Multiprocessors

- 작은 컴퓨터 여러개를 연결하여 강력한 성능의 컴퓨터 하나를 만드는 것이 기본 idea
- 장점 : Scalability, availability, power efficiency
  확장성 가용성

① Task level (process level) parallelism → high throughput for independent jobs
② Parallel processing Program → single program running on multiple processors
③ Multicore microprocessors → chips with multiple processors (cores)

Hardware ┌ serial (직렬)
         └ parallel (병렬)

Software ┌ Sequential (순차적)
         └ Cocurrent (병행적)

		Software	
		Sequential	Concurrent
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Core i7	Windows Vista Operating System running on an Intel Core i7

Parallel Programming

→ 병렬처리의 어려움은 하드웨어보다 소프트웨어로 인한 것
모든 application을 parallel 프로그래밍할 수 있는가? ⇒ X 특히 종속성이 있는 경우

- 병렬 프로그램의 사용 → significant performance improvement 요구된다
  그렇지 못하면 그냥 빠른 uniprocessor를 사용하는 쉬운 방법이 낫다
- Difficulties ┌ partitioning – 작업을 균등하게 나누기 어렵다
               ├ Coordination – 협업에 실패하면 오히려 단일 코어보다 성능이 떨어질수도…
               └ Communication Overhead

Amdahl's Law : sequential part can limit speedup
프로그램이 여러 프로세서를 효율적으로 사용하기 위해서는
프로그램의 작은 부분까지도 병렬화되어야 한다

→ ex) 100 processors 이용하여 90배의 성능개선하기

$$\text{개선 후 수행시간} = \frac{\text{개선에 의해 영향을 받는 수행시간}}{\text{개선의 크기}} + \text{영향을 받지 않는 수행시간}$$

$$\text{속도 개선} = \frac{1}{(1-\text{개선될 부분의 비율}) + \frac{\text{개선될 부분의 비율}}{\text{개선의 크기}}}$$

$$\Rightarrow 90 = \frac{1}{(1-x) + \frac{x}{100}}$$

↝ 개선될 부분의 비율 $x = 0.999\cdots$
즉, 전체프로그램에서 순차적 부분이 0.1% 미만
need sequential part to be 0.1%
of the original time

Scaling Example

- Work load : sum of 10 scalars & 10 x 10 matrix sum
  ⇒ speed up from 10 to 100 processors

Single processor : Time = (10 + 100) x tadd
10 processor : Time = (10 + 100/10) x tadd = 20 x tadd
Speed up = 110/20 = 5.5 (55% of the potential)
100 processor : Time = (10 + 100/100) x tadd = 11 x tadd
Speed up = 110/11 = 10 (10% of the potential)
Assumes load can be balanced across processors

- 100 x 100 matrix일 경우
  Single processor : Time = (10 + 10000) x tadd = 10010 tadd
  10 processors : Time = (10 + 10000/10) x tadd = 1010 tadd
  Speed up = 10010/1010 = 9.9
  100 processors : Time = (10 + 10000/100) x tadd = 110 tadd
  Speed up = 10010/110 = 91

Strong vs Weak Scaling

- Strong scaling : problem size fixed
  문제의 크기를 고정시킨 상태에서 얻어지는 속도 개선
- Weak scaling : 프로세서의 수에 비례하여 문제의 크기를 증가시키는 경우의 속도 개선

Instruction and Data Streams

- An alternate classification

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD Intel Pentium 4	SIMD ** SSE Instructions of X86
	Multiple	MISD No Example Today	MIMD Intel Xeon e5345

→ 벡터데이터 연산에 유용

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

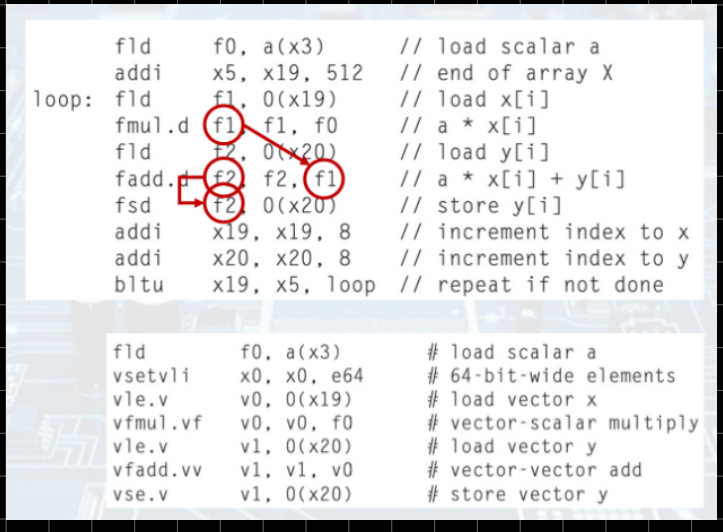
Vector Processors
- highly piplined function units
- Stream data from/to vector registers to units
  - Data collected from memory to registers
  - Results stored from registers to memory

메모리에서 데이터 원소들을 꺼내서 큰 레지스터 집합에 저장
레지스터에 있는 데이터에 대해 파이프라인 실행 유닛 사용해 순차적으로 연산
결과를 다시 메모리에 저장

ex) vector extension to RISC-V
- v0 ~ v31 : 32 x 64-element registers (64 bit elements)
  64개의 element를 한 번에 계산할 수 있는 instruction set을 지원
  64 bit element 64개를 저장할 수 있는 벡터 레지스터 32개
- vector instructions
  - fld.v, fsd.v : load / store vector
  - fadd.d.v : add vectors of double
  - fadd.d.vs : add scalar to each element of vector of double

⇒ significantly reduces instruction-fetch bandwidth
… 한 번에 64개의 연산 수행이 가능

ex) Y = a x X + Y 연산



```
        fld     f0, a(x3)       // load scalar a
        addi    x5, x19, 512    // end of array X
loop:   fld     f1, 0(x19)      // load x[i]
        fmul.d  f1, f1, f0      // a * x[i]
        fld     f2, 0(x20)      // load y[i]
        fadd.d  f2, f2, f1      // a * x[i] + y[i]
        fsd     f2, 0(x20)      // store y[i]
        addi    x19, x19, 8     // increment index to x
        addi    x20, x20, 8     // increment index to y
        bltu    x19, x5, loop   // repeat if not done
```

```
fld       f0, a(x3)       # load scalar a
vsetvli   x0, x0, e64     # 64-bit-wide elements
vle.v     v0, 0(x19)      # load vector x
vfmul.vf  v0, v0, f0      # vector-scalar multiply
vle.v     v1, 0(x20)      # load vector y
vfadd.vv  v1, v1, v0      # vector-vector add
vse.v     v1, 0(x20)      # store vector y
```

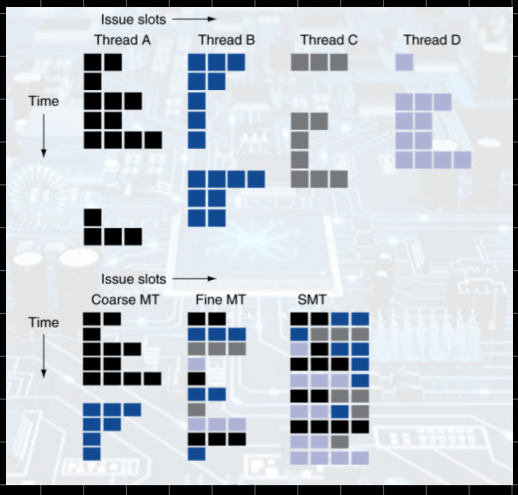
Vector vs Scalar
- Vector archtecture and Compilers
  - Simplify data-parallel programming
  - explicit statement of absence of loop-carried dependencies
    → reduced checking in hardware
  - regular access patterns benefit from interleaved and burst memory
  - avoid control hazards by avoiding loops

- more general than ad-hoc media extensions (MMX, SSE)
  → better match with compiler technology

SIMD : Single Instruction Multiple Data
- operate elementwise on vectors of data
  ex) MMX, SSE instructions in x86
      Multiple data Elements in 128 bit wide registers
- All processors execute the same instruction at the same time
  - each with different data access
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Vector vs Multimedia Extensions
- (vector instructions → variable vector width
  multimedia extensions → fixed width
- (V : support strided access
  M : X
- Vector units can be combination of pipelined and arrayed functional units

Multithreading - 하나의 CPU에 task 여러개
- performing multiple threads of execution in parallel
  → replicate registers, PC, etc
    fast switching between threads



- Fine-grain Multithreading
  - switch threads after each cycle
  - interleave instruction execution
  - if one thread stalls, others executed

- coarse-grain multithreading
  - only switch on long stall ex) L2 cache miss로 메인 메모리 접근하는 경우
  - simplifies hardware, x hide short stalls ex) data hazard

- simultaneous Multithreading
  - multiple-issue, dynamic scheduling 지원하는 pipeline processor 사용
  - schedule instructions from multiple threads
  - instructions from independent threads execute when function units are available
  - within threads, dependencies handled by scheduling & register remaining

Shared Memory → SMP : Shared memory multiprocessor
- Hardware provides single physical address space for all processors
- Synchronize shared variables using locks
- Memory access time : UMA (uniform) vs NUMA (Non uniform)
  - 메인 메모리 접근 시간에 따라 분류
  < UMA 어떤 프로세서가 어떤 워드를 접근하든 균일한 시간이 걸린다
  NUMA 접근시간 균일 x

GPUs - 그래픽 연산을 빠르게 하기위해
- Graphics Rendering

GPU Architectures
- Processing - highly data-parallel
  → highly multithreaded
  → use thread switching to hide memory latency
    - less relience on multi level caches
  → graphic memory : high bandwidth