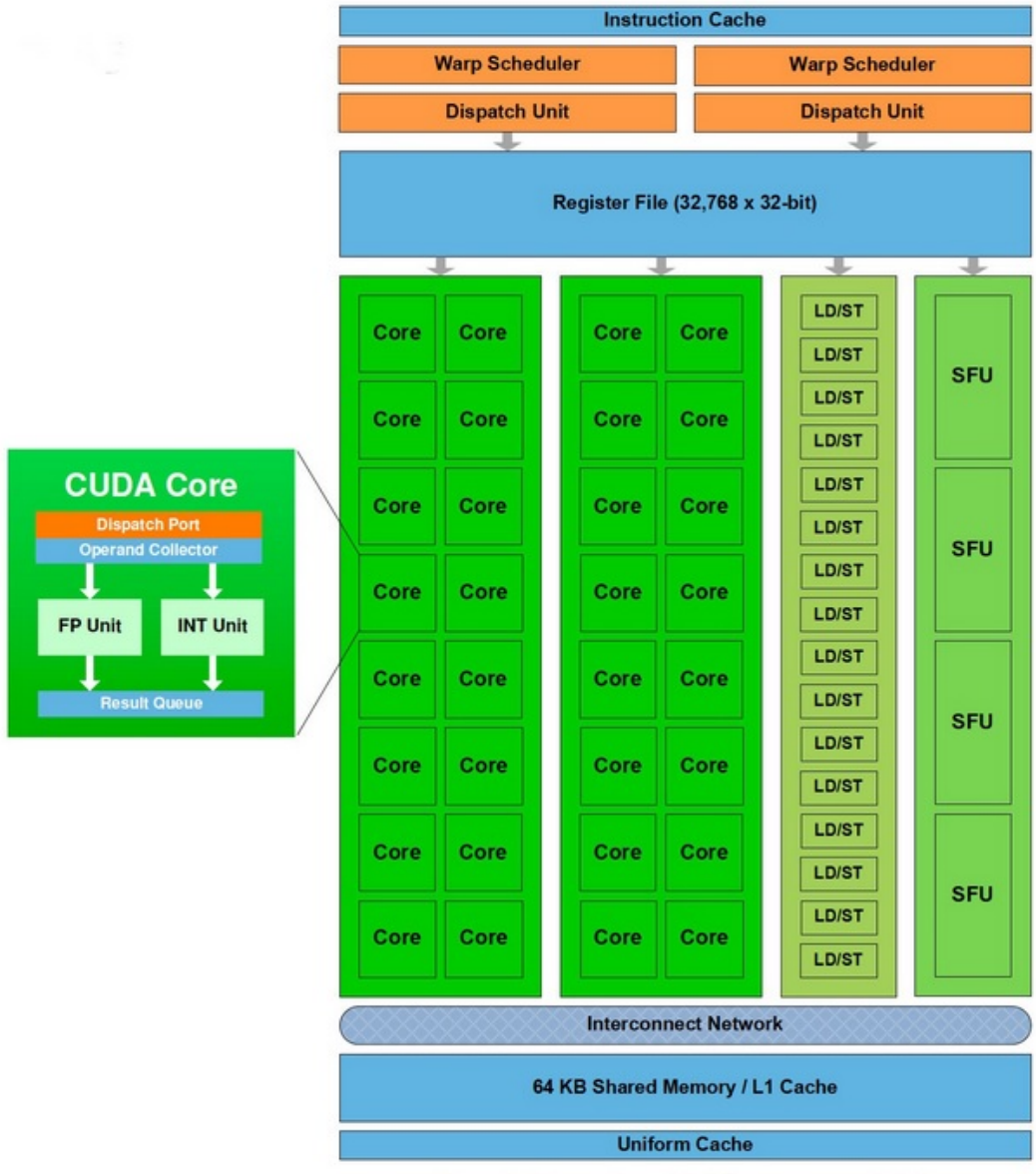
Fermi Architecture



Streaming Multiprocessor (SM)

- Each SM has 32 CUDA cores
- Every core is an execute unit for integer and floating point numbers
- The SM schedules threads in group of 32 threads called warps.
- Two warp schedulers and two dispatch units in each SM
- Load/store units: Load and store the data from/to cache or DRAM
- Special Functions Units (SFUs): Execute instructions such as sin, cosine, reciprocal, and square root

SM



Fermi Streaming Multiprocessor (SM)

MIMD Systems

- Multiprocessors and Multicomputers
  - Multiprocessors are tightly coupled systems due to high degree of resource sharing
  - Need special interconnect such as a bus or crossbar switch to be discussed later
  - A multicomputer system consists of a number of computers, sometimes called nodes, interconnected by a message passing network.

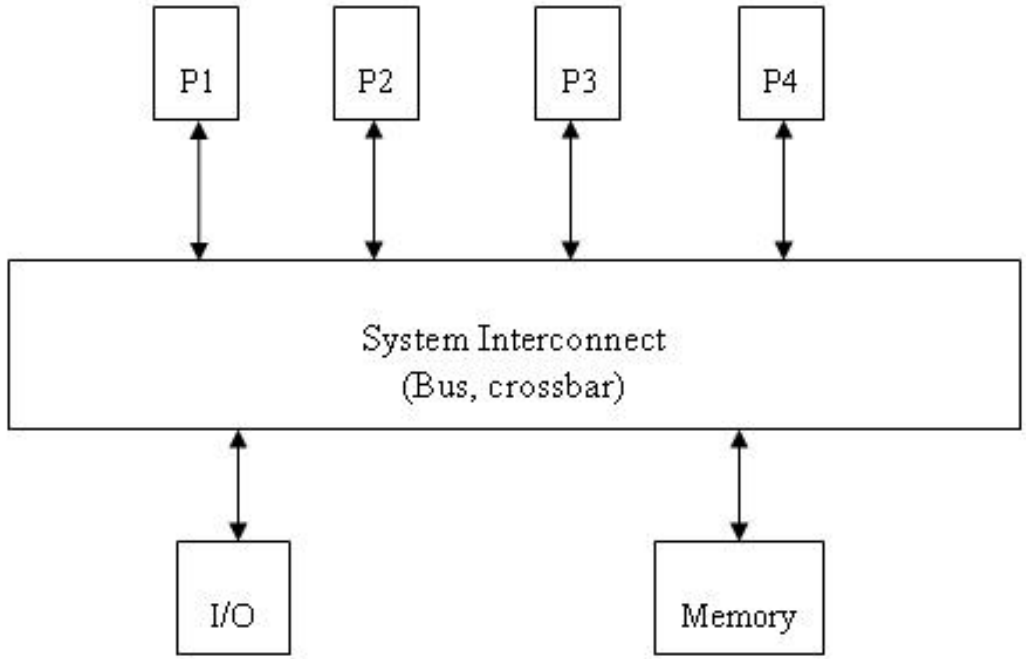
Multiprocessors

- More than one processors in the same physical box
- Now more than one processors on the same chip
- Typically workstations have 1 to 4 processors and servers may have 4 to 64 processors
- Also called shared memory multiprocessors
- Three different arrangements

UMA (Uniform Memory Access)

- Also called SMPs (symmetric multiprocessors)
- All processors share the same memory address space
- Memory is equally accessible to all processors with the same performance
- All processors have equal access to all peripheral devices
- Excellent at providing high throughput - databases, network/internet servers
- Hardware support required for arbitration in order to decide who will have access to what.
- Bus is the simplest interconnect and is less expensive
- Becomes a bottleneck at high loads

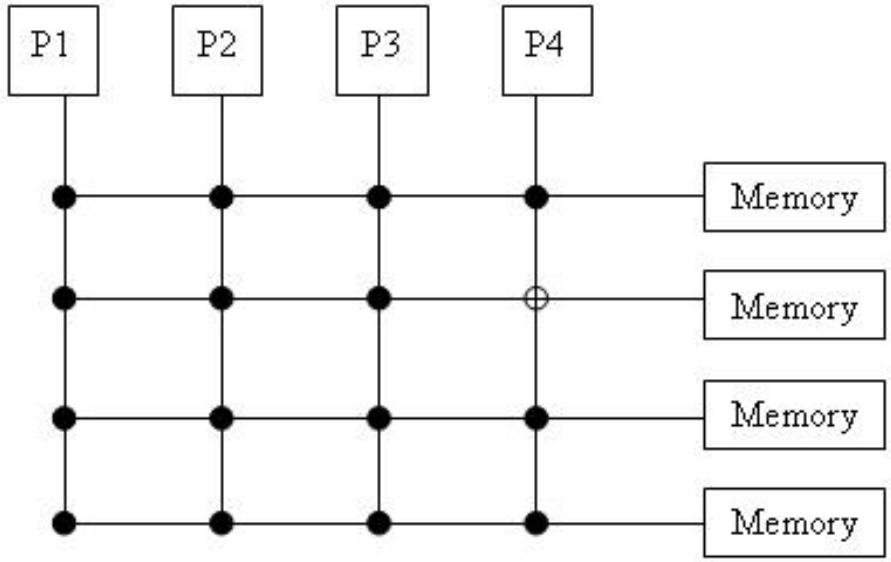
UMA (Uniform Memory Access)



UMA (Uniform Memory Access)

- Crossbar more suitable for high loads
- Arbitrates between two or more processors that want to access the sam memory or I/O device
- Each crosspoint switch can provide a dedicated switch between a pair
- Because of their cost crossbars are used only in high end systems

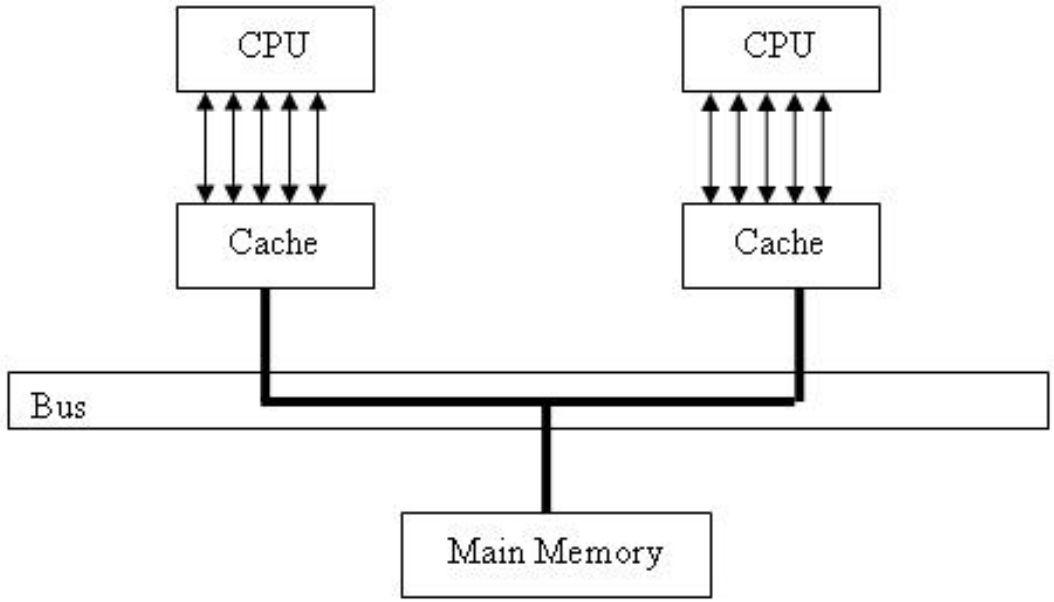
UMA with Cross Bars



Effect of Cache

- Effect of bandwidth bottleneck reduced
- Most of memory access traffice consists of cache lines loads and flushes
- Consider the system in figure
- Both CPUs access the first element in a cache line at the same time and cause cache miss
- The bus arbitrarily allows one CPU to fill its cache line from memory and start processing
- The second CPU is allowed to fill its cache line as soon as the first CPU has filled its cache line and then this CPU can also start processing data in its cache line

Effect of Cache



Effect of Cache ...

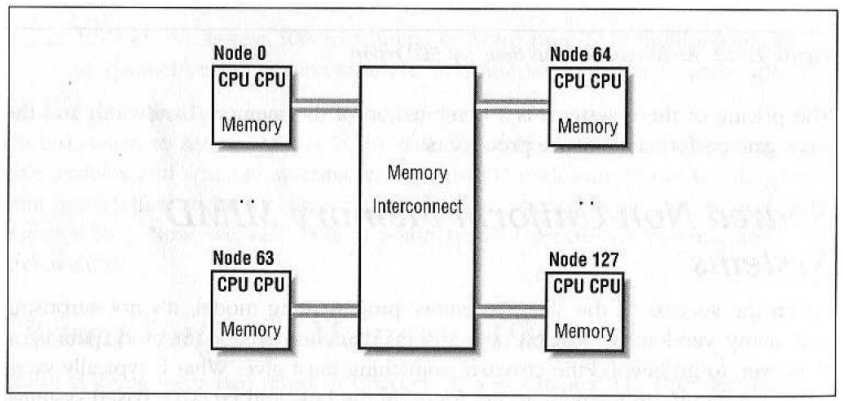
- Assume the time required to process data is more than the time to fill a cache line
- Both CPU can have conflict-free access to memory after the initial conflict is resolved.
- In practice, up to 20 processors can access memory using unit stride with little conflict.
- For non-unit stride accesses, the conflict become apparent with fewer processors.
- Sometimes a two-tear cache system is used on-chip local cache and a large board-level cache (4MB) to further improve performance

How is cache coherency achieved?

- Assume a multiprocessor system is running a single program in parallel
- One of the processors changes a shared variable and another tried to read it.
- Where does the value come from?
- Where does the first CPU store the changed value?
- In cache or memory?
- Does the cache of second CPU get updated?
- Depends on the cache coherence protocol used by the vendor of the system
- Write-through policy with snooping
- Writeback policy where hardware monitors and responds to memory transactions of the other caches in the system
- Possible states of cache lines: Modified, Exclusive, Shared, Invalid/Empty (MESI protocol)
- Write back considerably reduces memory traffic

NUMA(Non-uniform memory access)

- Every processor has a memory located close to it, called local memory
- The collection of all local memories forms a global address space available to all processors
- Access to local memory by a local processor is faster than access to a remote memory due to delay in the interconnect- (non-uniform part)
- It is possible to have more than one CPUs sharing local memories



NUMA(Non-uniform memory access) continued...

- Each node can even consist of a UMA cluster
- In some cases a golbal shared memory is included in addition to local memories.
- Access to local memory is fastest followed by access to global shared memory, followed by access to remote memory

ccNUMA (Cache-coherent non-uniform memory access)

- Some systems implement a cache coherency protocol across the interconnect
- Advantage is that programmers, operating systems and applications are used to working with cache coherent systems
- Cache coherency is much simpler in UMA systems using bus or crossbars:
  1. In a bus system all caches are connected to the same bus and can snoop the bus and see every transaction regardless of which processor initiated it.
  2. A crossbar broadcasts its cache coherency operations to all ports of the crossbars to notify all of the caches

ccNUMA (Cache-coherent non-uniform memory access) continued ...

- In NUMA systems that may have 64 or more processors and local memories with 128 or more caches broadcasting every cache operation to all these components is not practical.
- Directory based caching
- Processors can find out which other processors are sharing a particular piece of data by going through a list.
- Cache coherency operations are then performed only between the processors on that list.

COMA (Cache only memory architecture)

- A special case of NUMA machines
- There is no memory hierachy at each processor node instead each processor has only local caches available to them. All caches form a global address space.
- If a processor need data not in its local cache, it retrives data from other caches using a directory system. Instead of the data moving from cache to memory and then to another cache, it migrates from one cache to another and stays with it until another processor needs it.

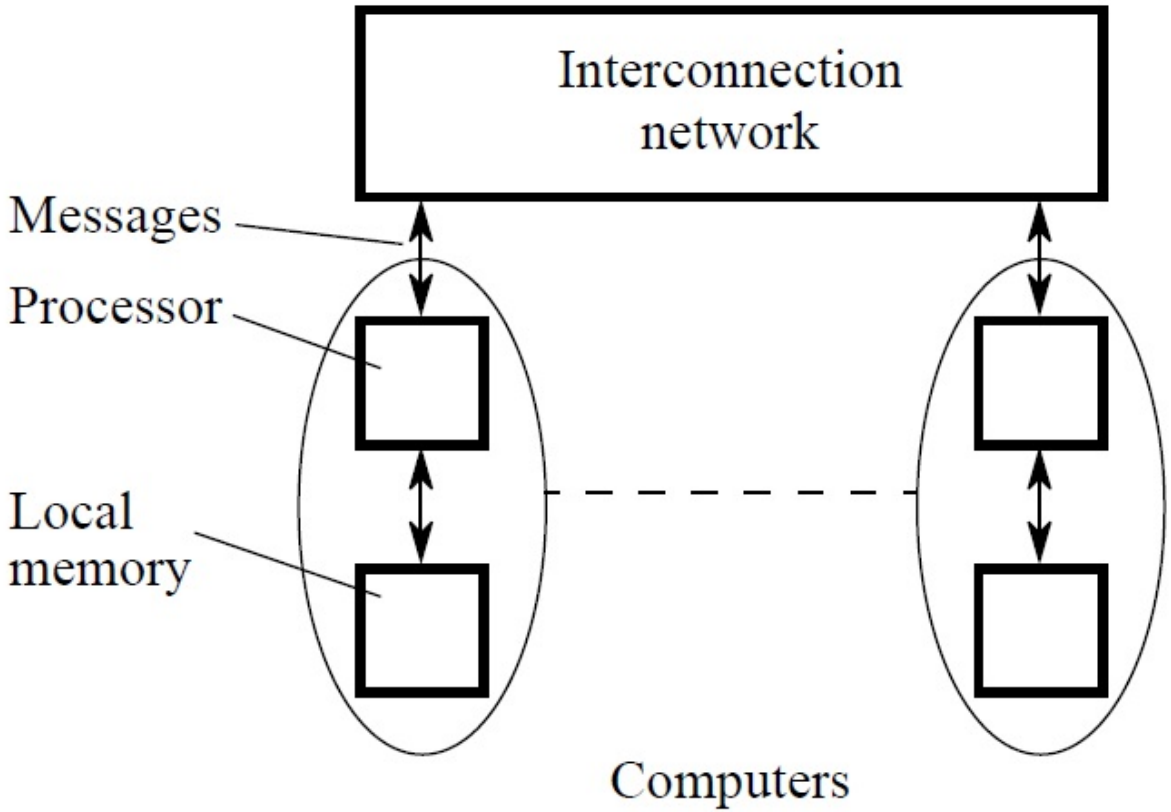
Distributed memory multicomputers

- Multiprocessor systems are suitable for general purpose multi-user applications
- A major shortcoming is lack of scalability
- Difficult to build MPP (massively parallel processing) machine with shared memory models
- Latency for remote memory access is another major limitation
- Packaging and cooling imposes additional constraints
- Alternative - Use multiple computers (nodes) interconneted by a message passing network providing point-to-point connections among nodes
- Each node is an autonomous computer with its own processor, local memory and sometimes peripherals
- All local memories are private and accessible to local processors only.

Distributed memory multicomputers

- Sometimes called NORMA (no remote memory access) machines.
- If a programmer can clearly decompose a problem, all that is required is to have a large number of processors with some basic communication ability.
- The program is fully aware of when it needs to communicate with other processors
- For such applications distributed memory multicomputers are a good choice
- Many programming environments work effectively on these systems
- MPI, PVM, OpenMP are examples of such environments

Distributed memory multicomputers



2

[2] B. Wilkinson, and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice-Hall, 1999.

Interconnects and Related Material

- Efficient interconnect required for fast communication between different components in a multicomputer system
- High performance computing and networking share many technologies and have strong similarities
  Network Peoperties:
- Used to estimate the complexity, communication efficiency and cost of a network
- Nerwork Size: The number of nodes in a network
- Network Diameter: the minimum number of links between two farthest nodes in the network. The number of physical links in a path is a major factor in determining the delay for a messsage.
- Bandwidth: maximum data transfer rate bits/sec or Mbytes/sec

Interconnects and Related Material

- Network Latency is the worst case time delay for a unit message to be transferred through the network
- Communication Latency: is the total time to send a message including software overhead and interface delays
- Start up time: time required to send a zero length message - finding route, packing, unpacking etc
- Bisection Width: Minimum number of links that must be broken to divide the network into two equal parts
- Bisection Bandwidth: maximum number of bits that can be transmitted per unit time from one part of the divided network to the other part
- Hardware Complexity/cost: implementation costs such as those for wires, switches, connectors, arbitration and interface logic
- Scalability:ability to be modularly expandable

Static Connection Networks

- Also called **Direct Interconnects**:
  - Use direct links that are fixed once built
  - Used for fixed connections among subsystems
  - suitable for building computer systems where communication patterns are predictable

    Examples:

    Completely connected networks:
  - Require $n(n-1)/2$ links
  - Practical only for small n

Static Connection Networks

- Linear arrays:
  - Simplest connection topology
  - N nodes connected by N - 1 links
  - $Diameter = N - 1$, $Bisection = 1$
  - Very different from Bus, which is time shared through switching among the many nodes attached to it.
  - Linear array allows concurrent use of different sections (channels) of the structure by different source and destination pairs
  - Structure not symmetric and results in large communication inefficiency when N is large

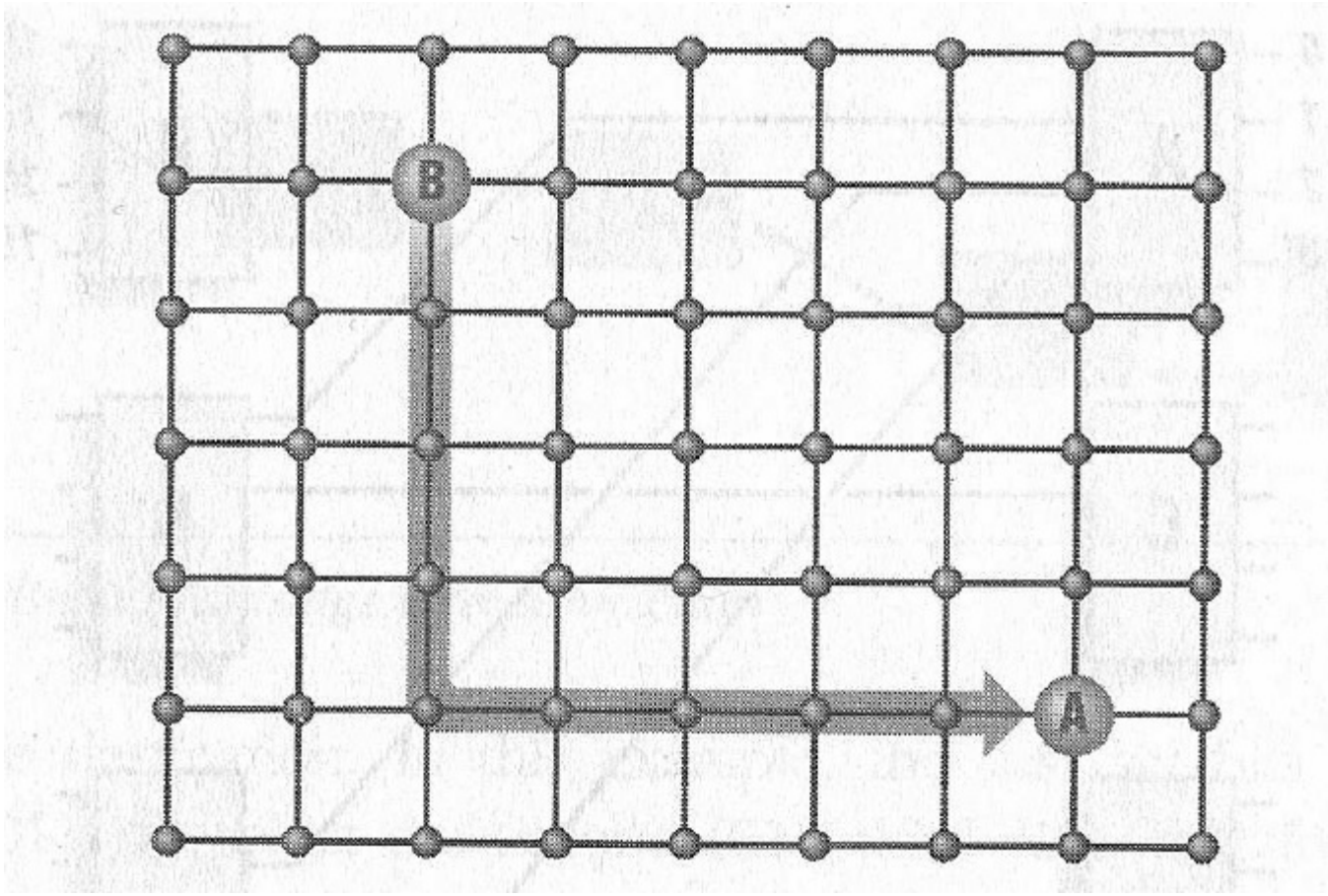
Static Connection Networks

- Ring or Chordal Ring:
  - Similar to linear arry with extra link connecting the last node to the first.
  - Can be unidirectional or bidirectional

Static Connection Networks

- Mesh:
  - Each node is connected to four Cartesian neighbours
  - To move data map a Cartesian route (X, Y) through the mesh
  - The first bits of a message are used at each router to determine whether the message is to go north, south, east or west.
  - For p nodes, assuming $\sqrt{p}$ is an integer, we have $\sqrt{p} \times \sqrt{p}$ mesh
  - The diameter is $2(\sqrt{p} - 1)$

Mesh



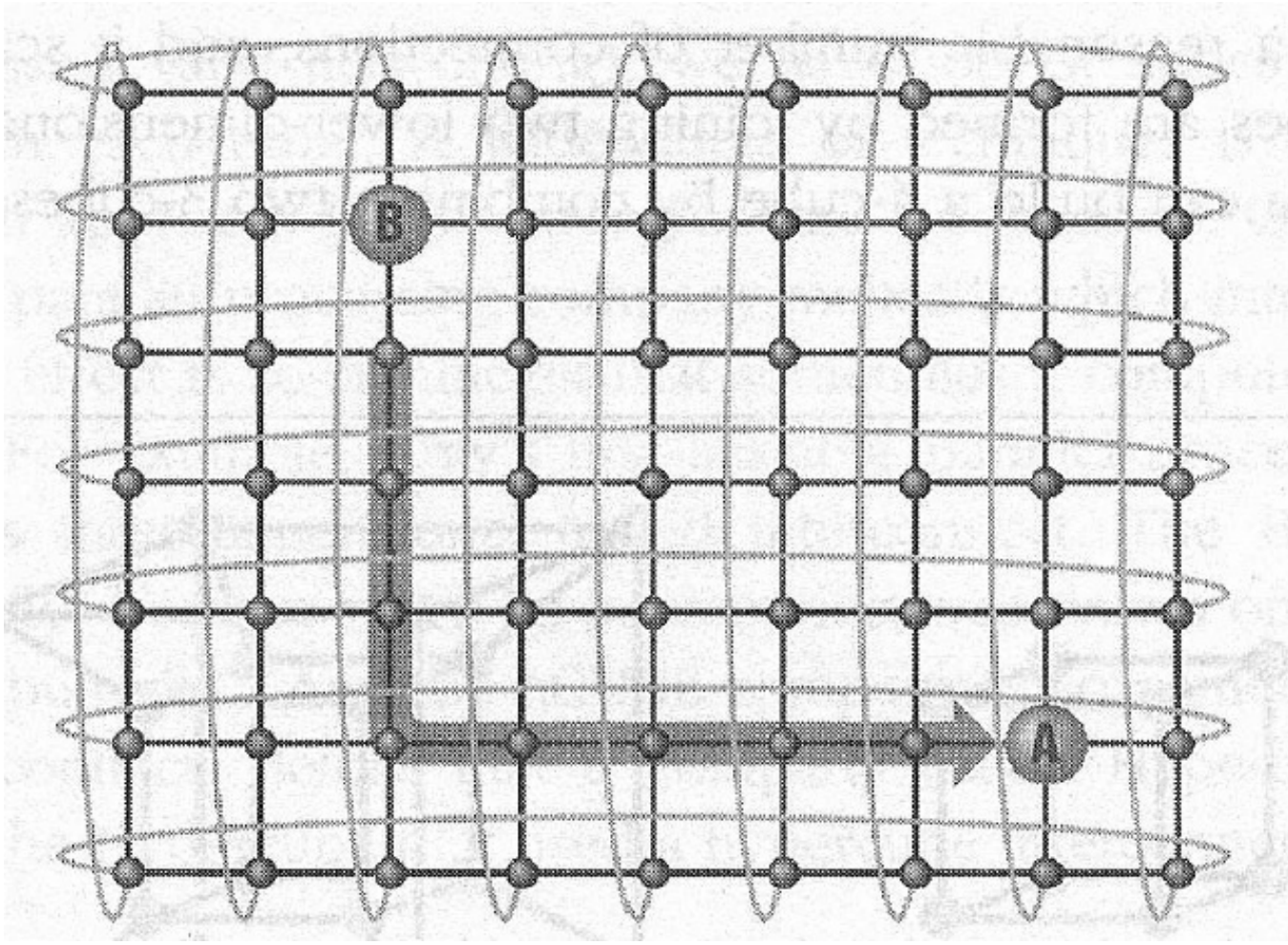
3

[3] B. Wilkinson, and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice-Hall, 1999.

Static Connection Networks

- Torus Networks:
  - Similar to mesh but the unused ends of the mesh are wrapped around to the opposite end
  - Reduces worst case number of hops by 50%
  - Each node in a $\sqrt{p} \times \sqrt{p}$ torus has four links with $2p$ links in total

Two Dimentional Toroid



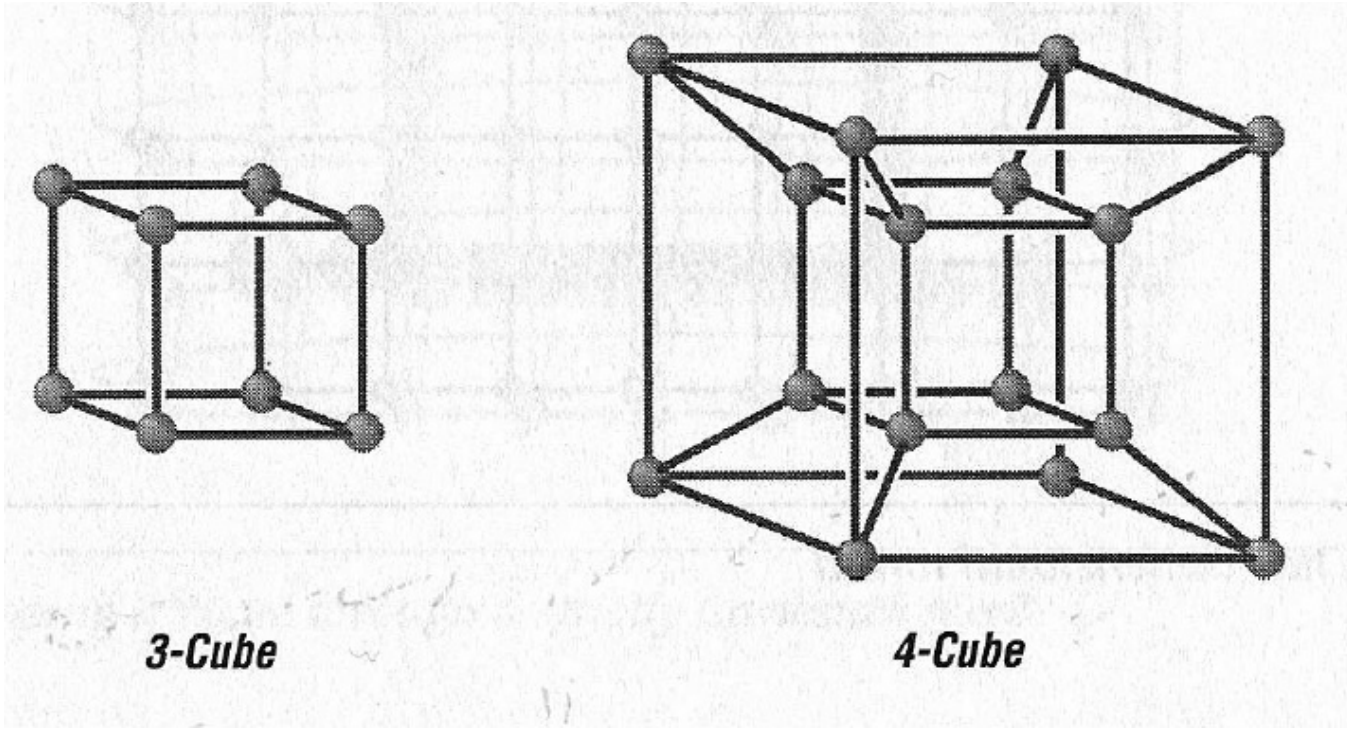
[4]

[4]B. Wilkinson, and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice-Hall, 1999.

Static Connection Networks

- Hypercubes:
  - 8 processors can be connected in a 3-cube
  - The number of processors in an n-cube $N = 2^n$ where n is the dimension of the cube
  - Diameter is $= log_2(N) = n$
  - Bisection width $= N/2$
  - Higher dimension cubes can be built using lower dimension cubes
  - Can build a 4-cube using two 3-cubes
  - In general one can build an n-cube using two $(n-1)$-cubes
  - Scales well - 1024 processors need a 10-cube with longest path of ten hops
  - Same system using a two dimensional mesh will have the longest path of 64 hops!

Hypercubes



5

[5] B. Wilkinson, and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice-Hall, 1999.

Static Connection Networks

- Hypercube Routing:
  - Has a simple minimum distance deadlock free routing algorithm, called *e-cube routing algorithm*
  - Each node assigned an n-bit binary address, each bit corresponds to a dimension
  - Assume a message is to go from node X to node Y
  - Each bit of Y that is different from X represents the dimension of the hypercube that the message should take
  - Can be found by taking an exclusive OR of the binary address of Y with the binary address of current node
  - At each step use the most significant 1-bit of the result to negate that bit of the current node address and get the next node address

Static Connection Networks

- Example:
  - Want to send a message from node X to node Y. Their addresses $X = 001101$ to $Y = 101010$
  - Exclusive OR addresses of X and Y to get 100111
  - The most significant 1-bit of the result is the 6th bit
  - Negate the fifth bit of the address of the current node $(001101) \Rightarrow (101101)$
  - Exclusive OR (101101) with $Y = 101010$ to get 000111 - most significant 1-bit is the third bit
  - Negate third bit of the address of the current node $(101101) \Rightarrow (101001)$

Static Connection Networks

- Example Continued ...:
  - Exclusive OR (101001) with $Y = 101010$ to get 000011 - most significant 1-bit is the second bit
  - Negate the second bit of the current address to get 101011
  - Next exclusive OR gives 0000001 - most significant 1-bit is the first bit
  - Negate the first bit of the current address to get 101010 - destination

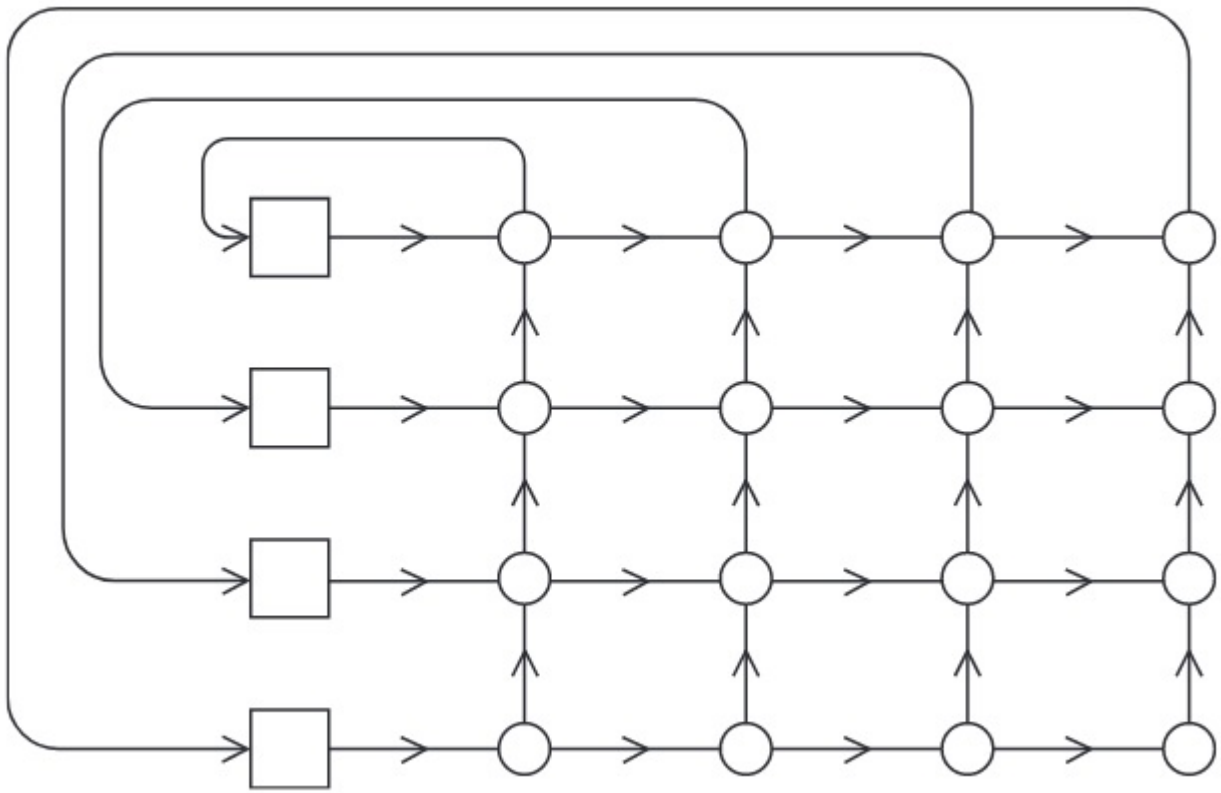
Static Connection Networks

- Binary Trees:
  - Height of a tree is the number of links from root to the lowest leaves
  - Root is considered as level 0
  - The the number of nodes, above level $k = 2^{k+1} - 1$
  - Diameter $2k$
  - There are $log_2 p$ levels of switches for p processes attached to leaves
  - Bottleneck problem towards the root
  - Fat trees - increase the number of links on each level towards the root.

Dynamic Connection Networks

- Also called **Indirect Interconnections**
- Can implement all communication patterns based on program demands
- Switches and arbiters are used for dynamic connectivity
- Distributed Memory Crossbars
  - Have unidirectional links
  - All processors can simultaneously communicate with another process as long as two of them don't try to communicate with the same process
- Omega Network
  - The switches are 2X2 cross bars
  - Uses a total of $2p\ log_2(p)$ of switches while crossbar uses p^2

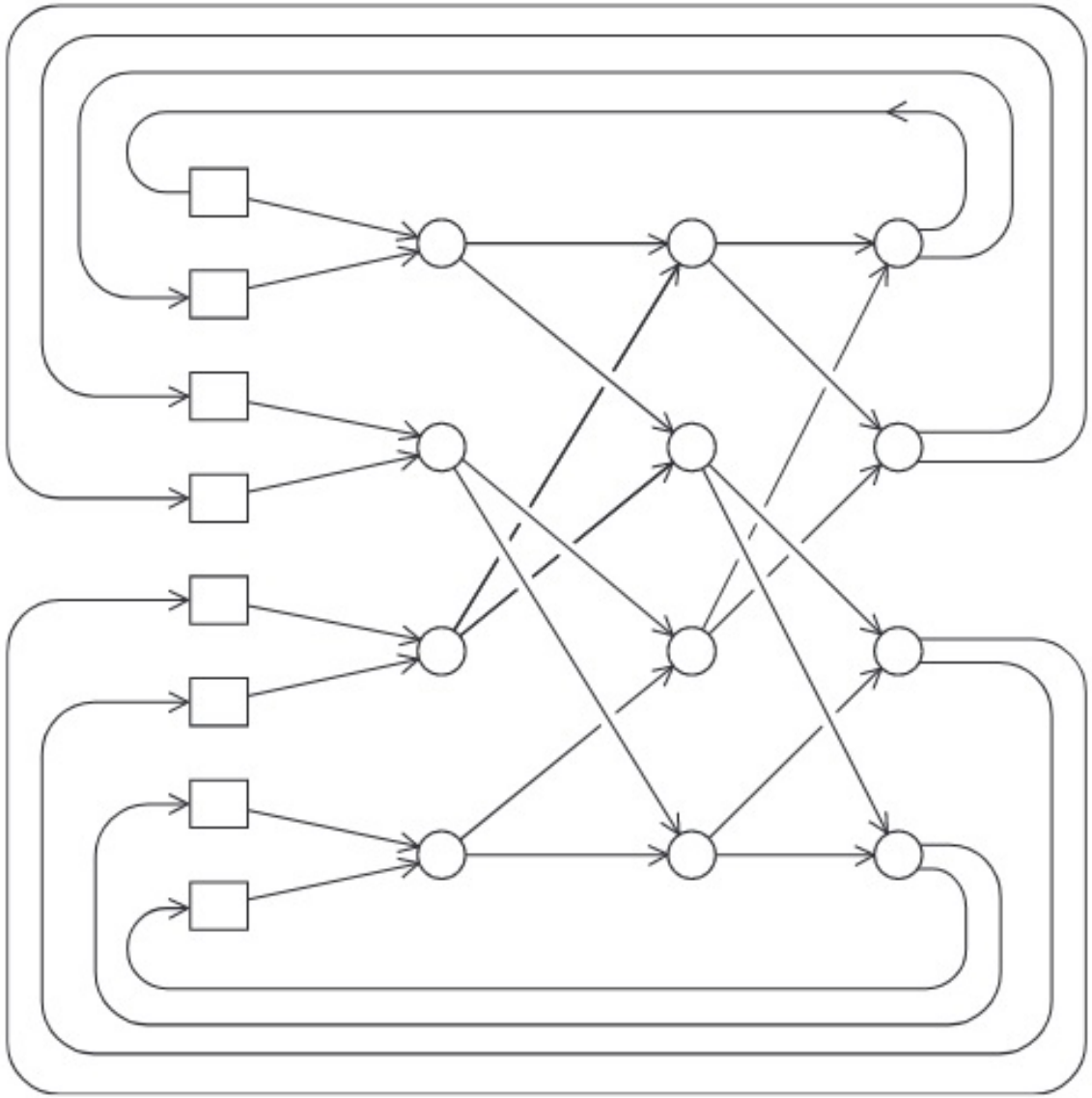
Distributed Memory Crossbar



Multistage interconnect network (MIN)

- Omega networks an example of this catagory
- Use a number of *axb* crossbars in each stage
- An *aXb* crossbar has *a* inputs and *b* outputs
- Theoretically a and b do not have to be equal, but in practice a and b are chosen as integer powers of 2 ($a = b = 2^2$)
- MIN can scale beyond a bus or a crossbar
- May have contention
- For example node 0 wants to communicate with node 6 and at the same time node 1 wants to communicates with node 7
- Number of stages = $log_2(p)$, p/2 crossbars in each stage
- Total number of crossbars = $1/2p * log_2(p)$
- Number of switches = $2p * log_2(p)$ (4 switches per crossbar)

Omega Network

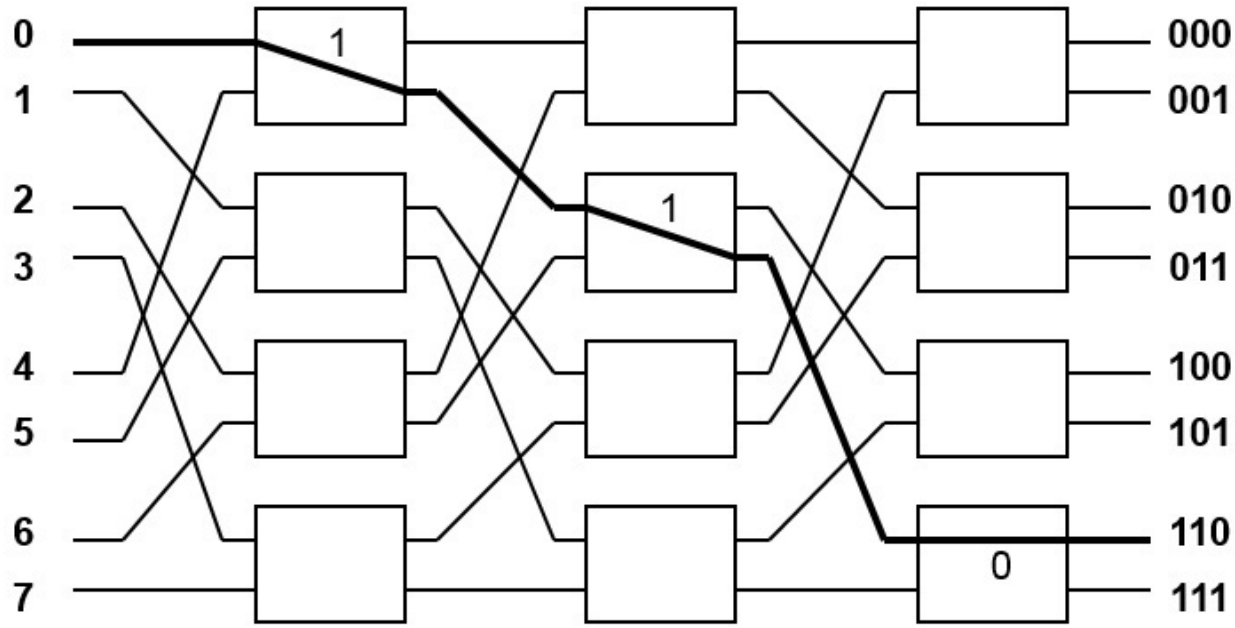


Multistage interconnect network (MIN)

Self Routing:

- The source node generates a tag, representing binary equivalent of the destination address
- At each stage the corresponding tag bit is checked
  - Can be considered in two ways:
    1) Tag represents the address by reversing the bits ($b_2 b_1 b_0 \rightarrow b_0 b_1 b_2$)
    2) The MSB is checked at first stage followed by next bits at subsequent stages.
  - If the bit is zero, the input is connected to the upper output
  - If the bit is one, the input is connected to the lower output.
  - If both inputs have the same value (0 or 1), it is a switch conflict. One of them is connected.
  - The other input is either rejected or buffered at the switch depending upon implementation

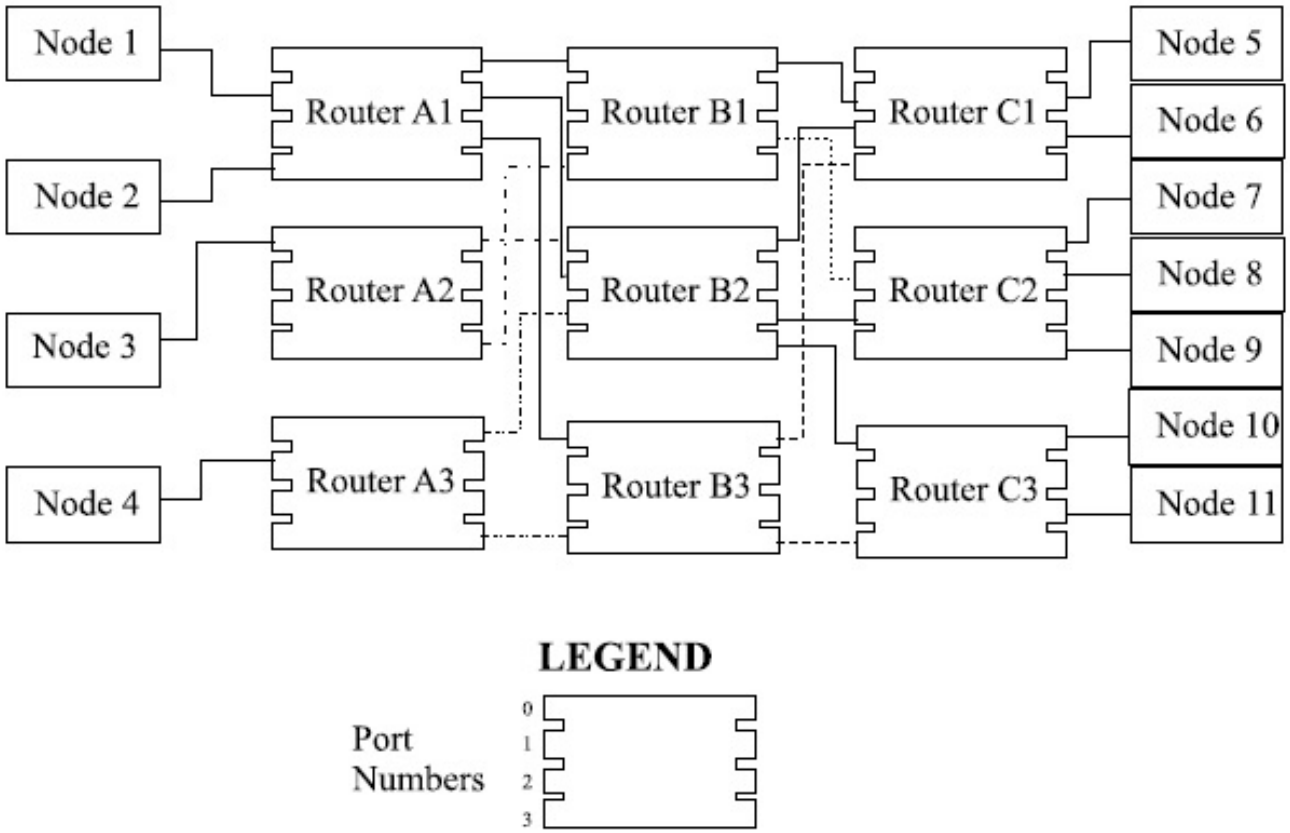
MIN



Pipelined MIN

- Sometimes called delta networks
- For large number of nodes, path may consist of many steps with a router in each step to receive and forward messages.
- Output at each stage connected to input of next.
- Each router can have several inputs and outputs
- Message is pipelined at each stage by dynamically establishing a temporary link through the network
- Depending upon the number of stages of routers and the number of inputs/outputs at each router, a suitable tag is sent prepended to the message
- Diagram shows 3 stages of 4x4 routers. The tag consists of 6 bits (2 consumed at each stage)
- Example: Node 1 sends a message with leading bits 101101 to connect to node 11

MIN Pipelined



Performance

- Speedup Factor
  - Measure of performance improvement between a single processor and parallel computer is the *speedup factor*, $S(n)$ (n the number of processors).

$$
\begin{aligned}
S(n) &= \frac{\text{Execution time using single CPU}}{\text{Execution time using } n \text{ processors}} \\
&= \frac{t_s}{t_p}
\end{aligned}
$$

Performance

- Linear Speedup
  - The maximum speedup factor due to parallelization is n with n processors. This is called *linear speedup.* For example:

$$S(n) = \frac{t_s}{t_s/n} = n$$

  - *Superlinear speedup* is when $S(n) > n$.
  - Unusual, and usually due to a suboptimal sequential algorithm or special features of the architecture which favor parallel programs.
  - Normally, can't fully parallelize a program. Some part must be done serially.
  - There could be periods when only one processor doing work, and others idle.

Performance

- Overhead
  - *Overhead* consists of the time delays that are added due to the parallelization.
  - Factors that cause overhead and reduce speedup are:
    - Periods when all processors are not doing useful work.
    - Additional computations not in serial version.
    - Communication time for sending messages.

Performance

- Speedup Limitations **Amdahl's Law**
  - Normally some fraction of a program must be performed serially
  - Let the serial fraction be $= f$.
  - The fraction that can be parallized $= (1 - f)$
  - Assume no overhead results when program divided into concurrent parts,
  - Then computation time with n processors is:

$$t_p = ft_s + (1 - f)t_s/n$$

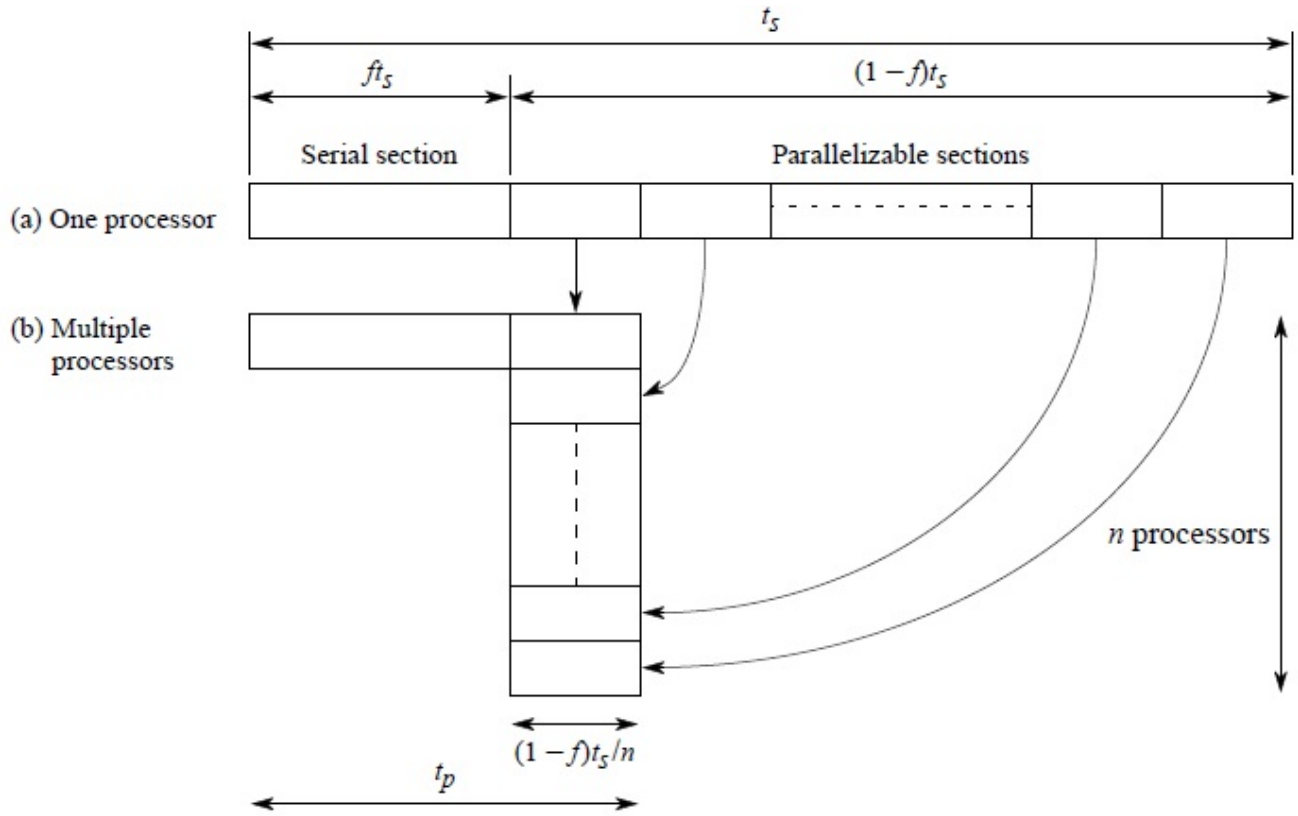
  - Amdahl's Law:

$$S(n) = \frac{t_s}{ft_s + (1 - f)t_s/n} = \frac{n}{1 + (n - 1)f}$$

  - Limit of speedup with infinite number of processors:

$$S(n)_{n \to \infty} = \frac{1}{f}$$

Amdahl's Law



Parallelizing sequential problem — Amdahl's law.

6

[6] B. Wilkinson, and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice-Hall, 1999.

Amdahl's Law

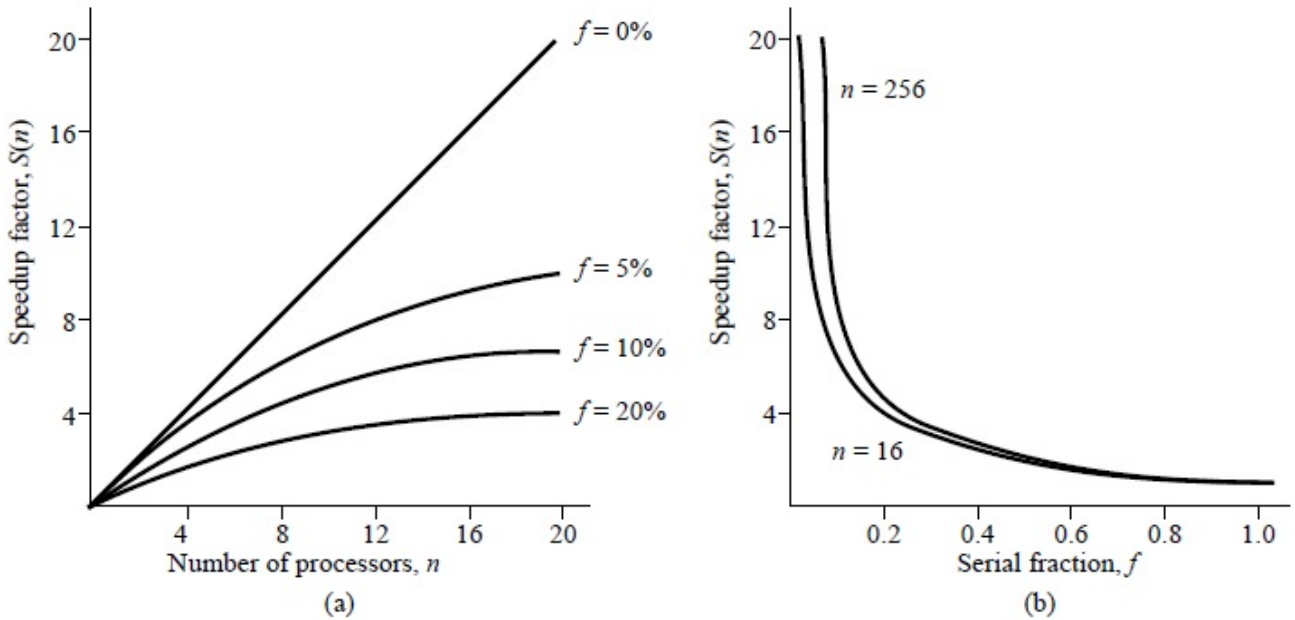


Figure 1.30 (a) Speedup against number of processors. (b) Speedup against serial fraction, f.

7

[7] B. Wilkinson, and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice-Hall, 1999.

Performance

- Efficiency
  - The system *efficiency*, E, is defined as:

$$E = \frac{\text{Execution time with single CPU}}{\text{Execution time with multiprocessor} \times \text{number of CPUs}}$$

$$= \frac{t_s}{t_p \times n}$$

  - Efficiency as a percentage is given by:

$$E = \frac{S(n)}{n} \times 100\ \%$$

  - The *processor-time* product, or *cost* of a computation is:

$$Cost = t_p \times n = \frac{t_s n}{S(n)} = \frac{t_s}{E}$$

Performance

- Scalability
  - Has different meanings.

    Architecture or Hardware Scalability: A hardware design that permits system to be enlarged which results in increased performance.
    Usually, adding more processors means network must be increased. Means greater delay and contention thus lower efficiency.

    Algorithmic Scalability: Means parallel algorithm can handle increase in data with low and bounded increase in computational steps.

Performance

- Scalability Continued ..
  - One way to define *problem size* is the number of data elements to be processed.
  - Problem: doubling data size doesn't necessarily mean doubling number of computation steps.
  - Alternative: equate *problem size* with number of steps in best sequential algorithm.
  - Still, increasing number of data elements will mean increasing "problem size."

Performance

- Gustafson's Law
  - Gustafson argued that Amdahl's law is not as limiting as it seems.
  - Observed that a large multiprocessor means you can handle bigger problems.
  - In practice, problem size is related to number of processors.
  - Assume parallel processing time fixed, not problem size.
  - Gustafson claimed that serial section doesn't increase as problem size increases.
  - Let s be the serial computation time. Let p be the time required to execute the parallel portion of the program on a single processor.
  - Assume $s + p$ is fixed. For convenience, we choose $s + p = 1$, so they represent fractions.

Performance

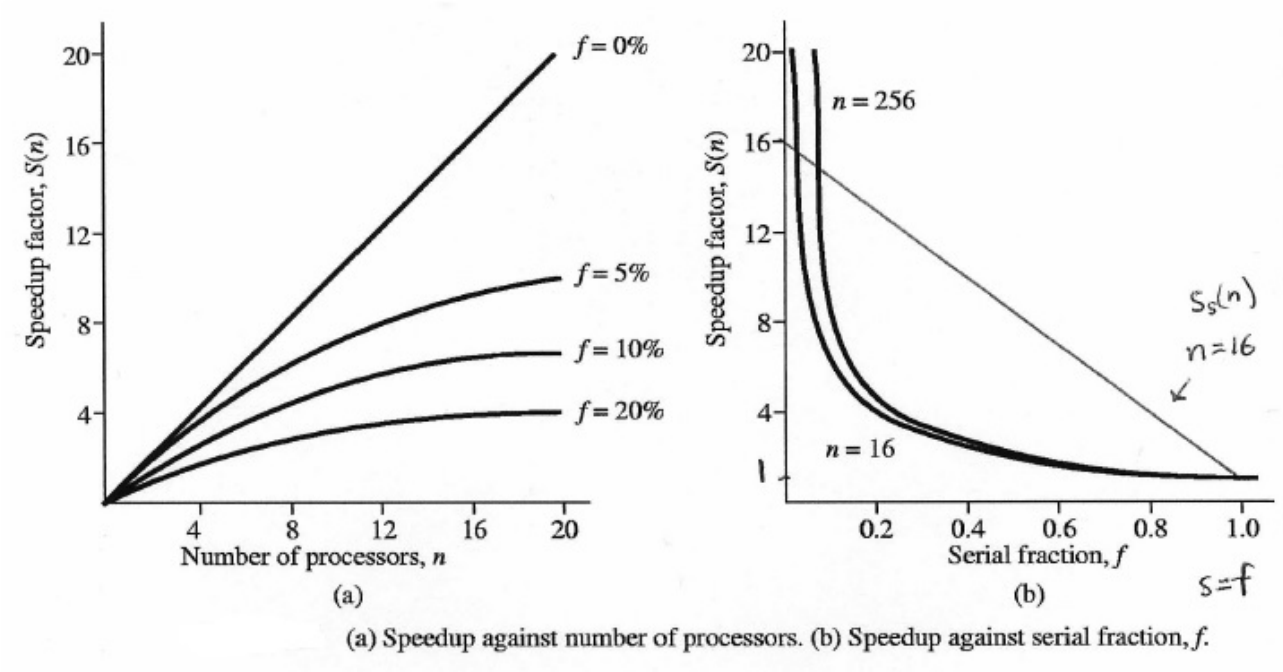
- Gustafson's Law
  - We can then represent Amdahl's law as:

$$S(n) = \frac{s+p}{s+p/n} = \frac{1}{s+(1-s)/n}$$

  - For Gustafson's *scaled speedup factor*, $S_s(n)$, he assumes the parallel execution is constant, and the serial time (t_s) changes.
  - Again, we have $s + p = 1$. Execution time on single processor is now $s + pn$. This gives us:

$$S_s(n) = \frac{s+np}{s+p} = s + np = n + (1-n)s$$

Amdahl's Law



(a) Speedup against number of processors. (b) Speedup against serial fraction, f.

8

[8] B. Wilkinson, and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice-Hall, 1999.

Performance

- Granularity
  - For any parallel computer, must break computation into multiple *processes* (tasks) that can run simultaneously.
  - *Granularity* describes the size of a process.
    - Coarse Granularity
    - Fine Granularity
    - Medium Granularity
  - Communication overhead for message passing
  - Want to reduce amount of intercomputer communication
  - There's a point when the communication time dominates execution time

$$\text{Computation/communication ratio} = \frac{\text{Computation time}}{\text{Communication time}} = \frac{t_{comp}}{t_{comm}}$$

  - Want to maximize ratio, but still maintain sufficient parallelism.

Performance

- Example
  - Consider two computers connected by a network.
  - Each CPU has clock period T, and the communication latency for the network is L.
  - Each process (1 per CPU) must process 10 commands.
  - Each command takes 30 clock cycles to execute. Process 1 then sends a message to process 2, who then replies.
  - For our system, T = 12.5ns, and L = 500us.
    $t_{comp} = 10 \cdot 30 \cdot T = 10 \cdot 30 \cdot 12.5 \times 10^{-9} s = 3.75 \times 10^{-6} s$

    $t_{comm} = 2 \cdot L = 2 \cdot 500 \times 10^{-6} s = 0.01 s$
    $ratio = \frac{t_{comp}}{t_{comm}} = \frac{3.75 \times 10^{-6} s}{0.01 s} = 0.00375$
    NOTE: total execution time of task is: $t_{comp} + t_{comm}$

Performance

- Comments
  - Granularity related to number of processors.
  - For domain decomposition (partitioning the data), the size of the per CPU data can be increased to improve ratio.
  - For a fixed data size, could reduce number of CPUs used.
  - Want to design a program where it is easy to vary granularity.

Title

-
-
- Template!