Topics:
- Parallel Machines
- UMA (Shared Memory)
- NUMA (Distributed Memory)
  - ↳ Message Passing

Fault Tolerance → Multi-Processing → Cache Coherence

- Coarse Grain Threading
- Fine Grain Threading
- SMT (aka Hyper-Threading)
- Multi-Threading Performance

$P \approx V^2 f \approx f^3$

Multi-Processing

Multiprocessors can have shared or distributed memory.
With a shared memory the programmer must ensure the processors cannot access the same memory location at the same time. Communication between processors is done through the shared memory.

With distributed memory the processors must communicate through message passing.

"Taxonomy" = "The branch of science concerned with classification, especially of organisms; systematics". Flynn's Taxonomy distinguishes between parallel machines according to how many instruction streams they have, and how many data streams these instruction streams operate on.

Flynn's Taxonomy of Parallel Machines

SISD - Single Instruction, Single Data - typical single core processor
Each instruction operates on 1 data stream. This is a normal Uniprocessor. A single core machine.

SIMD - SIngle Instruction, Multiple Data - vector processor
Instead of operating on normal scalar values, these instructions operate on vectors. For example, an add instruction that operates on two source and one destination vector. Operates on many items at a time.

MISD - Multiple Instruction, Single Data - stream processor, not used very often

MIMD - Multiple Instruction, Multiple Data - multi-processor
Normal multiprocessor where each processor has its own program that its running with its own program counter and so on, and each of them operates on its own data. They don't have to do things in lockstep to operate on the data. Multi-core processor

Why Multi-processors

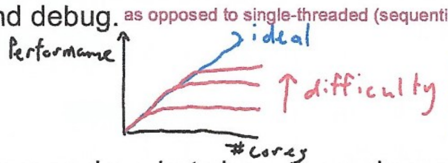
Uniprocessors are preferable, but are limited by the frequency and instruction width.

Multiprocessor Needs Parallel Programs

Disadvantages of Multiprocessors:

Parallel code is much harder to develop and debug. as opposed to single-threaded (sequential) code, which is easier.

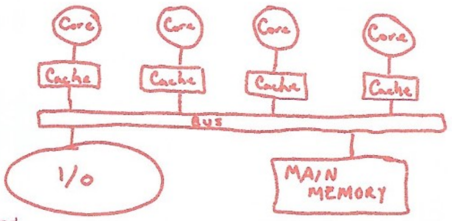
Performance Scaling is difficult to achieve



Ideally, performance would scale with the # of cores, but actually it tends to flatline and is more difficult to squeeze more performance out.

Centralized Shared Memory

UMA - Uniform Memory Access Time: All cores have caches, but share the main memory. Each core can share information by writing to the main memory.
Also called Symmetric Multiprocessor (SMP)



The Problems with Centralized Main Memory

Memory needs to be large (and is therefore slow)

Memory get too many accesses/second

Memory bandwidth contention Ex: If multiple cores have misses and need to go to memory at the same time.

Centralized Main Memory works only for small numbers of cores (<16)

Distributed ~~Shared~~ Memory

Also known as "Multi-Computer" or "Cluster Computer" since each core is pretty much a single uniprocessor computer.

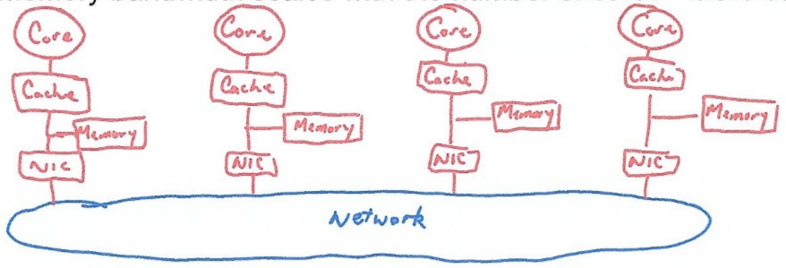
Non-Uniform Memory Access (NUMA): Each core will have its own cache and a slice of the main memory.

Cores do not share memory, they must explicitly send a message across the network to the other cores in order to access data that doesn't belong to its memory. These NICs and the network is typically much faster than a typical internet connection. This type of configuration forces the programming to think about memory accesses.

Advantages:

Each core will have fast access to its own slice of main memory.

The memory bandwidth scales with the number of cores - more cores, more bandwidth.



With NUMA the operating system should put the stack and data pages for core N in the memory slice associated with core N.

Distributed Memory

Multi-computer or Cluster computer: There is no sharing of memory, each core has its own cache, memory, and a network interface card. A network message must be used for communicating between cores.

This structure forces the programmer to think about the communication between computers.

A Message Passing Program (for Distributed Memory)

With an array manipulation program, each core gets a portion of the array. The cores each work on their portion and send the result to one processor for compilation.

For example, a "summing" program, where each core processes a portion of an array and calculates the sum. Then they all send their results to a single designated core, which then receives it and adds all the sub-results together.

Message passing ~~requires that messages be acknowledged between cores~~.

There must be an explicit "send" command called on sending cores while the receiving core is running a program with a corresponding "receive" command.

Shared Memory Program (for Centralized Memory aka Symmetric Multi-processor)

The shared memory must be locked and unlocked so that only one core at a time can access a memory location.

Barriers are required to make sure all the cores have completed their task before moving on to the next task. A Barrier would be like a semaphore, where each of the other cores could modify it once to signal that they've completed their part.

Message Passing vs Shared Memory

Message Passing:

- Communication is done by the programmer
- Data Distribution is done manually by the programmer
- HW Support is simple
- Program Correctness is difficult
- Program Performance is difficult

Shared Memory

- Communication between cores is automatically part of the system
- Data Distribution is automatically part of the system
- HW Support is extensive
- Programming Correctness is less difficult than message passing
- Programming Performance is very difficult to achieve

For Data distribution the number of lines of code:

Message Passing > Shared Memory   A program written for a Distributed Memory Processor will have to include code to distribute the Data across to the other cores. A SMP (shared memory) doesn't need to do this.

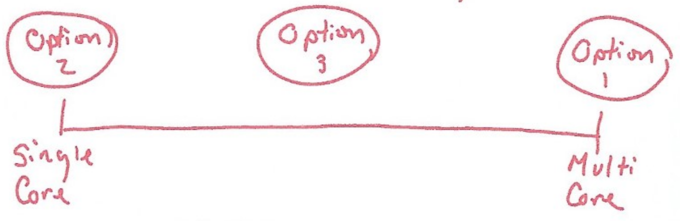
Synchronization adds

No lines of code in message passing

Extra code needed for shared memory

However, with SMP there is a need for the Programmer to explicitly synchronize access to shared memory across multiple cores. This Synchronization could occur though Synchronization primitives like Mutexes & Semaphores. There's no need to do Synchronization for Distributed Memory, there just need to be explicit "send" calls.

Shared Memory Multi-Processing Spectrum



Shared Memory Hardware

Shared memory needs extensive hardware support

Types of Shared Memory:

Option 1. Multiple cores share the same physical address space ex: UMA, NUMA

Option 2. Multi-Threading by time sharing a core This is a Single Core that is switching between different threads. This requires us to swap registers values every time we switch to a different thread.

Option 3. Hardware Multi-threading with coarse grain threading, fine grain threading,or simultaneous multi-threading This would have multiple "sets" of hardware registers, where each set would correspond to a different thread. If we only had 1 set of hardware registers, we'd need to swap out these register values whenever we switched threads.

Coarse Grain Threading: Switch between execution of thread (switch between "sets" of registers) every few cycles.
Fine Grain Threading: Switch between execution of threads (between "sets" of registers) EVERY cycle.
SMT/Hyper-Threading: In any given cycle, we could be doing instructions that belong to these different threads.

Multi-Threading Performance

Multi-threading: A core switches between threads, this makes it appear that the tasks are completed simultaneously.

In SMT: a core can execute multiple threads at the same time. (thanks to Hardware support of having multiple sets of Registers)

SMT Hardware Changes

SMT is not much more expensive than a UMA or Fine Grained Multi-threaded core.

For an SMT machine:

Add a program counter
Add a RAT
Add architectural registers

So we just need to add a few pieces to the pipeline for SMT, which is cheaper than having an entirely new pipeline for multiple cores.

VIVT = Virtually Indexed, Virtually Tagged Cache
VIPT = Virtually Indexed, Physically Tagged Cache
PIPT = Physically Indexed, Physically Tagged

SMT, Data Cache, and TLB

Since we're using the Virtual Address (all processes can have the same virtual addresses) to check with the Tag of the Cache, we'll need to flush the Cache every time we context switch across processes (which is also considered multi-threading). However, this flushing is not possible with SMT or even fine-grain (switches threads every cycle).

With a VIVT machine SMT will lead to the wrong data being used

With a VIPT and a PIPT machine the TLB must know which thread belongs to which processor

So for VIPT & PIPT, the TLB must be "thread aware". It can have an additional field in each entry that will tell us which "processor thread" the VA->PA translation belongs to. So if we use the Virtual Page # to do a TLB look up, we not only check to see if the Page # matches, but also that the Thread ID matches too.

SMT and Cache Performance

The cache is shared by all the SMT threads.

-This is good for fast data sharing between threads.

-This is bad because cache capacity is shared by all threads, which can lead to cache thrashing. which leads to more Cache Misses. This is especially harmful if the threads are not sharing much data, so they're kicking each others' data out of the cache, leading to Cache Thrashing.