Parallel Computing

Kamlesh Tiwari

1 Introduction Parallel Computing

Evaluation of the computer architecture have undergone following stages.

Sequential machines

Pipelined machines

Vector machines

Parallel machines

1.1 Pipelining

Instruction execution have following *stages.*

$\Rightarrow$ [Instruction fetch]
  $\hookrightarrow$ [instruction decode]
    $\hookrightarrow$ [argument decode]
      $\hookrightarrow$ [argument fetch]
        $\hookrightarrow$ [execution]
          $\hookrightarrow$ [result storage]

This can be made faster either by pushing technology, or by efficient use of hardware.

A pipeline is the continuous and somewhat overlapped movement of instruction to the processor or in the arithmetic steps taken by the processor to perform an instruction. It is an efficient hardware utilization to increase *throughput.*

1.1.1 Example

Consider a 5 stage pipeline of a RISC processor.

IF instruction fetch IR[1] = mem[PC[2]];
NPC[3] = PC + 4

ID instruction decode/Operand fetch A = Regs[$\text{IR}_{6..10}$];
B = Regs[$\text{IR}_{11..15}$];
Immediate = [$\text{IR}_{16..31}$];

EX execution stage

- Memory reference
  ALUoutput = A + Imm
- Register-Register ALU operation
  ALUoutput = A op B
- Register-Immediate ALU operation
  ALUoutput = A op Imm
- Branch
  ALUoutput = NPC + Imm
  Cond = (A op 0)

MEM memory access

- Memory reference
  MDR = Mem[ALUoutput] or,
  Mem[ALUoutput] = B
- Branch
  if (Cond) PC = ALUoutput
  else PC = NPC

WB write-back stage

- Reg-Reg ALU instruction
  Regs[$\text{IR}_{16..20}$] = ALUoutput
- Reg-Imm ALU operation
  Regs[$\text{IR}_{11..15}$] = ALUoutput
- Load instruction
  Regs[$\text{IR}_{11..15}$] = MDR

Instruction flow:

	Clock number								
Inst No.	1	2	3	4	5	6	7	8	9
Inst 1.	IF	ID	EX	MEM	WB				
Inst 2.		IF	ID	EX	MEM	WB			
Inst 3.			IF	ID	EX	MEM	WB		
Inst 4.				IF	ID	EX	MEM	WB	
Inst 5.					IF	ID	EX	MEM	WB

Question: *Assume a non-pipelined version of the processor with 10ns clock. Four cycles for non-memory (ALU and branch ops) Five cycles for memory ops. 40% instructions are memory ops. pipelining the machine needs 1ns extra on clock. What is speedup?*

Answer: Average time for the execution of an instruction on non pipelined processor. $(0.4 \times 5 + 0.6 \times 4) \times 10ns = 44ns$

Pipelined version will execute every instruction in one clock cycle. Since in pipelined version there is an over head of 1ns therefore time to execute one instruction will be of $10ns + 1ns = 11ns$.

Hence the speedup is $44ns/11ns = 4$

1.1.2 Pipeline Hazards

There can be basically three type of hazards.

[1]Instruction register
[2]program counter
[3]Next program counter

1. **Structural hazards** arises due to resource conflict. When two different instructions in the pipeline want to use same hardware this kind of hazards arises, the only solution is to introduce *bubble/stall.*

   Question: *Ideal CPI*[4] *1.0 for machine without structural hazard. 40% instructions causing structural hazards, clock rate of machine with structural hazards 1.05 time faster. compare the performance.*

   Answer: Out of 100 instructions 60 will be normal taking 1 CPI, rest 40 instructions will take one extra CPI due to structural hazards. Therefore for 100 instructions total CPI consumed is $60 \times 1 + 40 \times (1+1) = 140$. On an average one instruction will take 1.4 CPI.

   Since the clock pulse is 1.05 times faster, therefore the time taken by a single instruction is 1.4/1.05 = 1.33 CPI. *Conclusion: if machine takes t sec to execute an instruction in pipeline without structural hazards, then it will take* $1.33t$ *sec with structural hazards.*

2. **Data hazards** arises due to instruction dependence. For example in following code
   ADD R1, R2, R3 ;R1←R4+R3
   SUB R4, R5, R1 ;R4←R5-R1
   until the value of R1 becomes available the execution of second instruction can not be carried out.

   To minimize data hazards we can do *internal forwarding*, that is ALU output is fed back to the ALU input (if the hardware detects that the previous ALU op has modified the register corresponding to the source of current ALU op, ALU output is forwarded).

   Data hazards are classified as below. (in instruction i and j instruction i occurs before j)

   - **RAW** (Read After Write) j tries to read a source before i writes it.
   - **WAW** (Write After Write) j writes an operand before i writes it.
   - **WAR** (Write After Read) j writes an operand before i reads it.
   - **RAR** (Read After Read) NOT a hazard!!!

   Not all but some data hazards can be avoided. The general practice is to handle data hazards is to introduce sufficient *bubble* by *pipeline interlock* (This is done by a hardware).

   Some time **compiler** handle the data hazards efficiently; for example see following codes (which uses rescheduling of instructions).

With data hazard	Without data hazard
ADD R1, R2, R3	ADD R1, R2, R3
SUB R4, R5, R1	SUB R6, R2, R3
SUB R6, R2, R3	MUL R7, R5, R3
MUL R7, R5, R3	**SUB R4, R5, R1**

3. **Control hazards** arises due to branches[5].

   We can handle control hazards with compiler's assistance by static branch prediction, or by reducing dependency stalls by instruction rescheduling.

[4]cycle per instruction
[5]Every fifth instruction is typically a branch in compiled program

1.2 Advanced Techniques

- **Basic pipeline scheduling**
  To reduce stalls between instructions, instructions should be fairly independent of each other.

  Consider following loop
  for $(i = 1, i <= 1000; i++) X[i] = X[i] + S;$

  Which is compiles to

```
X:  LD    F0, 0(R1)    ;F0 = M(0+R1)
    ADD   F4, F0, F2   ;F4 = F0 + F2
    SD    0(R1), F4    ;M(0+R1) = F4
    SUB   R1, R1, 8    ;R1 = R1 + 8
    BNEX  R1, X
```

  Due to stalls introduced the execution of the program becomes as below

```
X:  LD    F0, 0(R1)    ;F0 = M(0+R1)
    Stall
    ADD   F4, F0, F2   ;F4 = F0 + F2
    Stall
    Stall
    SD    0(R1), F4    ;M(0+R1) = F4
    SUB   R1, R1, 8    ;R1 = R1 + 8
    BNEX  R1, X
    Stall
```

  Which takes 9 cycles per iteration.

  Compiler can *reschedule loop* as below.

```
X:  LD    F0, 0(R1)    ;F0 = M(0+R1)
    Stall
    ADD   F4, F0, F2   ;F4 = F0 + F2
    SUB   R1, R1, 8    ;R1 = R1 + 8
    BNEX  R1, X
    SD    0(R1), F4    ;M(0+R1) = F4
```

  Which will require only 6 cycles per iteration and hence will save 33% of execution time.

- **Loop unrolling**
  Loop unrolling reduces number of branches therefore control stalls. It is a better pipeline scheduling which reduces the overhead of loop control code as well.

  Continuing with the previous example of for loop, its unrolling is shown below.

```
X:  LD    F0, 0(R1)
    ADD   F4, F0, F2
    SD    0(R1), F4
    LD    F6, -8(R1)
    ADD   F8, F6, F2
    SD    -8(R1), F4
    LD    F10, -16(R1)
    ADD   F12, F10, F2
    SD    -16(R1), F4
    LD    F14, -24(R1)
    ADD   F16, F14, F2
    SD    -24(R1), F4
    SUB   R1, R1, 32
    BNEX  R1, X
```

  It will require 27 cycles per four iteration (due to hazards).

But if compiler reschedules the instructions as below

```
X:  LD     F0, 0(R1)
    LD     F6, -8(R1)
    LD     F10, -16(R1)
    LD     F14, -24(R1)
    ADD    F4, F0, F2
    ADD    F8, F6, F2
    ADD    F12, F10, F2
    ADD    F16, F14, F2
    SD     0(R1), F4
    SD     -8(R1), F4
    SD     -16(R1), F4
    SUB    R1, R1, 32
    BNEX   R1, X
    SD     -24(R1), F4
```

It will require only 14 cycles per four iteration, and thereby saving 60% as compared to original loop.

- Dynamic scheduling
- Register renaming
- Branch prediction
- Multiple issue of instructions
- Dependence analysis at compilation
- Software pipeline
- Memory pipeline

1.3 Data Dependence

There can be following types of data dependence.

1. **True dependence (or True Data Dependence)**
   See following example

   ```
   SUB R1, R1, 8    ;R1 = R1 - 8
   BNEZ R1, X
   ```

2. **Name dependencies**
   Arises due to reuse of register/memory.

3. **Anti-dependence**
   Instruction i executes first and reads a register which instruction j writes later. WAR hazards.

4. **Output dependence**
   Instruction i executes first and writes a register which instruction j writes later. There must be serialization. WAR hazards. WAW hazards.

5. **Control dependence**
   Consider following instruction
   if P1 {S1;}
   instruction S1 is control dependent on P1.

1.4 Loop level parallelism

Consider following instructions

1. $for(i = 0; i <= 1000; i++) A[i] = A[i] + s;$
   There is no dependence between two iterations.

2. Consider following instructions

   ```
   for(i = 0; i <= 1000; i++)
   {
       A[i+1] = A[i] + C[i];          //S1
       B[i+1] = B[i] + A[i+1];    //S2
   }
   ```

   S1 depends on S1 of previous iterations. S2 depends on S2 on previous iterations and S1 of this iterations.

   S1 can't be parallised.

3. Consider following instructions

   ```
   for(i = 0; i <= 1000; i++)
   {
       A[i] = A[i] + B[i];          //S1
       B[i+1] = C[i] + D[i];    //S2
   }
   ```

   S1 depends on S1 of previous iterations. Can it be parallised ? ....

4. Consider following instructions

   ```
   A[1] = A[1] + B[1];
   for(i = 0; i <= 1000; i++)
   {
       B[i+1] = C[i] + D[i];              //S1
       A[i+1] = B[i+1] + A[i+1];    //S2
   }
   B[101] = C[100] + D[100];
   ```

   ?.................................

1.5 Dynamic scheduling

Pipeline forces to in-order instruction execution, which sometimes introduces *stall* due to dependency between two closely spaced instructions. This in-order instruction execution is always not necessary. We can overcome data hazards by dynamically scheduling instructions.

Consider following example.

```
DIV F0, F2, F4       ; F0 = F2/F4
ADD F10, F0, F8      ; F10 = F0 + F8
SUB F12, F8, F12
```

Due to dependence between DIV and ADD instruction we can not execute SUB instruction. ADD instruction have to wail until DIV is complete, as a consequence SUB also have to wait. This *limitation* can be removed if in-order execution can be relaxed as below.

```
DIV F0, F2, F4
SUB F12, F8, F12
ADD F10, F0, F8
```

- Dynamic scheduling allow out-of-order execution as long as there is no dependence.

- *Issue* instructions in-order
- Instruction begins execution as soon as their data operands are available.
- Split ID pipeline stage into.
  - **Issue -** Decode instruction and check for structural hazards.
  - **Read operands -** wait until no data hazards, then read operands.

1.5.1 Dynamic scheduling with Score-boarding

Score-boarding allows the out-of-order execution when there are enough resources and no data dependence. Every instruction goes through scoreboard. Scoreboard keeps track of data dependence and decides when an instruction can read operands and begin execution.

In case, instruction can't begin execution immediately, it monitors the changes in hardware and decides when an instruction can begin execution. It also decides when an instruction can write its results.

Steps of Execution:

- **Issue:**
  Functional unit needed by instruction is free and no other active instruction has same destination register. Avoids WAW hazard.
- **Read Operands:**
  A source operand is available if no earlier issued active instruction is going to write it or if the register containing the operand is being written by a currently active functional unit. RAW hazards are resolved and out-of-order execution may take place.
- **Execution:**
  The functional unit begins execution and notifies scoreboard on completion.
- **Write result:**
  Scoreboard permits the writing only after it ensures that no WAR hazard will take place. A completing instruction cant be permitted to write the results if
  - There is an instruction that has not read its operands that precedes the completing instruction.
  - One of the operands is the same register as the result of the completing instruction.

Scoreboard doesnt take advantage of internal forwarding. It has also to take care of number of buses available for data transfer.

Structure of scoreboard:

1. Instruction status:

   Indicates which one of the four steps the instruction is in.
2. Functional unit status:
   - Busy
   - Operation to be performed
   - F_i : Destination register.
   - F_j, F_k : Source registers.
   - Q_j, Q_k : Functional unit producing source registers F_j, F_k.
   - R_j, R_k : Flags indicating when F_j, F_k are ready. Set to No after operands are read.
3. Register result status:

   Indicates which functional unit will write each register if an active instruction has the register as its destination.

Example-01 of scoreboard:

	Instruction status			
Instruction	Issue	Read operand	Execution complete	Write result
LD F6,34(R2)	✓	✓	✓	✓
LD F2,45(R3)	✓	✓	✓	
MULTD F0,F2,F4	✓			
SUBD F8,F6,F2	✓			
DIVD F10,F0,F6	✓			
ADDD F6,F8,F2				

	Functional unit status								
Name	Busy	OP	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	F3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status							
	F0	F2	F4	F6	F8	F10	F12 ...	F30
FU	Mult1	Integer			Sub	Divide		

Example-02 of scoreboard:

	Instruction status			
Instruction	Issue	Read operand	Execution complete	Write result
LD F6,34(R2)	✓	✓	✓	✓
LD F2,45(R3)	✓	✓	✓	✓
MULTD F0,F2,F4	✓	✓	✓	
SUBD F8,F6,F2	✓	✓	✓	✓
DIVD F10,F0,F6	✓			
ADDD F6,F8,F2	✓	✓	✓	

	Functional unit status								
Name	Busy	OP	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1			Add		Divide			

Example-03 of scoreboard:

	Instruction status			
Instruction	Issue	Read operand	Execution complete	Write result
LD F6,34(R2)	✓	✓	✓	✓
LD F2,45(R3)	✓	✓	✓	✓
MULTD F0,F2,F4	✓	✓	✓	✓
SUBD F8,F6,F2	✓	✓	✓	✓
DIVD F10,F0,F6	✓	✓	✓	
ADDD F6,F8,F2	✓	✓	✓	✓

	Functional unit status								
Name	Busy	OP	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Divide	Yes	Div	F10	F0	F6			No	No

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU						Divide			

Limitations of scoreboard:

1. Amount of parallelism available among instruction: To determine if independent instructions can be found to execute.
2. Number of scoreboard entries: This limits the window size of the instructions to be looked at. Window generally cant cross branch instructions or loop boundaries.
3. Number and types of functional units.
4. Presence of anti-dependence and output dependence which may lead to WAR or WAW stalls.

1.5.2 Register renaming

Reduces the name dependencies. See following example.

```
ADD    R4, R1, R2
BENZ   R4, X
ADD    R4, R1, R3
BENZ   R4,Y
```

This can be transformed to

ADD R4, R1, R2
BENZ R4, X
ADD **R5**, R1, R3
BENZ **R5**,Y

Register renaming can be static or dynamic.

Dynamic Scheduling: Tomasulo approach

- Uses Reservation Stations in place of Scoreboard.
- Implements register renaming in hardware by buffering operands.
- More reservation stations than registers.
- Distributed approach for hazard detection and execution control.
- Results passed directly from reservation stations to instruction.
- Reservation stations fetches and buffers the operand as soon as it is available.
- Pending instructions designate the reservation stations which will provide their inputs.
- In case of successive writes to a register, only the last one updates the register.
- Register renaming essentially means specifying different reservation stations for pending operands eliminating WAW and WAR.

Steps of execution:

1. **Issue:**
   Fetch instruction from queue. Issue it if empty reservation station is available and send the operands from registers to RS. In case of load or store, issue the instruction if empty buffer is available. Non-availability of RS or buffer leads to stall due to structural hazard.
2. **Execute:**
   If some operand is not available, monitor the common data bus (CDB) while waiting for the register to be computed. Operand is placed in RS as soon as it is available. Execution starts as soon as all operands are available. Checks RAW hazards.
3. **Write result:**
   Result is written on CDB and into registers and RS waiting for it.

Structure of Reservation Station:

- **OP :** Operation to be performed.
- **Qj,Qk :** RS which will provide the source operands.
- **Vj,Vk :** Values of source operands.
- **Busy**
- **Qi :** For each register and store buffer indicating RS that will provide result to be stored.

Example-01: Dynamic scheduling

	Instruction status		
Instruction	Issue	Execution	Write result
LD F6,34(R2)	✓	✓	✓
LD F2,45(R3)	✓	✓	
MULTD F0,F2,F4	✓		
SUBD F8,F6,F2	✓		
DIVD F10,F0,F6	✓		
ADDD F6,F8,F2	✓		

	Reservation status					
Name	Busy	OP	Vj	Vk	Qj	Qk
Add1	Yes	SUB	Mem[34+Regs[R2]]			Load2
Add2	Yes	ADD			Add1	Load2
Add3	No					
Mult1	Yes	MULT		Regs[F4]	Load2	
Mult2	Yes	DIV		Mem[34+Regs[R2]]	Mult1	

	Register status						
Field	F0	F2	F4	F6	F8	F10	F12 ... F30
Q_i	Mult1	Load2		Add2	Add1	Mult2	

Example-02: Dynamic scheduling

	Instruction status		
Instruction	Issue	Execution	Write result
LD F6,34(R2)	✓	✓	✓
LD F2,45(R3)	✓	✓	✓
MULTD F0,F2,F4	✓	✓	
SUBD F8,F6,F2	✓	✓	✓
DIVD F10,F0,F6	✓		
ADDD F6,F8,F2	✓	✓	✓

	Reservation status					
Name	Busy	OP	Vj	Vk	Qj	Qk
Add1	No					
Add2	No					
Add3	No					
Mult1	Yes	MULT	Mem[45+Regs[R3]]	Regs[F4]		
Mult2	Yes	DIV		Mem[34+Regs[R2]]	Mult1	

	Register status						
Field	F0	F2	F4	F6	F8	F10	F12 ... F30
Q_i	Mult1					Mult2	

==========================
insert page 66, 67 here
==========================

1.6 Branch Prediction

Reduces branch penalties by letting the processor to guess the outcome of a branch early enough such that stalls are reduced. For incorrect prediction, no harm is done because pipeline flush starts.

Branch prediction buffer or branch history table is used to store the program behavior. The table is indexed by the address of branch instruction. Information contained in this table helps in predicting whether this branch is taken or not.

The simplest scheme stores a single bit of information giving whether last time this branch was taken or not. Prediction is to take branch if it was taken last time. Update information after prediction turns out to be false. Assume initially table contains false (branch not taken)

How many miss-predictions for the following program?

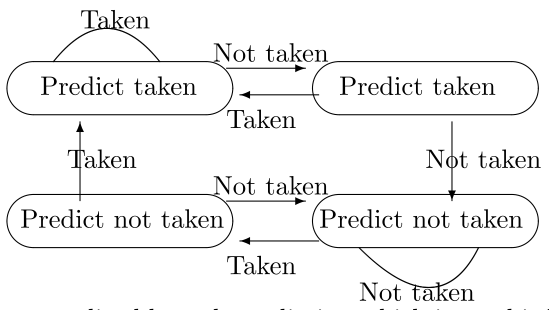
for i=1 to 100 do
X[i]=Y[i]
	Two Times

How many miss-predictions?

for i=1 to 100 do
if odd(i) X[i]=Y[i];
	Always

1.6.1 2-bit Branch Prediction

Miss-prediction should occur twice before table is changed.



The generalized branch prediction which is a n-bit branch prediction based on last m branches is called (n, m) scheme. Generally (2,2) scheme is a sufficient scheme.

Pipeline improvements aim for one clock per instruction (CPI), since because of stalls due to dependencies, effective CPI is more than one. Our aim is to get CPI$<$ 1. This is possible only if multiple instructions are issued in a single clock.

1.6.2 Co-relating branch predictors

Branch predictors that use the behavior of other branches to make a prediction are called correlating predictors or two-level predictors. In the general case an (m, n) predictor uses the behavior of the last m branches to choose from 2^m branch predictors, each of which is an n-bit predictor for a single branch. The number of bits in an (m, n) predictor is given by $2^m \times n \times$ *Number of prediction entries selected by the branch address*

1.6.3 Multiple issue of instruction

Consider the following program

```
X:  LD    F0, 0(R1)    ;F0 = M(0+R1)
    ADD   F4, F0, F2   ;F4 = F0 + F2
    SD    0(R1), F4    ;M(0+R1) = F4
    SUB   R1, R1, 8    ;R1 = R1 - 8
    BNEZ  R1, X
```

Assuming one floating point unit and one integer unit in the processor, we can schedule the unrolled version of this code as below. Unfolding of five loops is performed.

Integer unit			Floating point unit		Cycle
X:	LD	F0, 0(R1)			1
	LD	F6, -8(R1)			2
	LD	F10, -16(R1)	ADD	F4, F0, F2	3
	LD	F14, -24(R1)	ADD	F8, F6, F2	4
	LD	F18, -32(R1)	ADD	F12, F10, F2	5
	SD	0(R1), F4	ADD	F16, F14, F2	6
	SD	-8(R1), F8	ADD	F20, F18, F2	7
	SD	-16(R1), F12			8
	SD	-24(R1), F16			9
	SUB	R1, R1, 40			10
	BNEZ	X, R1			11
	SD	8(R1), F20			12

This will only require 12 cycles per 5 iterations!!!

Multiple issue with Dynamic Scheduling Issues as many instructions as possible. In case of dependence or resource conflict, stop issuing instructions till that conflict is resolved.

Multiple issue with Static Scheduling VLIW instruction and VLIW architectures. Compiler assisted scheduling in a very long instruction word. Difficult to scale, Incompatibility.

Compiler support for ILP exploitation Dependence analysis puts independent instructions next to each other such that hardware does not stall. Several dependence tests are available for example GCD test.

1.6.4 GCD-test

A dependence exists between two iterations of a loop if

1. There are two iteration indices, j and k, both within the limits of the loop. That is $m \leq j \leq n$, $m \leq k \leq n$.

2. The loop stores into an array element indexed by $a \times j + b$ and later fetches the same element when it is indexed by $c \times k + d$, i.e., $A[a \times j + b] = A[c \times k + d]$.

A loop-carried dependence occurs if GCD(c, a) divides $(d - b)$.

Example: In following code snippet.

$$\text{for}(i = 1; i <= 100; i++)$$
$$X[2 * i + 3] = X[2 * i] * 5.0;$$

GCD$(a, c) = 2$,
$d - b = -3$;

Therefore there is NO dependence.

1.6.5 Software Pipelining

Reorganize loops such that each iteration in the software pipelined code is chosen from different iteration of the loop. Like a pipeline, start up and clean up code is needed.

Consider the following code

```
X:  LD     F0, 0(R1)     ;F0 = M(0+R1)
    ADD    F4, F0, F2    ;F4 = F0 + F2
    SD     0(R1), F4     ;M(0+R1) = F4
    SUB    R1, R1, 8     ;R1 = R1 - 8
    BNEZ   R1, X
```

iteration(i)

```
LD     F0, 0(R1)
ADD    F4, F0, F2
SD     0(R1), F4
```

iteration(i + 1)

```
LD     F0, 0(R1)
ADD    F4, F0, F2
SD     0(R1), F4
```

iteration(i + 2)

```
LD     F0, 0(R1)
ADD    F4, F0, F2
SD     0(R1), F4
```

```
X:  SD     0(R1), F4     ; store A[i]
    ADD    F4, F0, F2    ; add to A[i+1]
    LD     F0, -16(R1)   ; load A[i+2]
    SUB    R1, R1, 8
    BNEZ   R1, X
```

Above transformation requires 5 cycles/iteration. Although it requires some start up/clean up code.

Hardware can also have the support for ILP exploitation. For example to replace a statement of kind "if (A==0) S=T;" Hardware can supply following instruction "CMOVZ R1, R2, R3". Control dependencies emplies data dependence.

1.6.6 Memory Pipeline

RISC processors demand high memory bandwidth, superscalar processors demand even higher memory bandwidth. There is always a long job queue for memory. Through processor support we can split phases of memory requests (put requests vs get data), able to schedule another instruction in between, multiple pending reads, out of sequence reads.

Processors supporting split phase memory operations like *Multi-threaded architectures* which run several threads of execution and schedule them automatically in case of dependencies (examples - PowerPC 620, DEC-Alpha and other recent super-scalars).

RAMBUS memory structure utilizing pipelined accesses.

1.6.7 Summary

Processors with low CPI may not always be fast. Increasing issue rate while sacrificing the clock rate may lead to lower performance.

2 Scalable computer architecture

Scaling-up refers to improving the computer resources to accommodate performance and functionality demand. Scaling-down is done to reduce cast. Scaling-up may include

- Functionality and performance.
- Scaling in cast
- Compatibility

Common Architectures for scalable parallel computers are

Shared nothing architecture: Macro architecture.

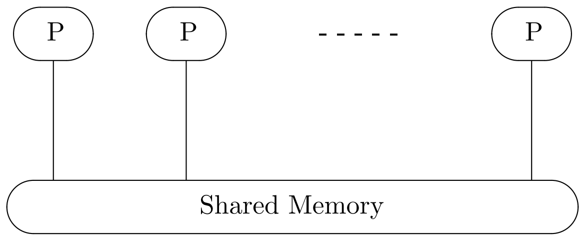
Shared disk architecture:

Shared memory architecture: Micro architecture.

Advantages: size scalability, resource scalability, software scalability, memory scalability, scalability in problem size, generation scalability, space scalabiliy, hetrogenity scalability.

2.1 Parallel ram (PRAM) model

Fully synchronous at instruction level. At each cycle, all memory read from all n instructions must be performed before any processor can perform memory write/branch.



2.2 Sementec Attributes

- **Homogeneity**
  Likeness of processors. Single processors: SISD; multiple processors: MIMD, SIMD, SPMD[6]
- **Synchrony**
  Extent of synchronization among the processors. Fully synchronous at instruction level PRAM[7], SIMD. MIND : asynchronous; synchronous operation must be executed. BSP[8]: Synchronization at every “superstep”.
- **Interaction mechanism**
  The way process interact each other.
  - ⋆ Shared Variable
  - ⋆ Message Passing
  - ⋆ PRAM shared variable/memory
  - ⋆ Multiprocessors: asynchronous MIMD using shared varaible/memory
  - ⋆ Multicomputers: Asynchronous MIND using message passing.
- **Address space**
  Single address space for all memory location vs.
  - ⋆ Multiple address space in Multicomputers
  - ⋆ Uniform memory access (UMA) machine
  - ⋆ Non Uniform Memory Access (NUMA) machine
  - ⋆ Distributed shared memory (DSM)
  - ⋆ Concept of local and global memory.
- **Memory model**
  How to handle shared-memory access conflicts.
  - ⋆ Consistency rules.
  - ⋆ PRAM : Exclusive read exclusive write (EREW) rule.
  - ⋆ Cocurrent read exclusive write (CREW) rule
  - ⋆ Cocurrent read Cocurrent write (CRCW) rule

2.2.1 Read/write conflicts in PRAM

The read/write conflicts in accessing the same shared memory location simultaneously are resolved by one of the following strategies:

1. Exclusive Read Exclusive Write (EREW) - every memory cell can be read or written to by only one processor at a time
2. Concurrent Read Exclusive Write (CREW) - multiple processors can read a memory cell but only one can write at a time
3. Exclusive Read Concurrent Write (ERCW) - never considered
4. Concurrent Read Concurrent Write (CRCW) - multiple processors can read and write.

Here, E and C stand for ’exclusive’ and ’concurrent’ correspondingly. The read causes no discrepancies while the concurrent write is further defined as:

Common all processors write the same value; otherwise is illegal

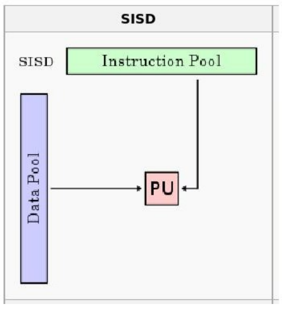
Arbitrary only one arbitrary attempt is successful, others retire

Priority processor rank indicates who gets to write

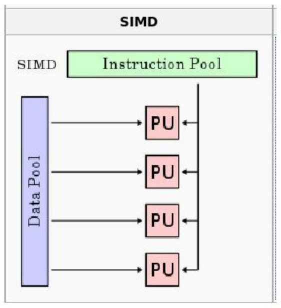
[6]Single Program, Multiple Data
[7]A Parallel Random Access Machine (PRAM) is a shared memory abstract machine which is used by parallel algorithms designers to estimate the algorithm performance.
[8]Bulk Synchronous Parallel Model

2.3 Flynn's classification

Single Instruction, Single Data (SISD):
A sequential computer which exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are the traditional uni-processor machines like a PC.

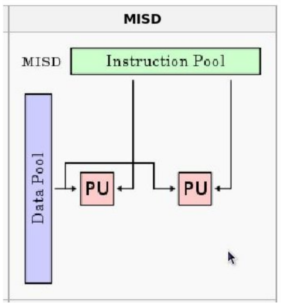


Single Instruction, Multiple Data (SIMD):
A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. For example, an array processor or GPU.

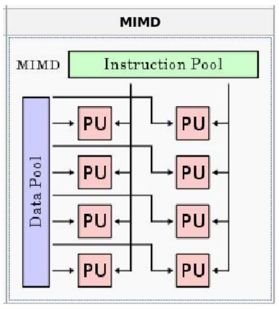


Fully synchronous at instruction level. At each cycle, all memory read from all n instructions must be performed before any processor can perform memory write/branch.

Multiple Instruction, Single Data (MISD):
Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer.



Multiple Instruction, Multiple Data (MIMD):
Multiple autonomous processors simultaneously executing different instructions on different data. Distributed systems are generally recognized to be MIMD architectures; either exploiting a single shared memory space or a distributed memory space.



Asynchronous. Synchronization operations must be executed.

Single program multiple data (SPMD):
Multiple autonomous processors simultaneously executing the same program (but at independent points, rather than in the lockstep that SIMD imposes) on different data. Also referred to as *Single Process, multiple data*. SPMD is the most common style of parallel programming.

Multiple program multiple data (MPMD):
Multiple autonomous processors simultaneously operating at least 2 independent programs.

Bulk Synchronous Parallel Model (BSP):
A BSP computer consists of processors connected by a communication network. Each processor has a fast local memory, and may follow different threads of computation. A BSP computation proceeds in a series of global supersteps. A superstep consists of three ordered stages:

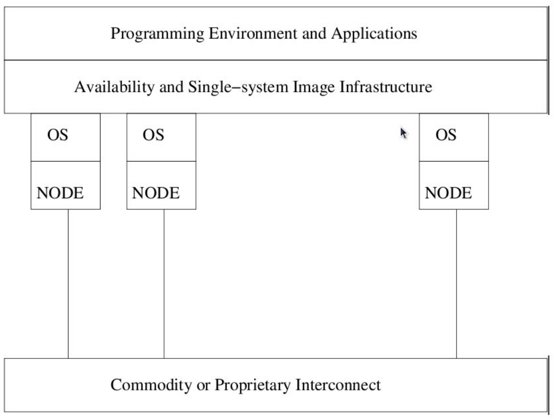
1. Concurrent computation : Several computations take place on every participating processor. Each process only uses values stored in the local memory of the processor. The computations are independent in the sense that they occur asynchronously of all the others.

2. Communication : At this stage, the processes exchange data between themselves.

3. Barrier synchronization : When a process reaches this point (the barrier), it waits until all other processes have finished their communication actions.

BSP is a MIMD system which does synchronization at every superstep. Message passing or shared variable are used.

Cluster: A computer cluster is a group of linked computers, working together closely so that in many respects they form a single computer. The components of a cluster are commonly, but not always, connected to each other through fast local area networks. Clusters are usually deployed to improve performance and/or availability over that of a single computer, while typically being much more cost-effective than single computers of comparable speed or availability.


Programming Environment and Applications
Availability and Single−system Image Infrastructure
OS
OS
OS
NODE
NODE
NODE
Commodity or Proprietary Interconnect

The advantages are Single-system image, inter-node connection, and enhanced availability.

2.3.1 Interaction mechanism

There are two ways

1. Shared variables.
2. Message passing

2.3.2 Granularity

Granularity is the extent to which a system is broken down into small parts, either the system itself or its description or observation. It is the extent to which a larger entity is subdivided. For example, a yard broken into inches has finer granularity than a yard broken into feet.

Coarse-grained systems consist of fewer, larger components than **fine-grained** systems; a coarse-grained description of a system regards large sub components while a fine-grained description regards smaller components of which the larger ones are composed. The terms granularity, coarse and fine are relative, used when comparing systems or descriptions of systems.

2.3.3 Address space

Parallel vector processor (PVP):
MIMD, UMA, large grain. small number of powerful processors connected by custom designed crossbar switch.

Uniform memory access (UMA):
All the processors in the UMA model share the physical memory uniformly. In a UMA architecture, access time to a memory location is independent of which processor makes the request or which memory chip contains the transferred data. Uniform Memory Access computer architectures are often contrasted with Non-Uniform Memory Access (NUMA) architectures. In the UMA architecture, each processor may use a private cache. Peripherals are also shared in some fashion, The UMA model is suitable for general purpose and time sharing applications by multiple users. It can be used to speed up the execution of a single large program in time critical applications.

Physical memory is uniformly shared by all processors. all processors have equal access time to all memory world. Suitable for general purpose or time-sharing application.

Non uniform memory access (NUMA):
Non-Uniform Memory Access or Non-Uniform Memory Architecture (NUMA) is a computer memory design used in multiprocessors, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors. NUMA architectures logically follow in scaling from symmetric multiprocessing (SMP) architectures.

Cache coherent NUMA (CC-NUMA):
Nearly all CPU architectures use a small amount of very fast non-shared memory known as cache to exploit locality of reference in memory accesses. With NUMA, maintaining cache coherence across shared memory has a significant overhead.

Although simpler to design and build, non-cache-coherent NUMA systems become prohibitively complex to program in the standard von Neumann architecture programming model. As a result, all NUMA computers sold to the market use special-purpose hardware to maintain cache coherence, and thus class as "cache-coherent NUMA", or CC-NUMA.

Typically, this takes place by using inter-processor communication between cache controllers to keep a consistent memory image when more than one cache stores the same memory location. For this reason, CCNUMA performs poorly when multiple processors attempt to access the same memory area in rapid succession. Operating-system support for NUMA attempts to reduce the frequency of this kind of access by allocating processors and memory in NUMA-friendly ways and by avoiding scheduling and locking algorithms that make NUMA-unfriendly accesses necessary. Alternatively, cache coherency protocols such as the MESIF protocol attempt to reduce the communication required to maintain cache coherency.

Intel announced NUMA introduction to its x86 and Itanium servers in late 2007 with Nehalem and Tukwila CPUs, both CPU families will share a common chipset; the interconnection is called Intel Quick Path Interconnect (QPI).

Distribute shared memory (DSM):
It refers to a wide class of software and hardware implementations, in which each node of a cluster has access to shared memory in addition to each node's non-shared private memory. It is also known as a distributed global address space (DGAS).

MIMD, NUMA, NORMA, large grain. Shared memory architecture. Cache directory is used to support distributed coherent caches.

Cluster of workstations (COW):
It is a computer network that connects several computer

workstations together with special software forming a cluster.

MIMD, NUMA, coarse grain. Distributed memory architecture. Each node is a complete computer. Low cost commodity network is used. There is always a local disk.

Cache only memory architecture (COMA):
It is a computer memory organization for use in multiprocessors in which the local memories (typically DRAM) at each node are used as cache. This is in contrast to using the local memories as actual main memory, as in NUMA organizations.

In NUMA, each address in the global address space is typically assigned a fixed home node. When processors access some data, a copy is made in their local cache, but space remains allocated in the home node. Instead, with COMA, there is no home. An access from a remote node may cause that data to migrate. Compared to NUMA, this reduces the number of redundant copies and may allow more efficient use of the memory resources. On the other hand, it raises problems of how to find a particular data and what to do if a local memory fills up (migrating some data into the local memory then needs to evict some other data, which doesn't have a home to go to). Hardware memory coherence mechanisms are typically used to implement the migration.

It is special case of NUMA, in which the distributed memories are converted to cache. all caches forms a global address space.

No remote memory access (NORMA):
A distributed memory multi-computer system is called a NORMA model if all memories are private and accessible only by local processors. In DSM system NORMA will disappear.

2.3.4 Memory models

Exclusive read exclusive write (EREW):

Concurrent read exclusive write (CREW):

Concurrent read Concurrent write (CRCW):

2.3.5 ACID

Atomicity: Transactions are guaranteed to either completely occur, or have no effects. **Consistency:** Always transfers from one consistent state to another. **Isolation:** Result not revealed to other transactions until committed. **Durability:** Once committed, the effect of transition persists even if system fails.

2.3.6 Overheads in parallel processing

- ⋆ Parallelism Overhead : Due to process management.
- ⋆ Communication Overhead
- ⋆ Synchronization Overhead
- ⋆ Load Imbalance Overhead

Example: *Algorithms A,B,C with complexities* $7n$, $(n \log n)/4$, $n \log\log n$ *on n-processor computer. For,* $n = 1024$ *the fastest one is B.*

2.3.7 Process

Process is a four-tuple

1. Program code
2. Control state
3. Data state
4. Status

Thread is a light-weighted process which may share the address with other threads and parent process.

2.3.8 Interaction modules

Synchronous: All participant must arrive before interaction begins. A process can exit when all other processors have finished interaction. A two barrier synchronization.

Blocking: A process can enter its portion of code as soon as it arrives and can can exit as soon as it finishes.

Non-Blocking: A process can enter its portion of code as soon as it arrives and can exit even before it has finished.

Interaction patterns : This can be point-to-point, Broadcast, Scatter, Gather, Total-exchange, Shift, Reduction. Scan.

2.3.9 Bulk Synchronous Parallel (BSP) model

Has concept of superstep (Computation, Communication, Barrier).

- ⋆ w: Maximum computation time within each superstep. Takes care of load imbalance.
- ⋆ l: Barrier synchronization overhead (lower bound of the communication network latency).
- ⋆ g: h relation coefficient(each node sends/receives at most h words within gh cycles.
- ⋆ g is platform-dependent but is independent of communication pattern.

Time for a superstep = w +gh+l or max(w, gh, l) in case of total overlapping of operations.

Example: Inner product of two N-dimensional vectors on 8-processor BSP machine.

1. **Superstep 1**
   - *Computation:* Each processor computes its local sum in w = 2N/8 cycles.
   - *Communication:* Processors 0,2,4,6 send their local sums to processors 1,3,5, 7 using (h=1) relation.

- *Barrier Synchronization*

2. **Superstep 2**
   - *Computation:* processors 1,3,5,7 perform one addition (w = 1).
   - *Communication:* Processors 1 and 5 send their local sums to processors 3 and 7 using (h=1) relation.
   - *Barrier Synchronization*
3. **Superstep 3**
   - *Computation:*Processors 3 and 7 perform one addition (w = 1)
   - *Communication:* Processor 3 sends its intermediate result to processor 7 using (h=1) relation.
   - *Barrier Synchronization*
4. **Superstep 4**
   - *Computation:* Processor 7 performs one addition (w = 1)

Execution time $= 2N/8 + 3g + 3l + 3 = 2N/n + (g + l + 1)\log n$.

2.4 Clusters

Collection of complete computers, physically interconnected by a high-performance network.

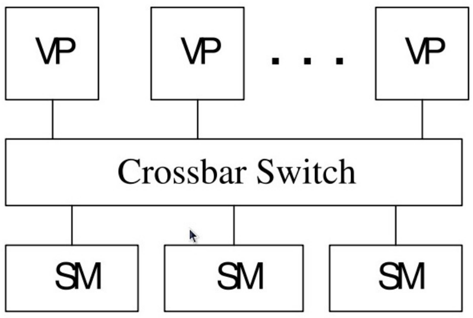
Features: Cluster Nodes, Single-System Image, Internode connection, Enhanced availability, Better Performance.

Benefits and Difficulties of Clusters: Usability, Availability, Scalable Performance, Performance/Cost Ratio.

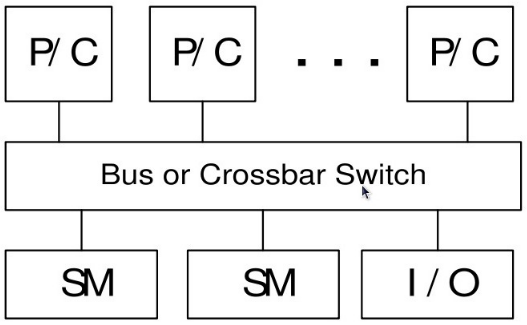
Clusters have high availability due to : Processors and Memories, Disk Arrays, Operating System. Clusters are highly scalable in terms of processors, memories, I/O devices and disks.

3 Physical machine model

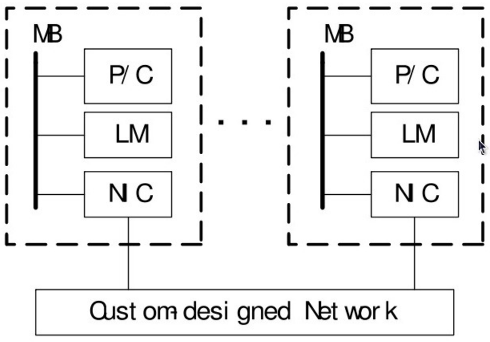
1. **PVP: Parallel Vector processor**



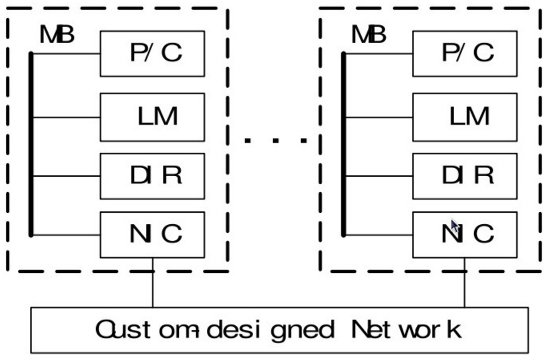
2. **SMP: Symmetric Multiprocessor**



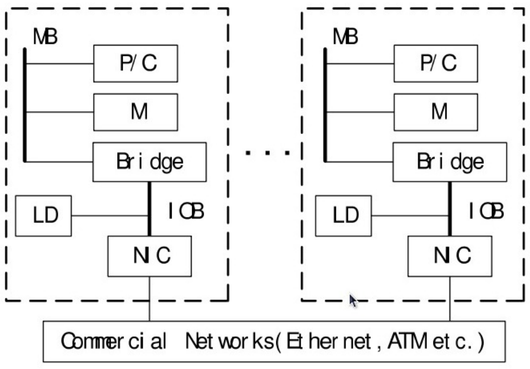
3. **MPP: Massively Parallel Processors**



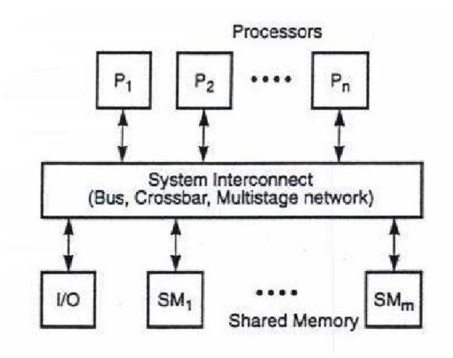
4. **DSM: Distributed Shared Memory**



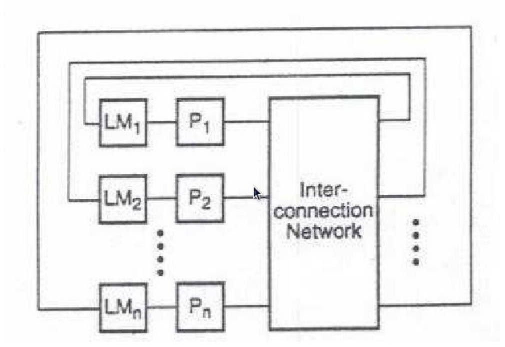
5. **COW: Cluster of Workstations**



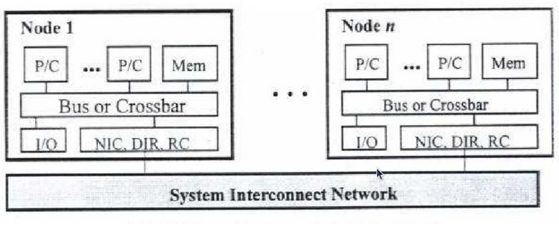
6. **Uniform Memory Access (UMA)**



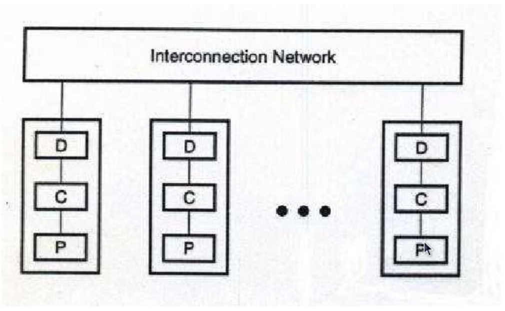
7. **Non-Uniform Memory Access (NUMA)**



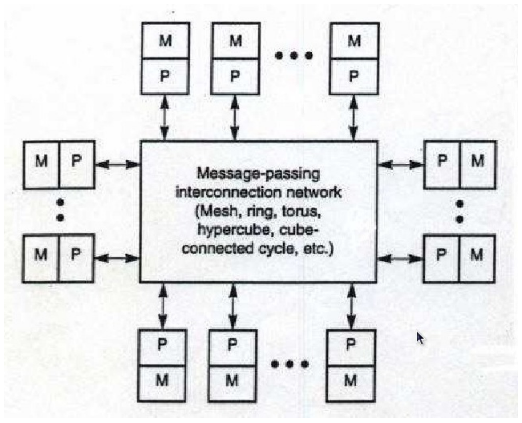
8. **CC-NUMA: Cache coherance NUMA**



9. **Cache only Memory Access (COMA)**

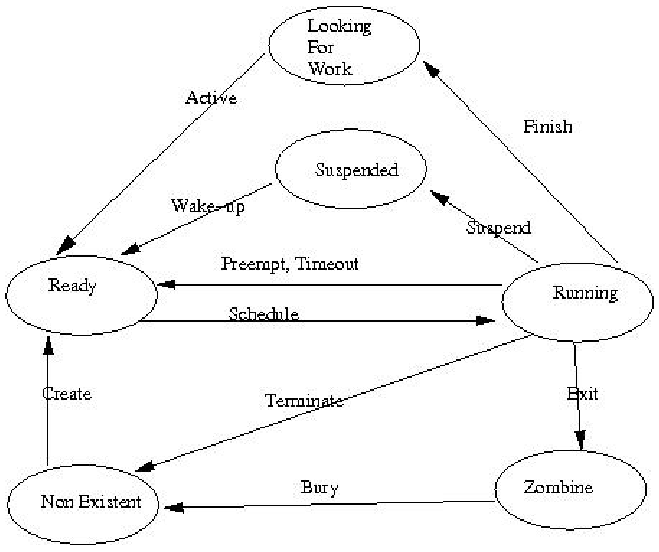


10. **No-Remote memory Access (NORMA)**


M
P
Message-passing interconnection network (Mesh, ring, torus, hypercube, cube-connected cycle, etc.)

4 Parellel Programming

4.1 Process states



4.2 Fork() Example

```
main(){
        int i=0;
        fork();
        fork();
        printf("Hello");
}
```

4.3 Parallel blocks

```
 parbegin
         S1;
         S2;

         S3;
parend

parfor( i=0 ; i<=10 ; i++ ){
      Process(i);
}
```

```
forall( i=1, N){
      C[i] = A[i] + B[i];
}

int pid = my_process_id(i);
```

4.4 Dynamic Parallelism

Uses constructs as fork() and join()

```
While( C>0 ) begin
                   fork(f1(C));
                   C=f2(C);
             end
```

4.5 Intercation/Communication Issues

4.5.1 Communication

Can be done through 1) through shared variables 2) through parameter passing 3) through message passing

4.5.2 Synchronization

Causes processes to wait for one other or allows the waiting proceses to resume.

- **Atomicity**

```
        parfor(i:=1;i<n;i++){
              atomic{
                     x=x+1;
                     y=y-1;
                    }
        }
```

- **Control Synchronization**
  Process waits until execution reaches certain control states.

```
        parfor(i:=1;i<n;i++){
               Pi; barrier; Qi;
        }
        parfor(i:=1;i<n;i++){
              critical{
                       x=x+1;
                       y=y-1;
                    }
        }
```

- **Data Synchronization**
  Process waits until execution reaches certain data states.

```
        wait(x>1)
        parfor(i:=1;i<n;i++){
        lock(S);x=x+1;y=y-1;unlock(S);}
```

4.5.3 Aggregation

To merge partial results generated by component processes. Can be realized as a sequence of supersteps and communication/synchronization.

```
      parfor(i:=1;i<n;i++){
            x[i]:=A[i]*B[i];
            inner product=aggregate_sum(x[i]);
      }
```

4.6 Commonly used interaction modes

4.6.1 Synchronous

All participants must arrive before the interaction begins. A process can exit the interaction and continue to execute the subsequent operation only when all other processes have finished the interaction. Basically, a two-barrier synchronization. Example: send/receive operations.

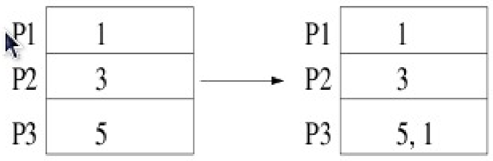
4.6.2 Blocking

A process can enter its portion of C as soon as it arrives and can exit as soon as it finishes irrespective of status of other processes. Example: Blocking send (completion means sending the message not necessarily yet received by destination.

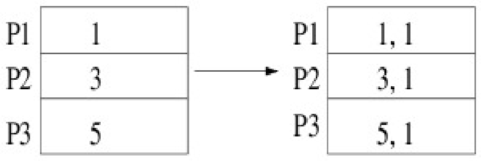
4.6.3 Non-blocking

A process can enter its portion of C as soon as it arrives and can exit even before it has finished its portion of C. Example: Non-blocking send (completion means requesting the sending of the message and not necessarily sending the message).

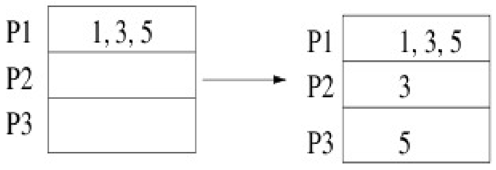
4.7 Interaction Patterns



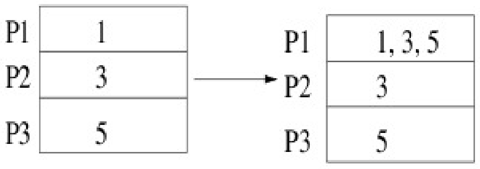
(a) Point–to–point: P1 sends 1 to P3



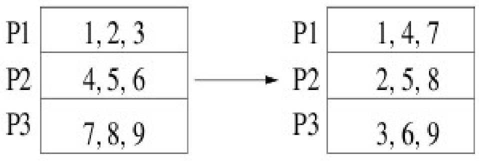
(b)Broadcast: P1 sends 1 to all



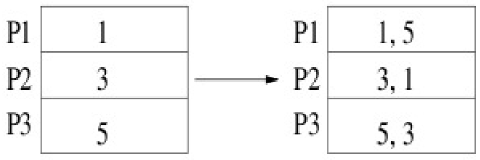
(c) Scatter: P1 sends one value to each node



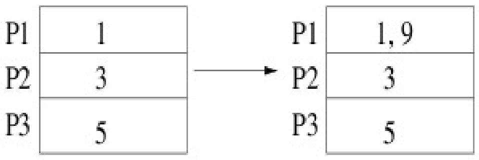
(d)Gather: P1 gets one value from each node



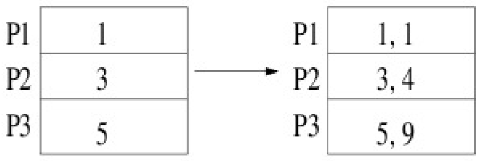
(e)Total exchange: each node sends a distinct message to every node



(f) Shift: each node sends one value to the next and receives one from the previous



(g)Reduction: P1 gets the sum 1 + 3 + 5 = 9



(h)Scan: P1 gets 1, P2 gets 1 + 3 = 4, and P3 gets 1 + 3 + 5 = 9

5 Distributed memory and latency tolerance

5.1 Memory Hierarchy Properties

Three fundamental properties for designing an efficient memory hierarchy.

- **Inclusion** $M_1 \subset M_2 \subset M_3 ...... \subset M_n$
- **Coherence** Same information at all levels. Effects at memory/cache replacement policies.
  - Write-through (mainly for on-chip I-cache and D-cache)
  - Write-back (Off-chip cache)
- **Locality of Reference**
  1. Temporal locality: favours LRU replacement
  2. Spatial locality assists in determining the size of unit data transfer
  3. Sequential locality affects the detrmination of grain size for optimal scheduling and also affects prefetch techniques.

5.2 Memory Capacity Planning

- **Hit ratio**
  h_i at level i represents the probability of finding information item at level i. $h_0 = 0, h_n = 1$.
- **Access friquency to M_i**

$$f_i = (1 - h_1)(1 - h_2)....(1 - h_{i-1})h_i$$

$$f_1 \gg f_2 \gg f_3 \gg f_4 \gg .... \gg f_n$$

- **Effective Access Time**

$$T_{\text{eff}} = \Sigma_{i=1}^{n} f_i.t_i$$

- **Hirarchy Optimization**
  Minimize T_{eff} subject to following

$$C_{\text{total}} = \Sigma_{i=1}^{n} c_i.S_i < C_0$$

Where S_i is size of memory M_i, and c_i is per unit cost of M_i, and C_0 is some value.

5.3 Cache Coherence Protocols

5.3.1 Sources of Incoherence

- The write by different processors into their cached copies of the same cache line in memory, asynschronously.
- Process migration among multiple processors without alerting each other.
- I/O operations bypassing the owners of cached copies.

5.3.2 Snoopy Coherency Protocols

Constantly monitor the caching events across the bus between processor and memory modules. Generally implemented using snoopy buses and require a broadcast mechanism. Write-Invalidate and Write-Update protocols.

5.3.3 MESI Snoopy Protocol

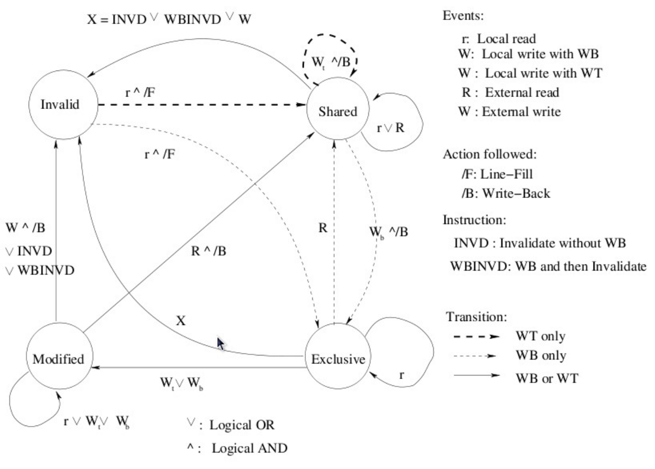
Keeps track of state of cache line, considering all reads/writes, cache hits/misses etc. Write-invalidate protocol. Uses non-write-allocate policy: no fill from the memory in case of write-miss. Cache line can go in following stages.

Modified (M) cache line updated with a write hit.

Exclusive(E) Cache is valid and is invalid in other caches. Memory not updated yet.

Shared(S) Cache is valid and may be valid in one or more caches and memory.

Invalid(I) Cache is invalid after reset or has been invalidated by write hit by another cache.



State transition diagram of the MESI protocol for data cache in a Pentium–based multiprocessor

5.3.4 Memory consistancy models

- Sequential Consistency
  All reads, writes and swaps by all processors appera to execute serially in a single global memory order confirming to program order.
- Weak Consistency
  Relates memory order to synchronization points.
- Processor Consistency
  Write issued by each individual processor are always in program order, but writes issued by different processors can be out of program order.
- Release consistency
  Requires that synchronization accesses in program can be identified either acquires(locks) or releases(unlocks).

To reduce the overhead of cache coherence control without affecting the correctness a Relaxed Memory is also proposed.

5.3.5 Software-implemented DSM

Software-coherent NUMA (SC-NUMA) or Distributed Shared Memory (DSM) model proposed to enable Shared Memory computing on NORMA and NCC-NUMA. Software-implemented DSM uses SW extensions to achieve single address space, data sharing and coherence control. In shared virtual memory (SVM), virtual memory management mechanism in a traditional node OS is modified to provide data sharing at page level.

Alternatively, instead of modifying OS, compilers and library functions are used to convert single-address-space codes to run on multiple address spaces. Application codes may also have to be modified to include data sharing, synchroniztion and coherence primitives.

5.3.6 Directory-Based Coherency Protocol

Dont use broadcast-based snoopy protocols. Cache Directory: Directory to record the locations and states of all cached lines of shared data. A directory entry for each cache line of data containing dirty bit and number of pointers to specify the location of all remote copies. Concept of central directory to record all cache conditions (including the presence information and all cache line states) suitable only for a small-scale SMP with centralized shared memory. Central directory for large-cache SMP, NUMA or distributed memory platform would require a huge memory to implement and must be associatively searched to reduce the update time.

Cache Directory Structure can be 1) Full map cache directory 2) Limited cache directory 3) Chained cache directory

5.4 Latency Tolerance Techniques

1. Latency Avoidance
   By organizing user applications to achieve data/program locality to avoid long latency of remote access.

2. Latency Reduction
   Data locality may be limited, Difficult to discover and change. Latency reduction is achieved by making communication subsystem efficient.

3. Latency Hiding
   Prefetching Techniques, Distributed Coherent Caches, Relaxed Memory Consistency, Multiple-context processors.

5.5 Data Prefetching

Using knowledge about the expected misses in a program. Controlled by SW/HW.

1. Binding Prefetch
   Value directly loaded into working register during prefetch. Value may become stale if another processor modifies the location between prefetch and actual reference.

2. Non-binding Prefetch
   Brings the data to the cache remaining visible to cache coherence protocol(CCP). HW controlled prefetch includes schemes such as long cache lines (effectiveness limited by reduced spatial locality in multiprocessor applications) and instruction lookahead (effectiveness limited by branches and finite buffer size).

5.6 Context-switching policies

1. Switch on cache miss
   R: Av. interval between misses, L: Recovery time.

2. Switch on every load
   R: Av. interval between loads. L1, L2: Latencies with and without misses. p1, p2: prob. of switch with or without miss.

3. Switch on every instruction
   Interleaving of instructions. Supports pipelined execution but cache misses may increase due to breaking of locality. Can hide pipeline dependence.

4. Switch on block of instructions
   Improved cache-hit ratio due to preservance of locality.

6 System Interconnect and Network Topologies

6.1 Switched Networks

Allocate/deallocate media resources to one request at a time. Even shared-media networks can be converted to switched networks.

1. Circuit switched networks
   Entire path from source node to destination reserved for entire period of transmission.

2. Packet switched networks
   Long messages broken into sequence of smaller packets containing routing information and segment of data payload. Packets can be routed separately on different paths. Better utilization of resources but needs disassembly and reassembly of messages. Different messages may have different packet sizes.

3. Cell switched networks
   Partitioning of long packets into fixed-size small cells. Small packets need not wait for long packets. Constant transmission delay is possible. Simplified HW design of cell switches because of fixed size. May lead to low network performance if retransmission of lost cells not allowed.

6.2 Network Performance Metrics or Communication Latency

1. SW Overhead associated with sending and receiving of messages at bot ends of network. Contributed by host kernels in handling the message.

2. Channel Delay caused by channel occupancy. Determined by bottleneck link or channel.

3. Routing Delay caused by successive switches in making sequence of routing decisions along the routing path. Depends on routing distance.

4. Contention Delay caused by traffic contentions in the network. Difficult to predict. Network Latency Sum of Channel Delay and Routing Delay.

Network Latency is sum of Channel Delay and Routing Delay.

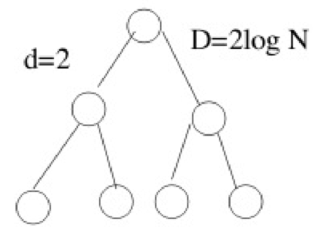
6.3 Routing Schemes and Functions

1. Store-and-Forward Routing
   Packet stored in packet buffer in a node before forwarding to outgoing link. Non-overlapped transmission of packets. Header determines the routing path and then entire packet forwarded to next node.

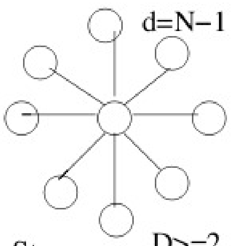
2. Cut-Through or Wormhole Routing
   Each node uses a flit buffer to hold one flit or cell of the packet which is automatically forwarded through an out-link to next node once the header is decoded. Rest of the data flits of the packet follow the same path in a pipelined fashion reducing the transmission time significantly.

6.4 Network Topologies

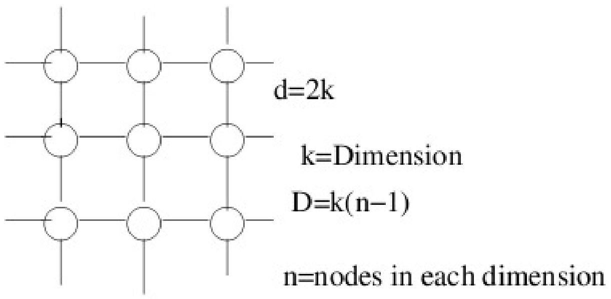
Binary Tree, star, near neibour mesh, systolic array, illiac mesh, 2-D tauras, cube, binary 4-cube.



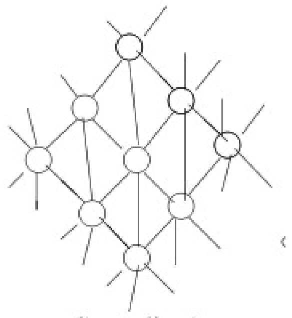
Binary Tree



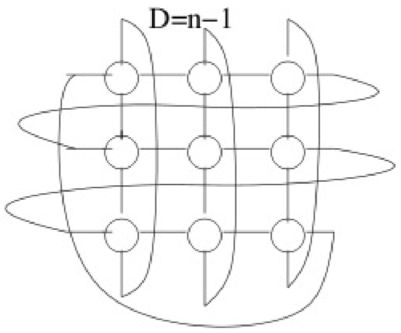
Star



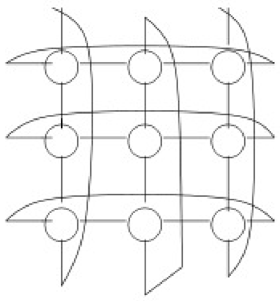
Near−neighbour Mesh



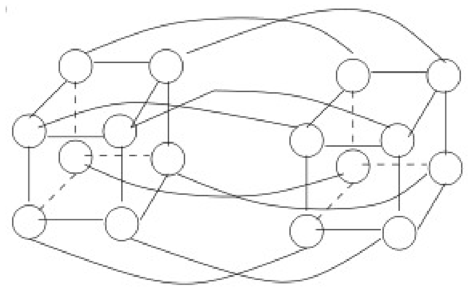
Systolic Array



Illiac−Mesh
(Chordal ring of degree 4)



2−D−Tauras



Binary 4 −cube

Kind of networks.

- Multistage Interconnection Networks
- Mesh Connected Illiac Network
- Cube Interconnection Network
- Shuffle-exchange and Omega Networks

Matrix Multiplication Cube-network is done as below.

1. Let p2m1....pmpm1...p1p0 be the PE address in 2m cube. Distribute n rows of A over n distinct PEs whose address satisfy the condition: p2m1....pm = pm1...p1p0
2. Broadcast rows of A over the fourth dimension and front to back edges.
3. Transpose B to form B t over the m cubes x2m1....xm0....0 in nlogn steps.
4. N way broadcast each row of bt to all PEs in the m cube p2m1....pmxm1...0 in nlogn steps.
5. Get the multiplication in O(n) time.