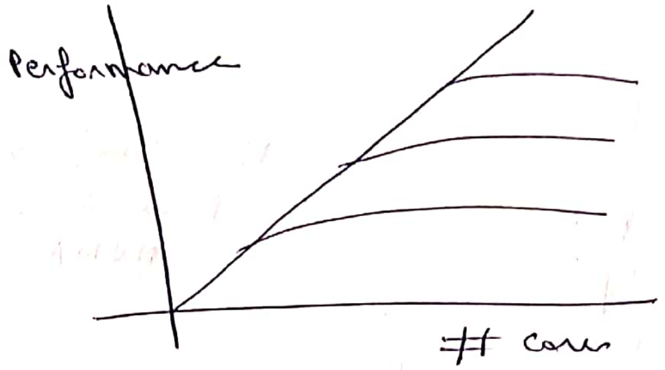
Multi Processing

Flynn's Taxonomy of Parallel machines.

		Instruction Stream	Data Stream
uniprocessor	SISD	1	1
vector SSE/MMX	SIMD	1	>1
Stream Processor —	MISD	>1	1 (not really used much)
	MIMD Multi Processor	>1	>1

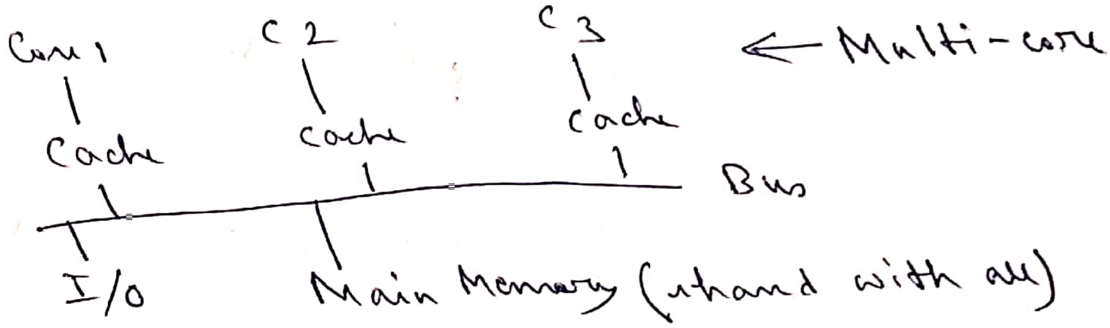
Multi Processors need parallel programs

original single threaded code → difficult parallel code development difficult debugging also difficult



difficult to write scalable parallel code.

Centralized Shared Mem



UMA : uniform memory access (time)

SMP : Symmetric Multi Processor.

Problems with centralized memory

A large memory is slow (access time more)

Memory B/W — all cores try to access one big slow mem

— B/W contention

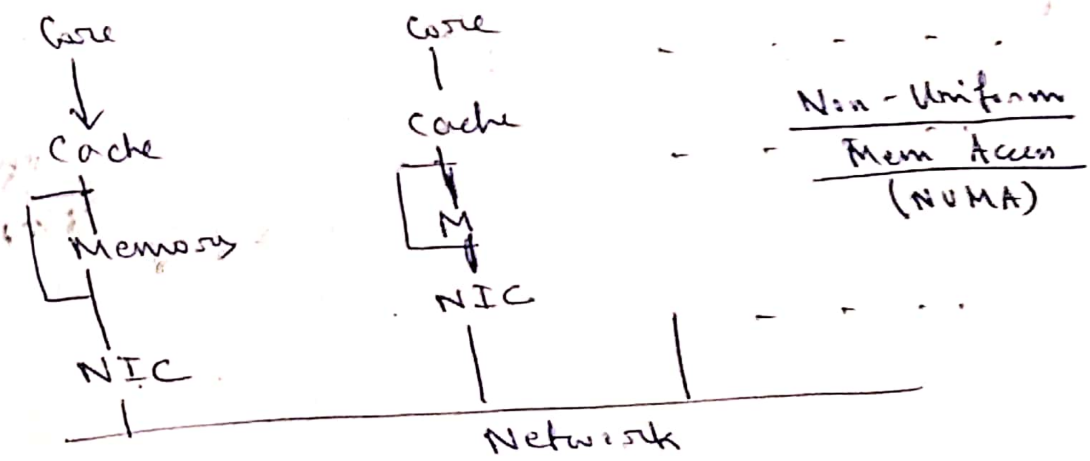
— accesses get serialized. (queued)

∴ Centralized main memory is not good solution for machines with large no. of cores.

ok for 2~16 cores.

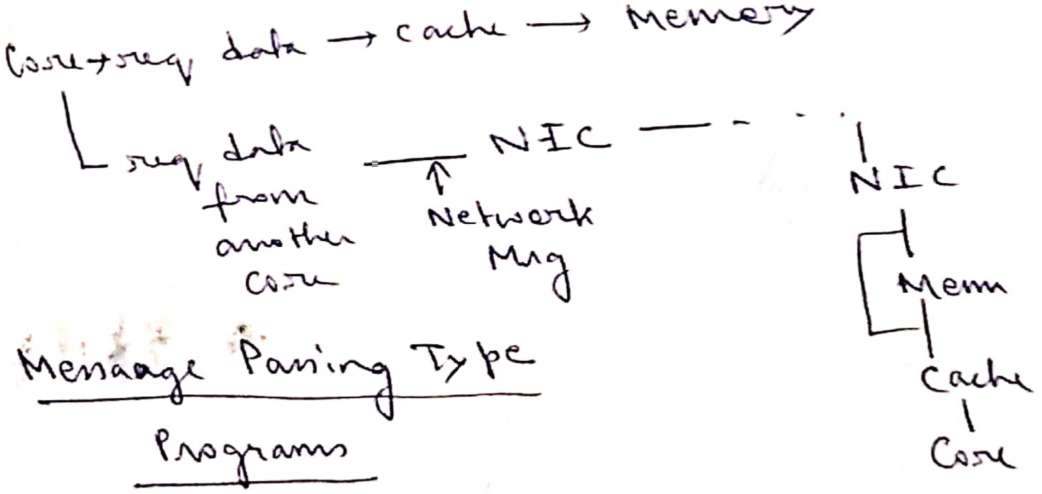
↓

Distributed ~~Shared~~ Memory



Non-Uniform Mem Access (NUMA)

Multi-computer /Cluster Computer



Message Passing Type Programs

Scalable Solution → support large # processors.

↓

programmer manages communication explicitly.

In shared mem, this is oblivious to the programmar.

Ex of MP program (message passing)

```
# define ASIZE 1024
# define Numproc 4
double myArray (ASIZE/Numproc);
double mySum = 0;   // local sum variable

for (int i=0; i< ASIZE/Numproc; i++)
    mySum += myArray[i];   // local sum
if (myPID = 0) {
    for (int p=1; p< Numproc; p++) {
        int pSum; recv (p, pSum);
        mySum += pSum;
    }
    printf("Sum: %lf \n", mySum);
}
else
send (0, mySum);
```

assume each proc already has 1/4 of array

one proc. compute overall sum

Shared Mem Program

```
#define ASIZE 1024
#define Numproc 4
shared double array[ASIZE];   ← array in shared mem
shared double allSum=0;
shared mutex sumLock;
double ~~shared~~ mySum = 0;
for(int i = myPID*ASIZE/Numproc;
        i<(myPID+1)*ASIZE/Numproc; i++)
        mySum += array[i];
lock(sumLock);           } critical section
allSum += mySum;
unlock(sumLock);
if(myPID = 0) print("Sum: %lf \n", allSum);
```

job /proc →

i) no send /receive

ii) no array distribution

insert a barrier here

	Message Passing	Shared Mem
Communication	Programmar	Auto
Data distribution	Manual	Auto
HW support	Simple (need network HW)	Extensive
Program Correctness	Difficult to guarantee	Less difficult
Program Performance	Difficult	Very diff

↓

with pcorrectness, performance also comes.

Shared Mem HW

Hyperthreading / Simultaneous Multi-threading (SMT)

using core-level HW to i) switch among threads across clock ~~cly~~ cycles.

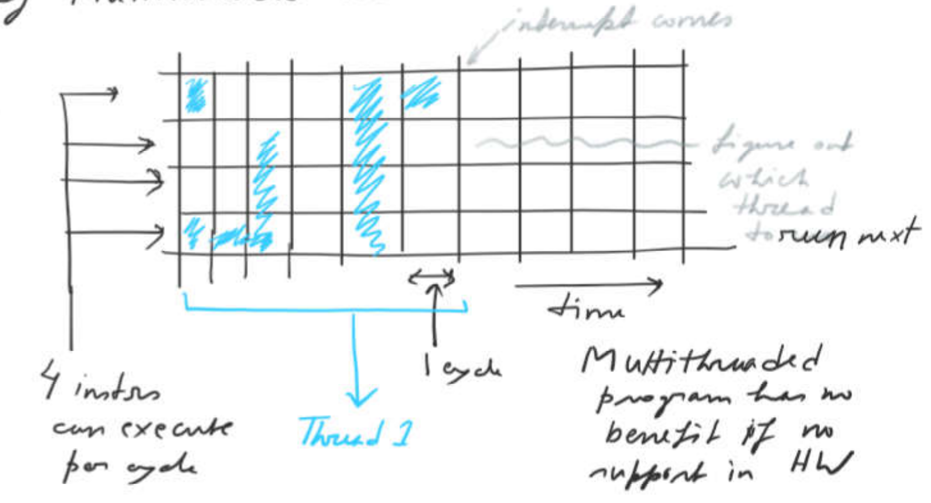
ii) execute multiple thread context in parallel. (saving & restoring registers for thread contexts)

HPCA notes

19 April 2020 13:43

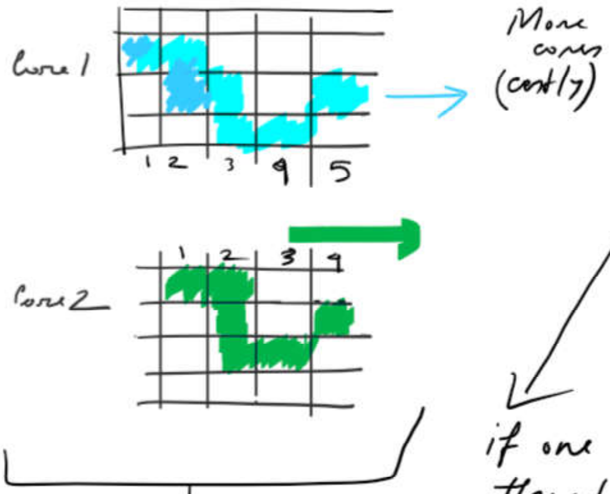
Performance of Multithreaded code

Processor with no MT support

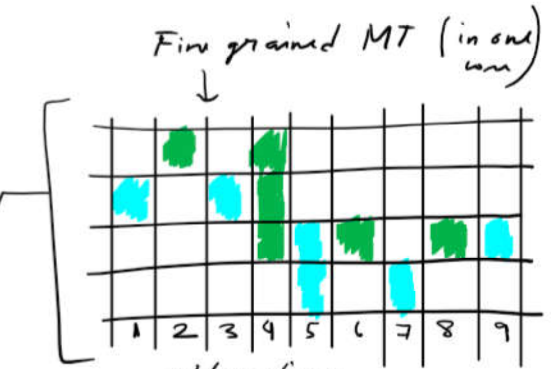

interrupt comes
figure out which thread to run next
time
1 cycle
4 instrs can execute per cycle
Thread 1

Multithreaded program has no benefit if no support in HW

Chip Multiprocessor (CMP)



Fine grained MT (in one core)



alternating (in every cycle) among threads using HW support

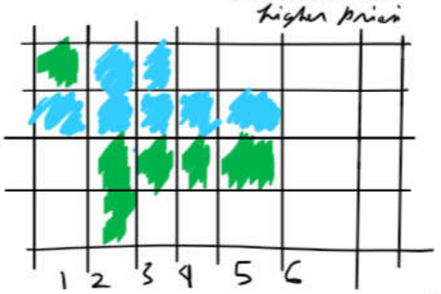
Coarsegrain — alternate among threads after multiple cycles, not after every cycle

if one thread has lot of stalls (due to cache miss & other causes)

↳ other threads with 'ready' instructions can fill in the next cycles.

SMT ↓

Thread 1 has higher priori



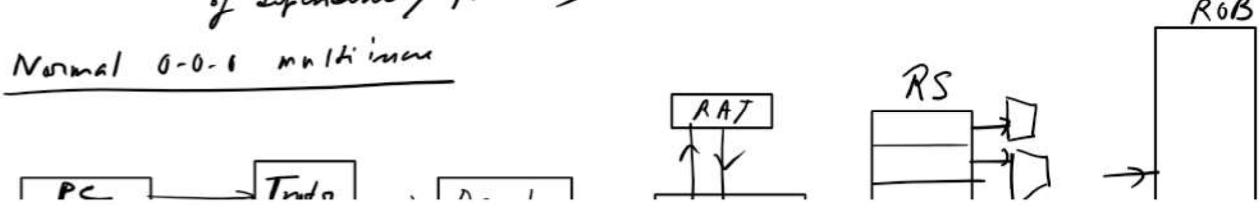
using the underutilized issue slots of the multi-issue 4 way o-o-o superscalar processor.

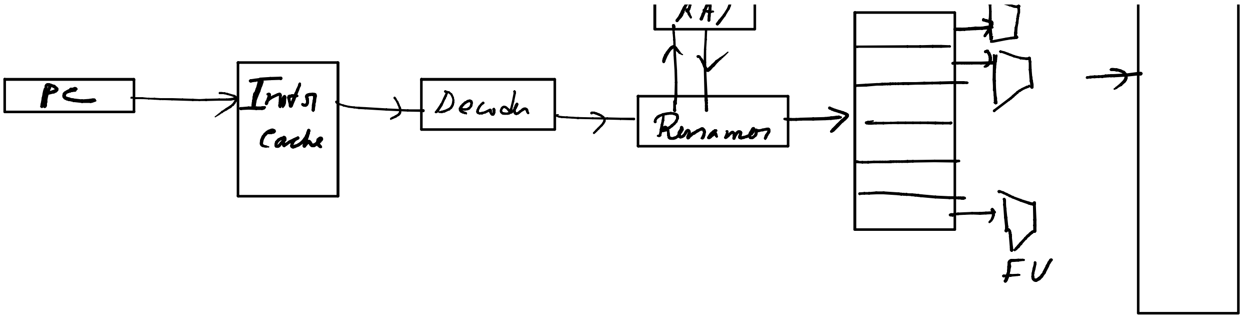
HW cost → fetch stage need to be able to fetch from two PCs.

Register files → 2 set of Architectural registers.

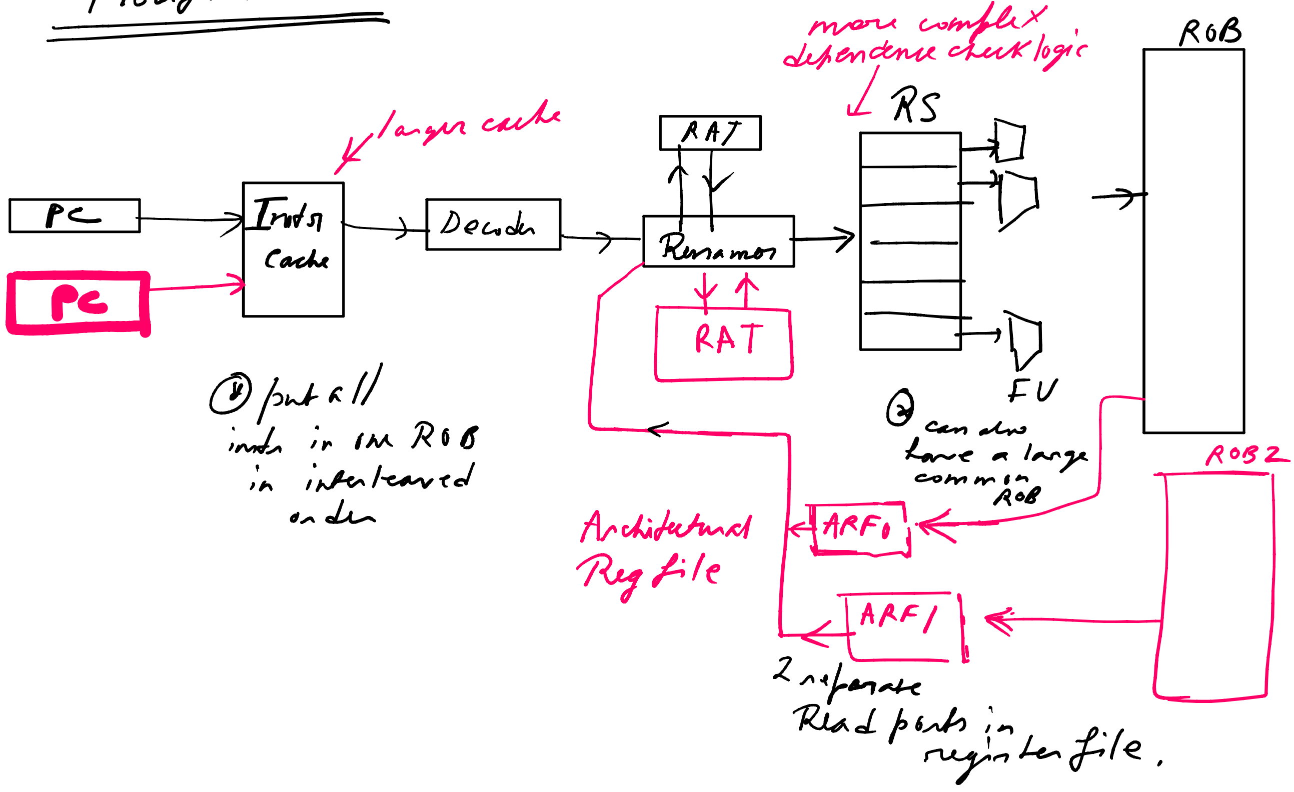
Performance almost like CMP, little cost overhead of dependence/priority check during issue slot fill-up.

Normal o-o-o multi issue





Modifications



larger cache

more complex dependence check logic

① put all instrs in the ROB in interleaved order

② can also have a large common ROB

Architectural Reg file

2 separate Read ports in register file.