Multiprocessors

Programming Massively Parallel Multiprocessors and Heterogeneous Systems (Understanding and programming the devices powering AI)

Jonathan Appavoo

Multiprocessor?

Multiprocessors

- **EASY: A computer with more than one processing unit (CPU/Core)**
- **The Promise / Hope : Make things bigger (scale up) and things get better**
  - Amdhal's Law Perspective : Reduce execution by adding more processors – get your work done faster
  - Gustafson's Law Perspective: Get more work done in the same time by adding more processors

Multiprocessors

- **HARD: How do we organize them and program them?**
  - How do we make them bigger? What **is** and what **is not** **shared**?
    - Connect computers on a network – Distributed system
      - Easy to scale
      - But very different programming model
    - Add more CPUs to an existing computer
      - Hard to scale
      - But a “familiar” programming model

Concurrency vs Parallelism?

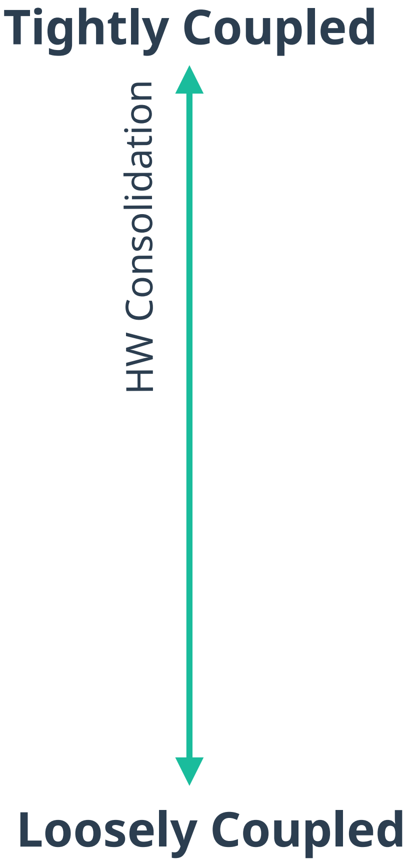
Concurrency vs Parallelism

- **Concurrency:**
  - Computing: the ability to execute more that one program or task simultaneously
- **Parallelism:**
  - Computing: the use of parallel processing in computing systems

Do multiple things at the same time to get work done faster!

All good cooks understand parallelism!

Multiprocessors: Spectrum

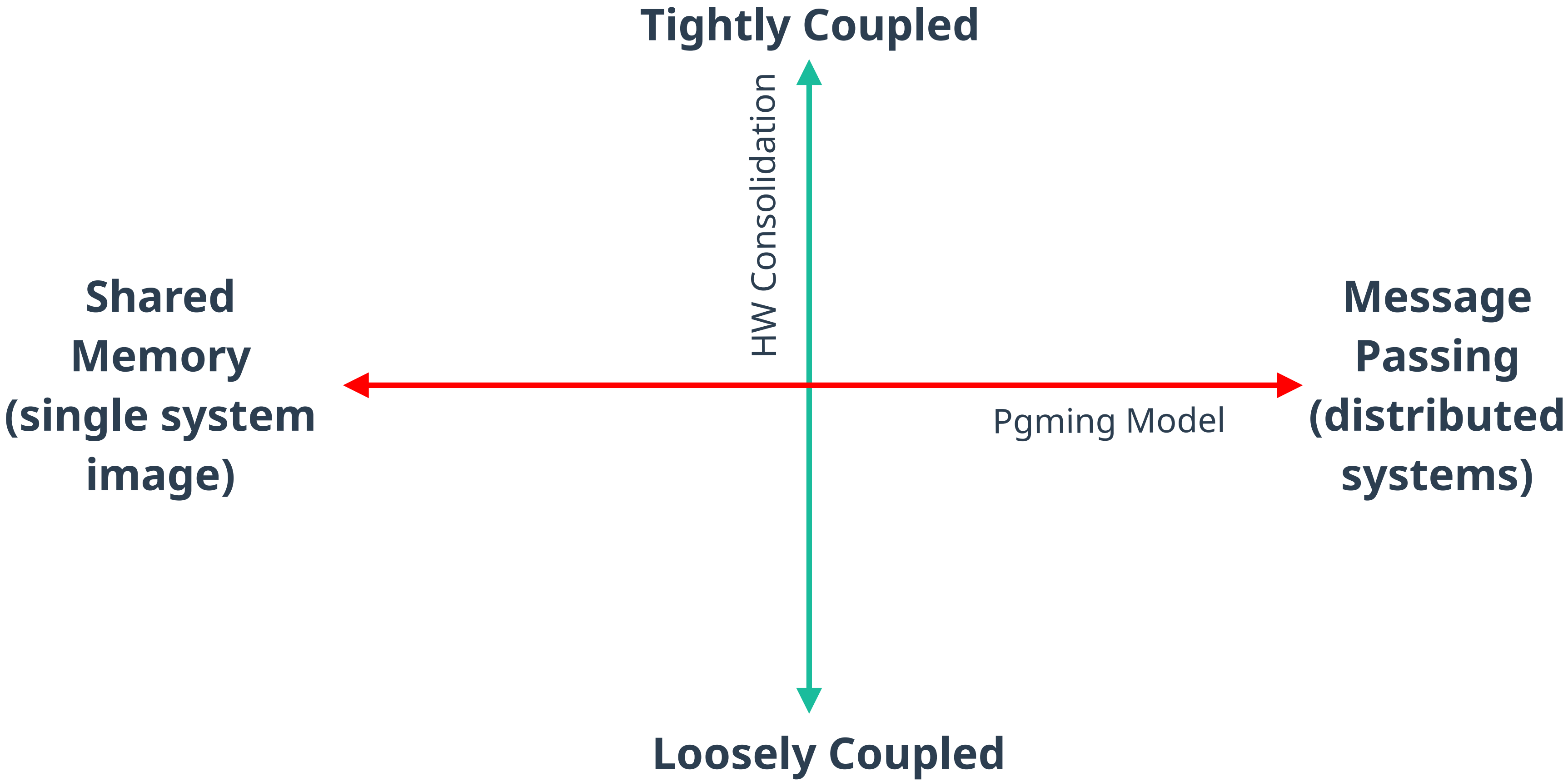


Tightly Coupled

HW Consolidation

Loosely Coupled

Multiprocessors: Spectrum



Tightly Coupled

HW Consolidation

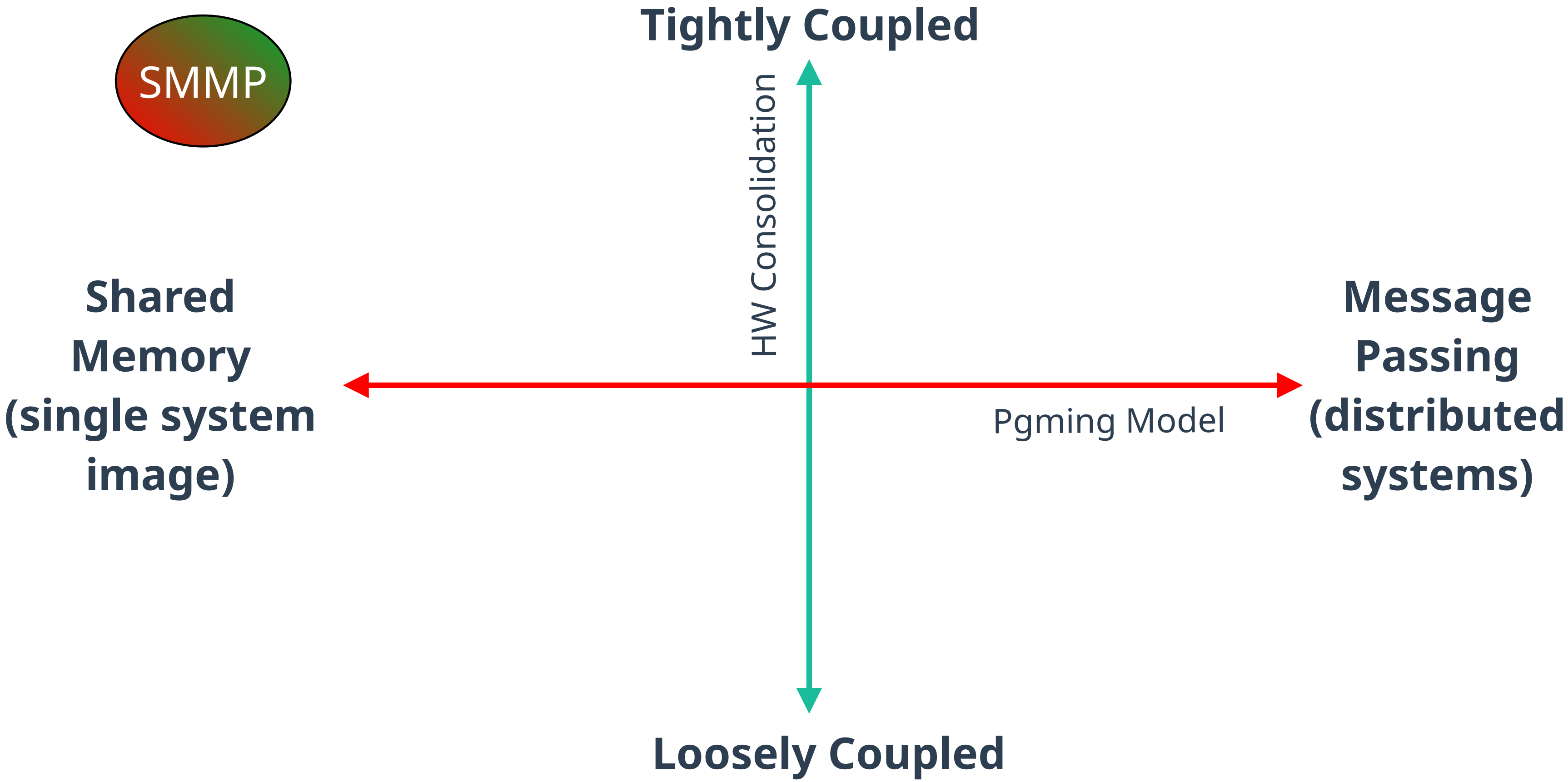
Shared Memory (single system image)

Message Passing (distributed systems)

Pgming Model

Loosely Coupled

Multiprocessors: Spectrum



SMMP

Tightly Coupled

HW Consolidation

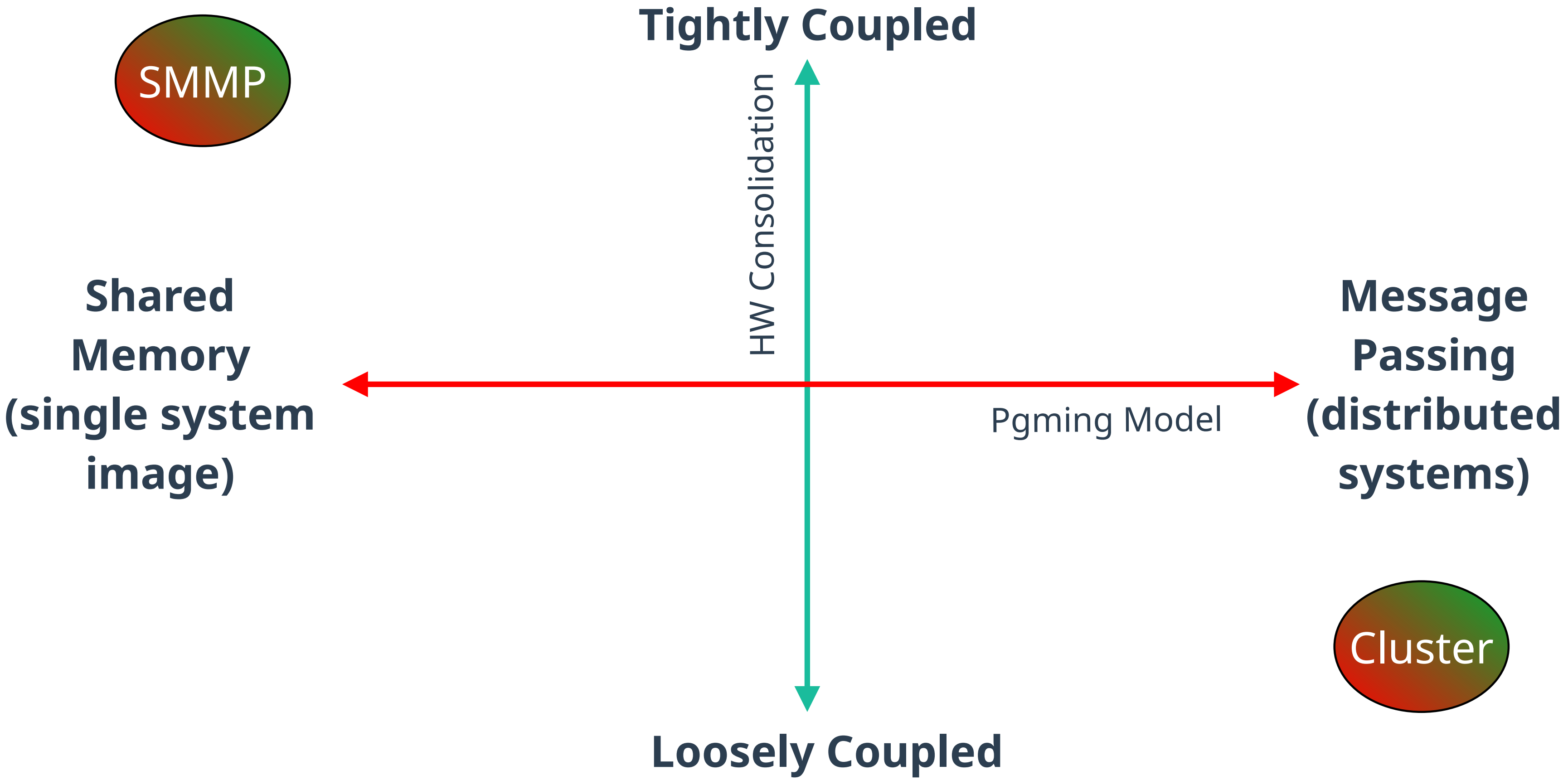
Shared Memory (single system image)

Message Passing (distributed systems)

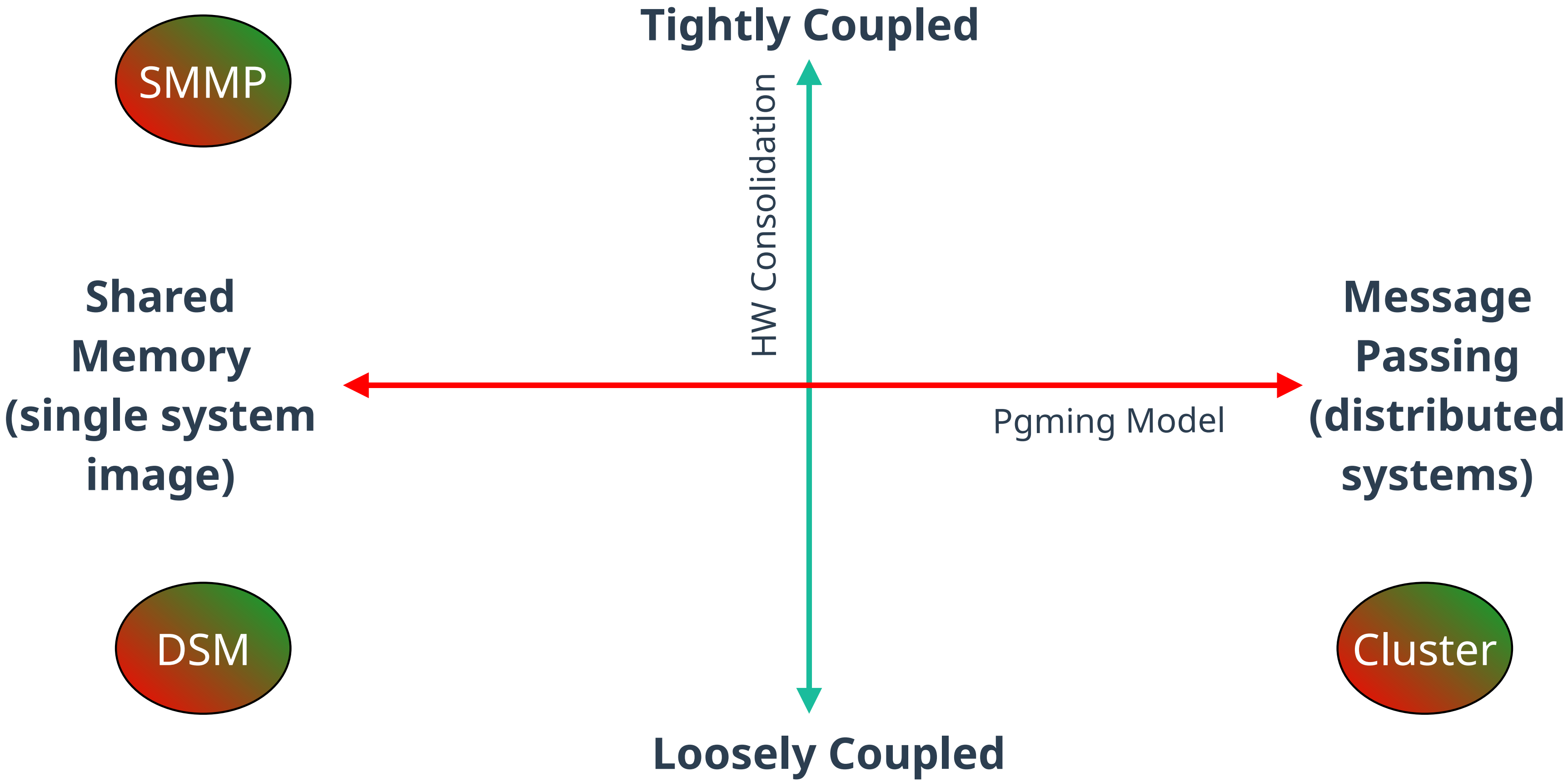
Pgming Model

Loosely Coupled

Multiprocessors: Spectrum


SMMP
Tightly Coupled
HW Consolidation
Shared Memory (single system image)
Message Passing (distributed systems)
Pgming Model
Cluster
Loosely Coupled

Multiprocessors: Spectrum



Tightly Coupled

HW Consolidation

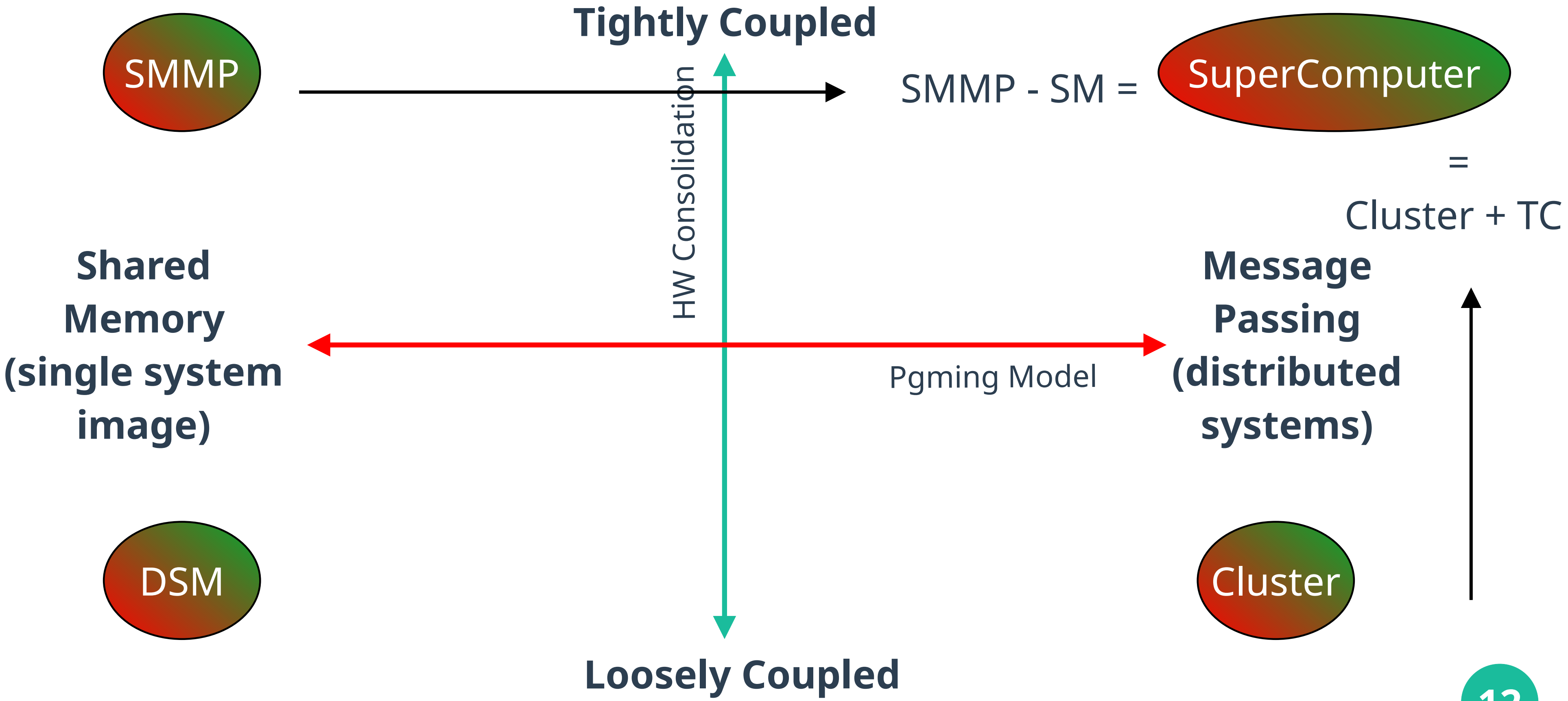
Shared Memory (single system image)

Message Passing (distributed systems)

Pgming Model

Loosely Coupled

Multiprocessors: Spectrum



Tightly Coupled

SMMP

SMMP - SM = SuperComputer

=

Cluster + TC

HW Consolidation

Shared Memory (single system image)

Message Passing (distributed systems)

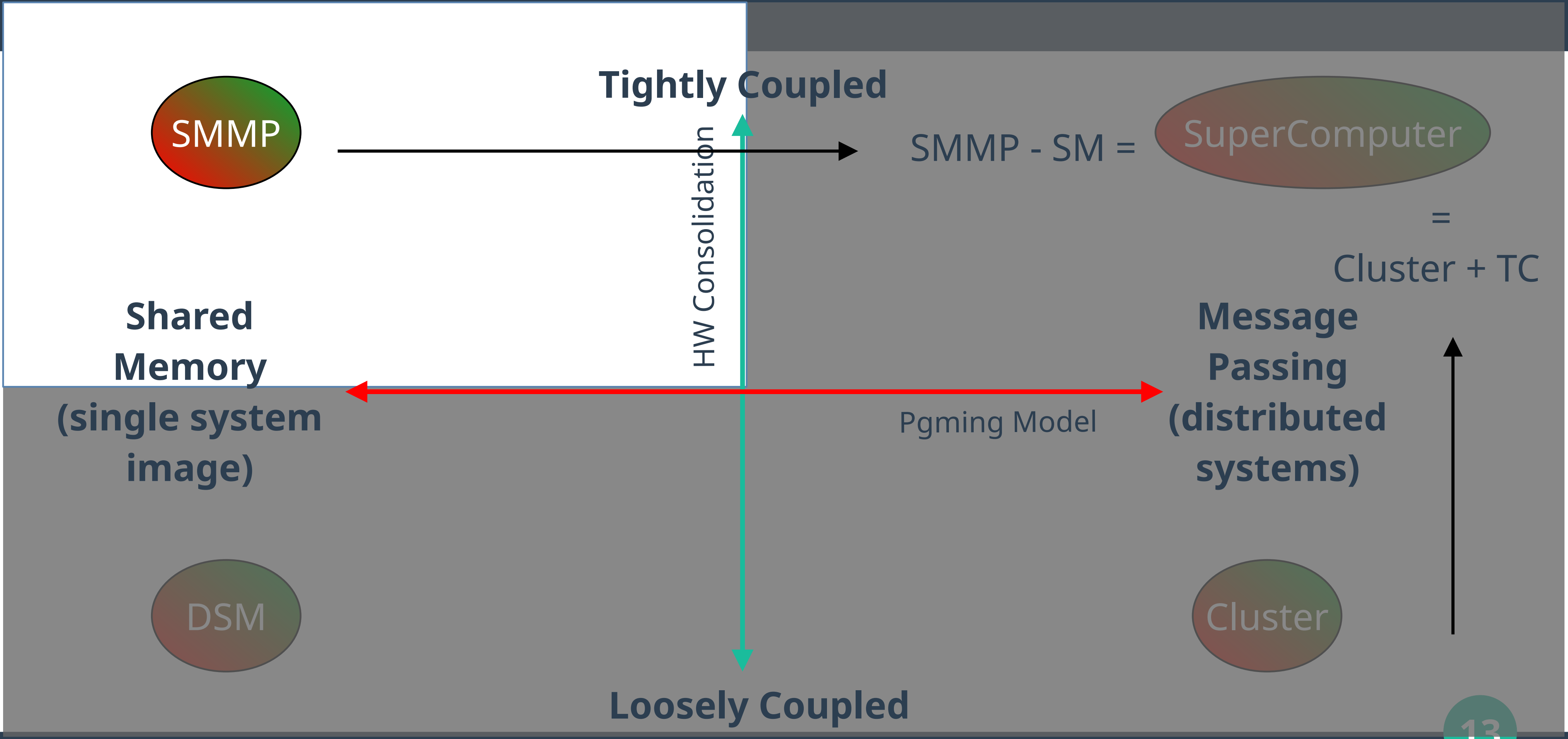
Pgming Model

DSM

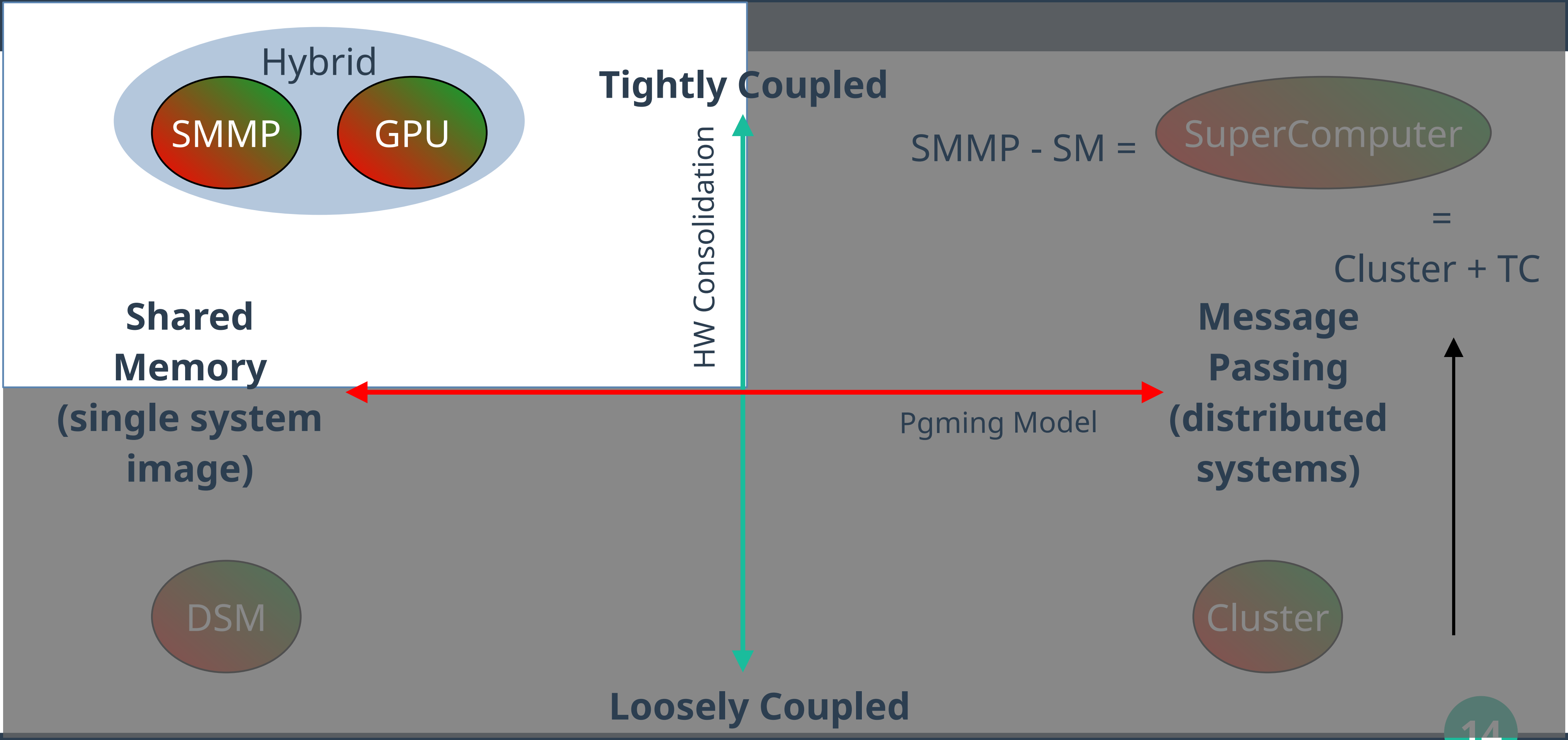
Cluster

Loosely Coupled

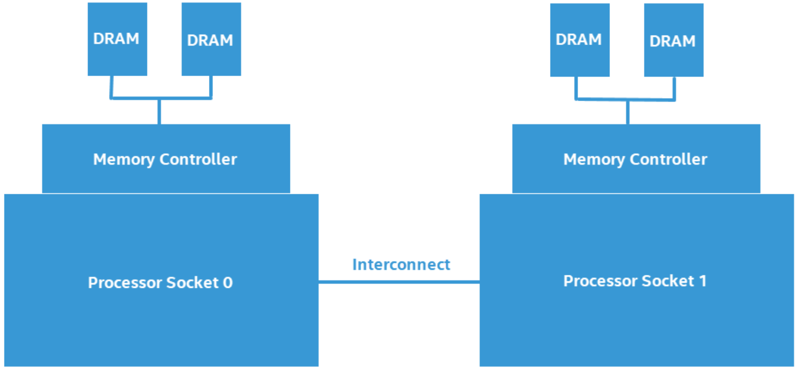
Multiprocessors: Our Focus: SMMP

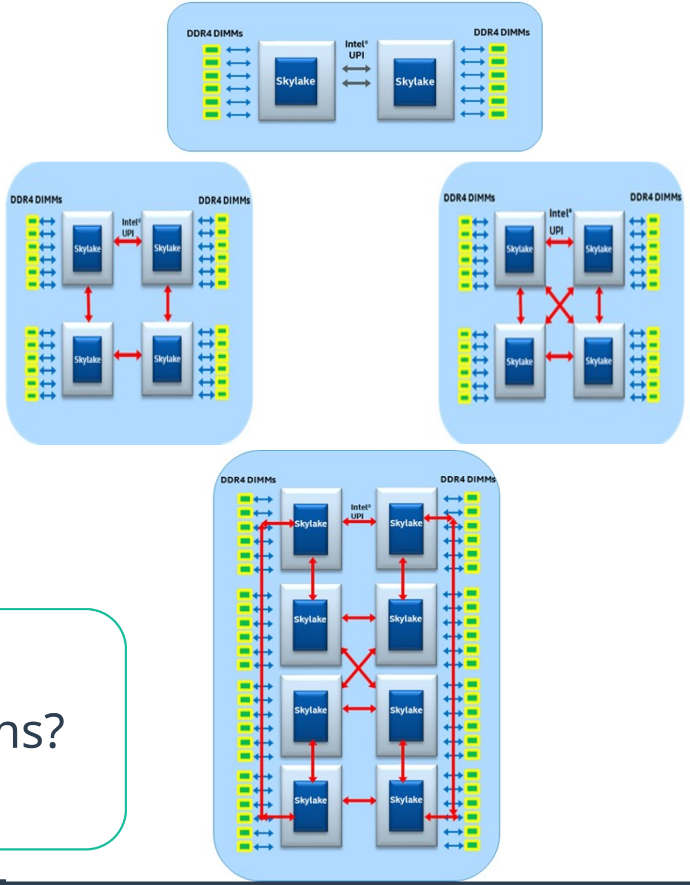


Multiprocessors: Our Focus: Hybrid SMMP**



General Purpose Shared Memory Multiprocessors (SMMPs): Our Starting point





2 Socket

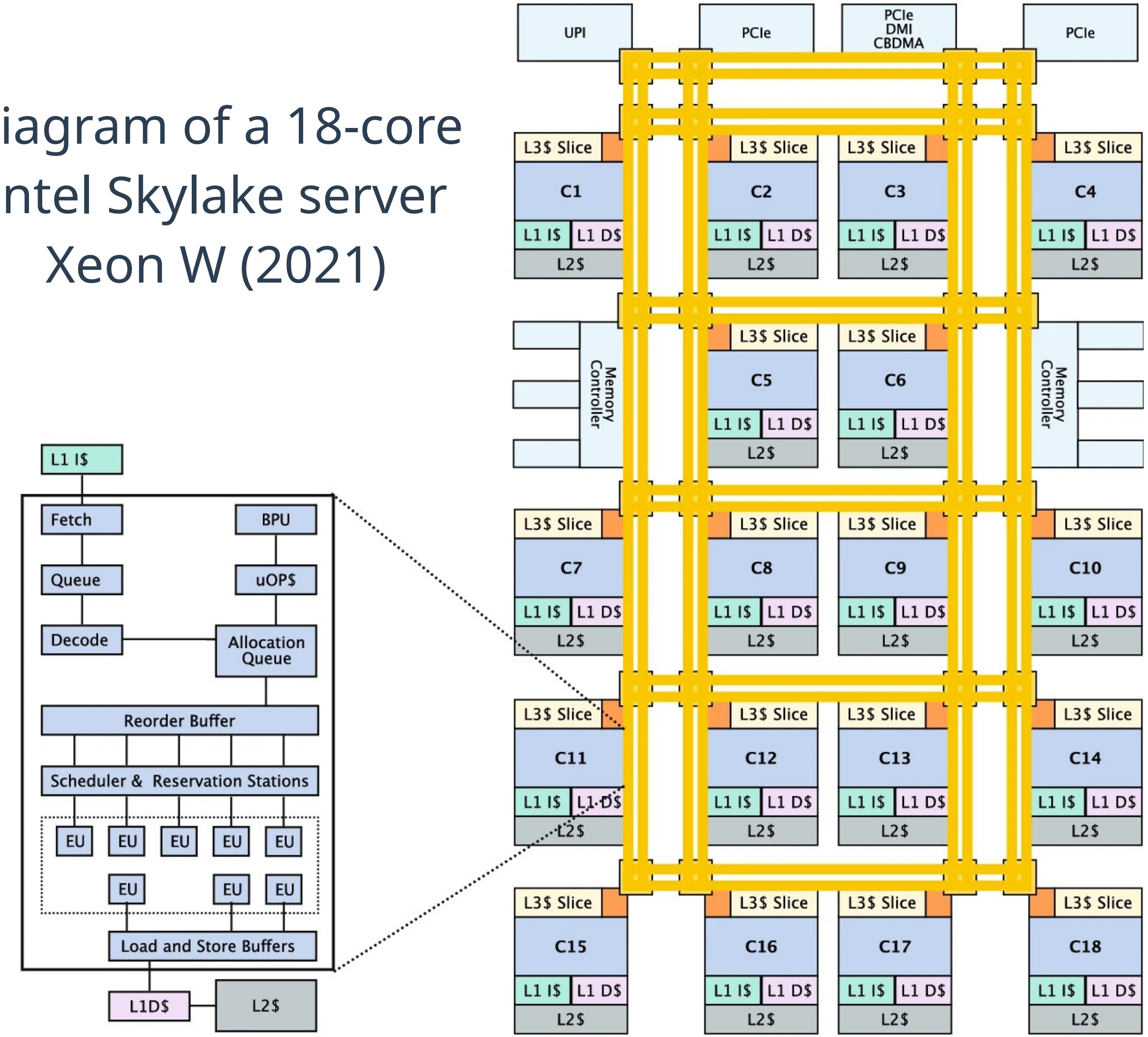
4 Socket

8 Socket

What are some of your observations?

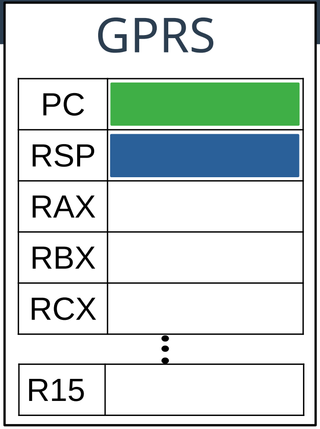
Genera
Multipr point

diagram of a 18-core Intel Skylake server Xeon W (2021)



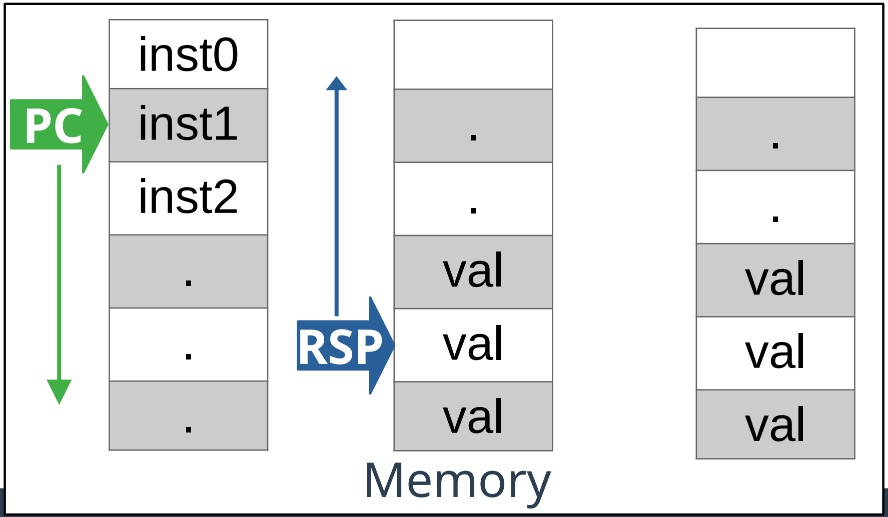
https://link.springer.com/article/10.1007/s11227-021-03985-0/figures/4

Parallelism : Some intuition



GPRS

PC	🟩
RSP	🟦
RAX	
RBX	
RCX	
⋮	
R15	



```
void
vecAdd(long *a,long *b,long *c,int dim)
{
  for (int i=0;i<dim;i++){
    c[i] = a[i] + b[i];
  }
}
```

- Some Obvious approaches
- and some not so obvious

Parallelism : Obvious 1: Multi-threading

HW Thread 0

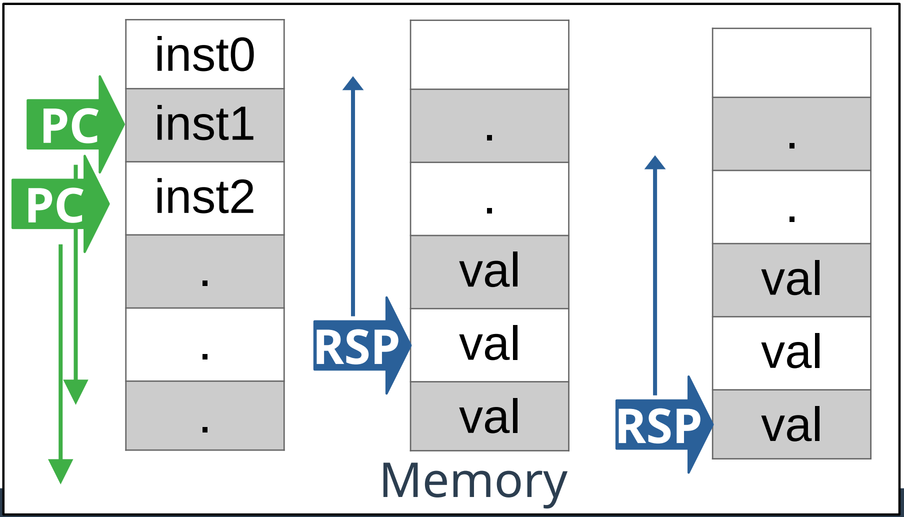
GPRS

PC	🟩
RSP	🟦
RAX	
RBX	
RCX	
⋮	
R15	

HW Thread 1

GPRS

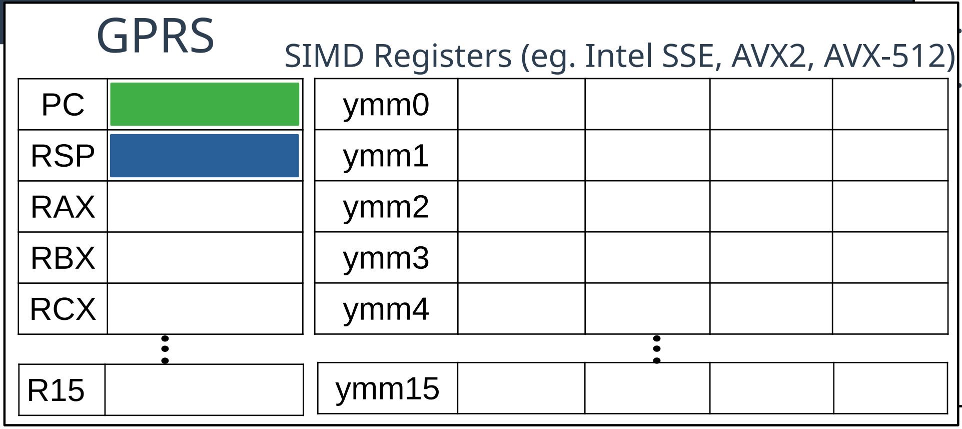
PC	🟩
RSP	🟦
RAX	
RBX	
RCX	
⋮	
R15	



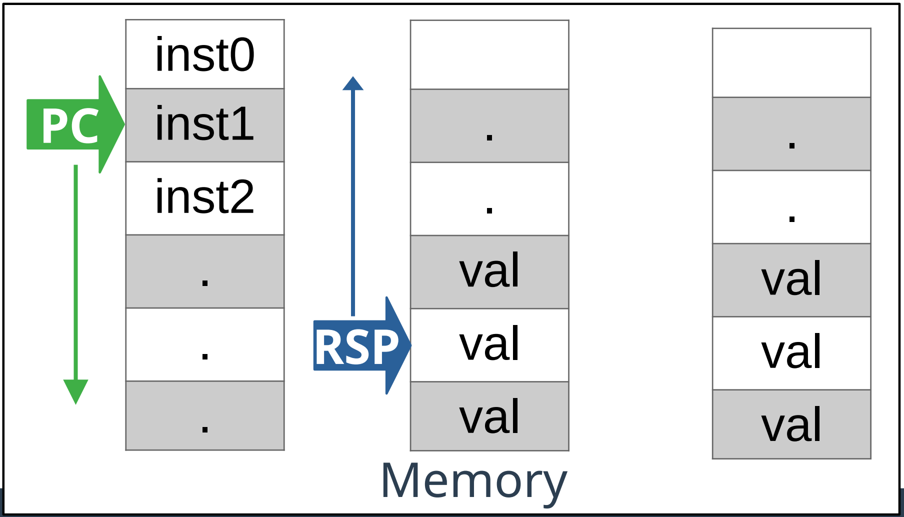
```
void
vecAdd(long *a,long *b,long *c,int dim)
{
  for (int i=0;i<dim;i++){
    c[i] = a[i] + b[i];
  }
}
```

- **HW: Duplicate GPRS and execution loop**
- **SW: Modify to support and use “threads”**
  - eg. split work across two thread each doing DIM/2 additions
  - OS/Libraries organize memory so that each thread has its own stack and provide ways to create and destroy
- **SW threads do not have to have a HW thread but thread parallelism only if there are at least 2 HW Threads**

Parallelism : Obvious 2: Vectorize HW (SIMD)



```
oid
ecAdd(long *a,long *b,long *c,int dim)

 for (int i=0;i<dim;i++){
   c[i] = a[i] + b[i];
 }
```



- **HW: Add a ncew type of “wide” register that can hold multiple values and associated instructions for loading, storing and operating on the new registers**
  - New instructions apply operation in parallel across all values in the registers: eg. VPADDQ ymm1, ymm2/mem
- **SW: Modify to support and use new SIMD types**
  - Don’t have to explicitly write parallel code but must use new types and instructions:
    - Code must be written to specific widths supported by hardware
    - eg. load 4 longs, add 4 longs and store 4 longs

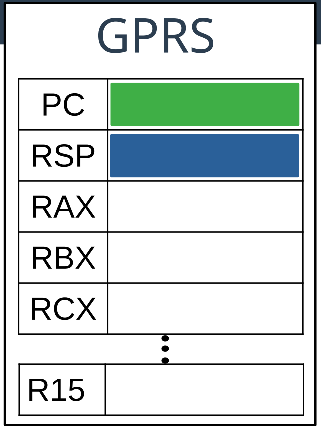
Parallelisms: Multi-threading and SIMD

Both required HW changes and SW changes. While SIMD seems less intrusive it is also less general and requires you to carefully know what you’re hardware supports and “map” your data to it! If a new vector width is added you need to at least recompile if not rewrite.

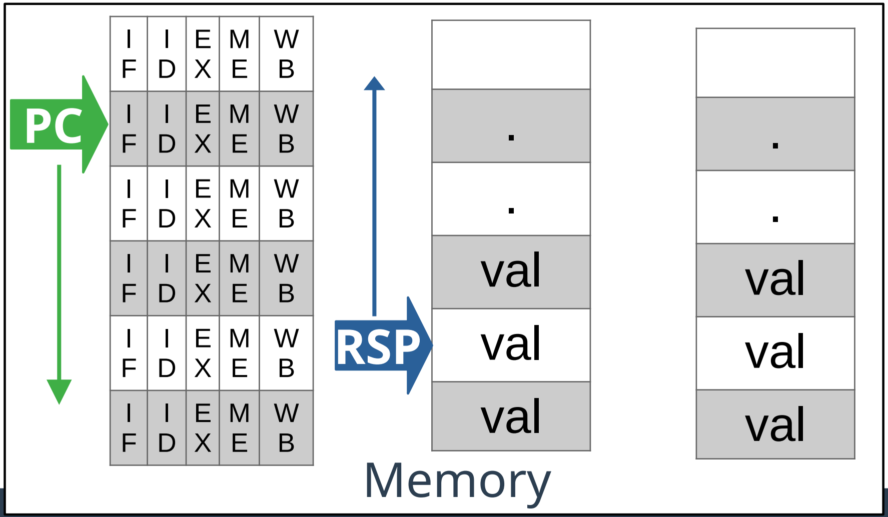
Not Obvious : Instruction Level Parallelism (ILP)

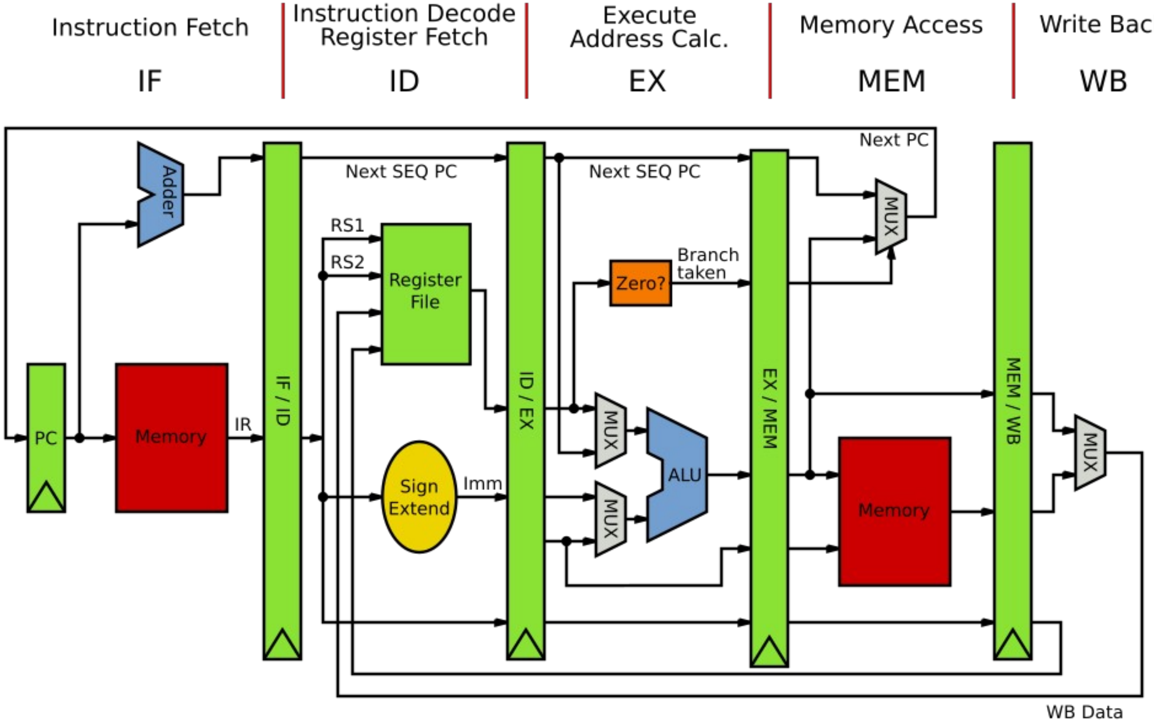
- **Can we modify HW without modifying the SW**
  - Instruction Level Parallelism (typical ideas used)
    - Idea 1: Pipelining : overlapped instruction execution
    - Idea 2: Superscaler: multiple instruction execution
    - Idea 3: Out-of-Order execution
    - Idea 4: Speculative execution: a head of time execution

Instruction Level Parallelism: “execution”



- **HW breaks instruction “Exeuction” into stages**
- **And allows, *when possible*, stages of instructions to overlap**

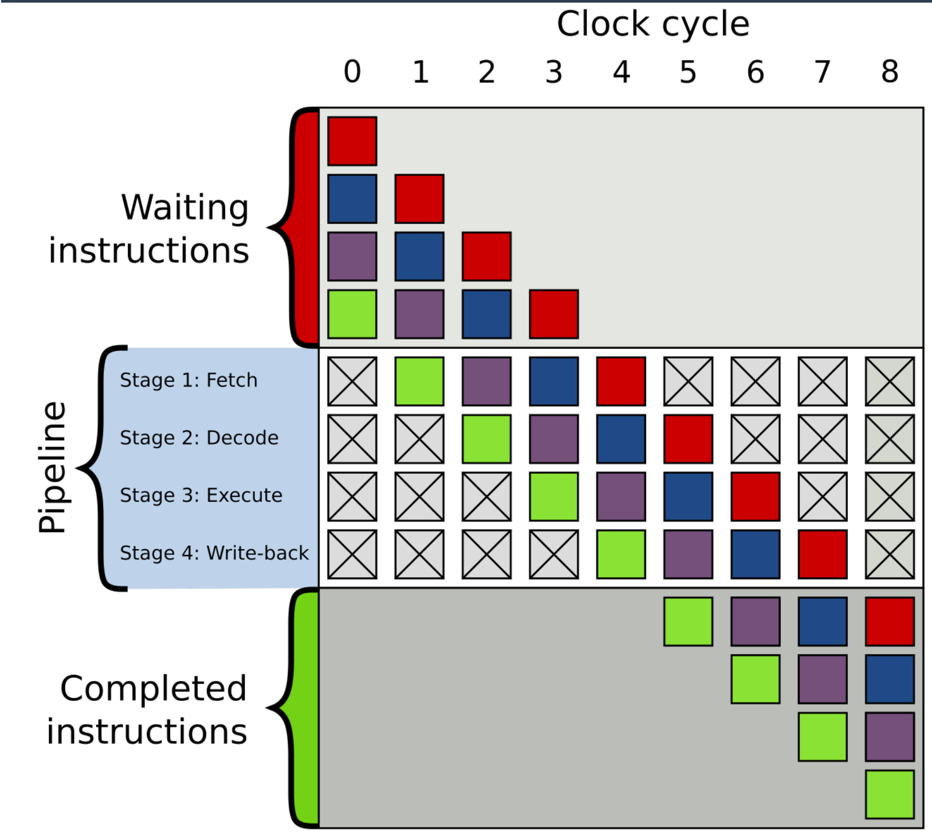




https://en.wikibooks.org/wiki/Microprocessor_Design/Pipelined_Processors

Instruction Level Parallelism: Pipelines

Cycle	Stage
0	Four instructions are waiting to be executed
1	Green fetch
2	Green decode, Purple fetch
3	Green execute, Purple decode, Blue fetch
4	Green write-back, Purple execute, Blue decode, Red fetch
5	Green completed, Purple write-back, Blue execute, Red decode
6	Purple completed, Blue write-back, Red execute
7	Blue completed, Red write-back
8	Red completed



https://en.wikipedia.org/wiki/Instruction_pipelining

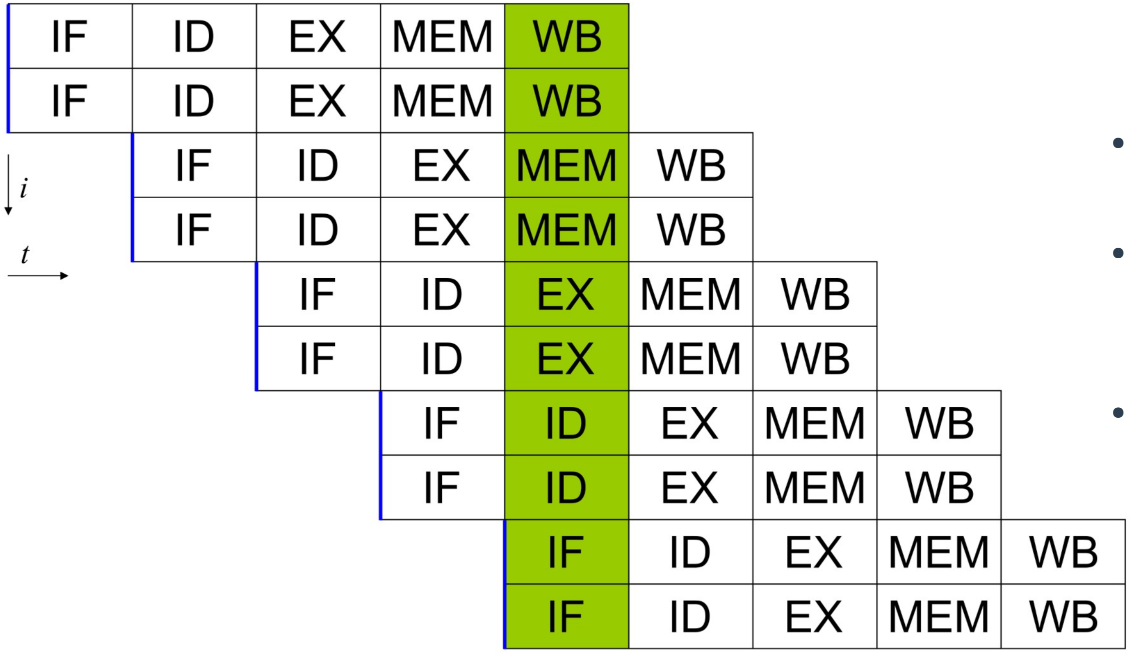
ILP: pipeline depth and hazards

- **Depth: Number of Stages – more stages more overlap**
- **Pipeline Hazards → causes stalls and bubble : pipeline stage must wait (cycle is empty)**
  - Read after Write (RAR) (next instruction needs data not yet written by a previous instruction) – ADD R1, R2, R3; SUB R4, R1, R5
  - Structural hazard – not enough resources (eg. three memory operations needed but only two “ports” to memory)
  - Control hazard – Branching : can’t determine which instruction comes next due to branch/jump
- **Some instruction orders are better than others**
  - Compiler and programmer can improve performance by choosing instructions and orders that improves pipeline performance
    - Organize instructions to maximize independence
    - Reduce branches – straighten code – eg. unroll loops
- **Many HW ILP techniques work to reduce hazards**

https://commons.wikimedia.org/wiki/File:Pipeline,_4_stage.svg#/media/File:Pipeline,_4_stage.svg

ILP: Super Scalar

- **Width: Number of pipelines – Super Scalar (multi-issue)**



IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

- **Not SIMD (vector processing) but can execute more than one instruction at a time**
- **Can fetch and execute more than one instruction at a time**
- **eg. more than one pipeline**
  - Eg. fetch two instructions, decode two, execute two…
- **Again only works if instructions do not “conflict” for resources**
  - Or depend on each other
  - Compiler and programmer can write code that better exploits supers-scalar design

By Amit6, original version (File:Superscalarpipeline.png) by User:Poil - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=9748747

ILP: Out-of-Order Execution

- **HW allowed to execute instruction Out of the the order they appear in the program!**
  - Help to reduce the impact of hazards due to in-order execution
  - HW can move on to another instruction while prior instructions are waiting (stalled due to data not ready)
  - Later instructions, in program order, can complete before prior ones!
  - HW is responsible for figuring out when it is “safe” and how to avoid “clobbering” registers
- **More like scheduling – HW has a scheduler that executes instructions as they become ready.**
  - However, processor must do it in a way that avoids “problems”
- **Note out-of-order memory operations can happen even without Out-of-Order execution:**
  - caches already introduce this problem.

ILP: Speculative Execution

- **Taking a walk on the wild side to avoid hazards and Increase ILP**
  - Predict branch and start executing before we know for sure if our guess was right
  - Allows us to fill the pipeline even if a branch is encountered
- **How do we predict?**
- **What do we do if we are wrong?**

Parallelism in the modern age

- **Most modern processors include a mixture of all these techniques**
  - Multi-cores: Several HW threads on a single chip
    - Each core implements various ILP techiques
  - Multi-socket: Several multi-core chip
- **Large Caches to try and help hide memory latency**
  - Can’t execute instructions fast when memory is slow
    - Caches used to mitigate latency

Parallelism: Simultaneous Multi-threading/Hyper-threading

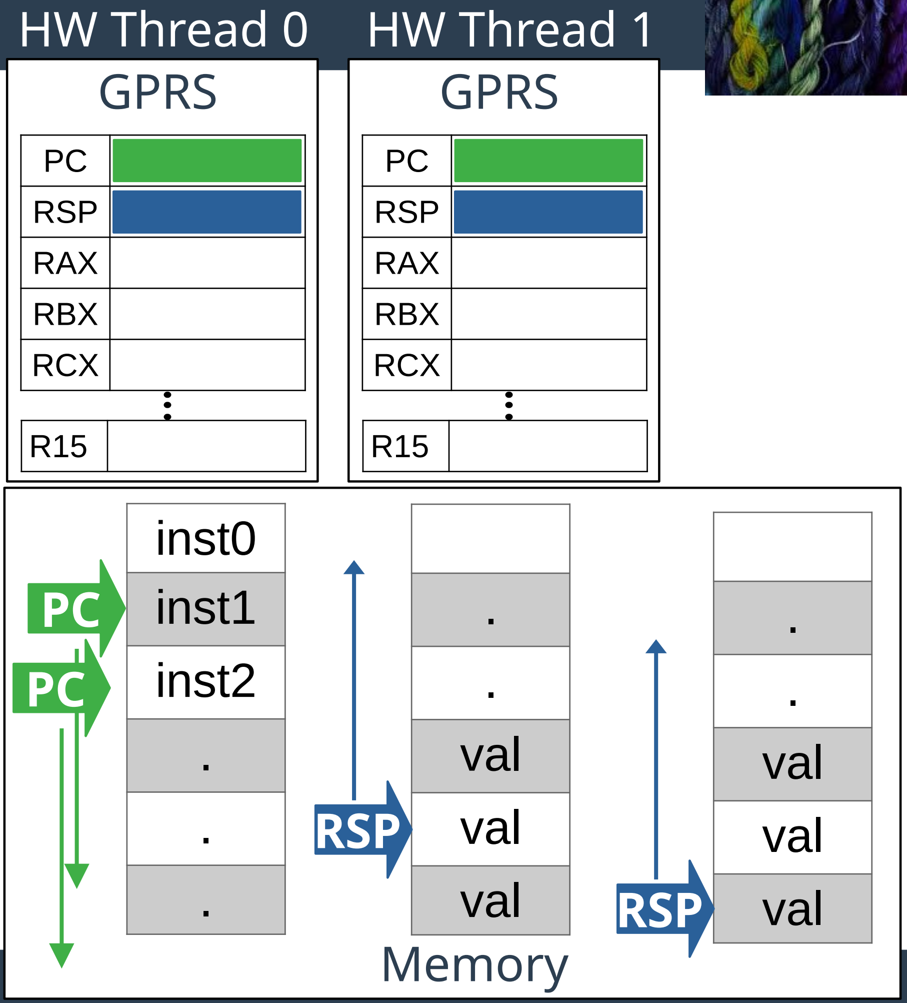
- **Like super-scalar but core/processor can issue multiple instructions from more than one thread (stream of instructions)**
- **Each “Hyper-thread” has its own registers but**
- **Shares resources normally independent for multi-core hw threads**
  - Shared L1 Cache
  - Typically shared page-table (Virtual Address Space aka the same process)
- **Idea is that that hardware will be able to execute hyper-threads to hide memory latency more easily as the instructions come from independently programmed, but related threads (stalling one and continuing with the other)**
- **JA RANT**

What is a core (of a General Purpose processor)?

- **HW Unit for “Task” level parallelism**
  - Supports one more HW threads
    - Has a set registers
    - Has a collection of “Execution Units” that can be used as stages of an instruction pipeline
  - Each can execute threads from different or the same process
    - Two cores allows two
      - independent programs to run in parallel
      - Or at least two threads of the same program to run in parallel

Explicit Multi-processing with Threads: Thread -level concurrency

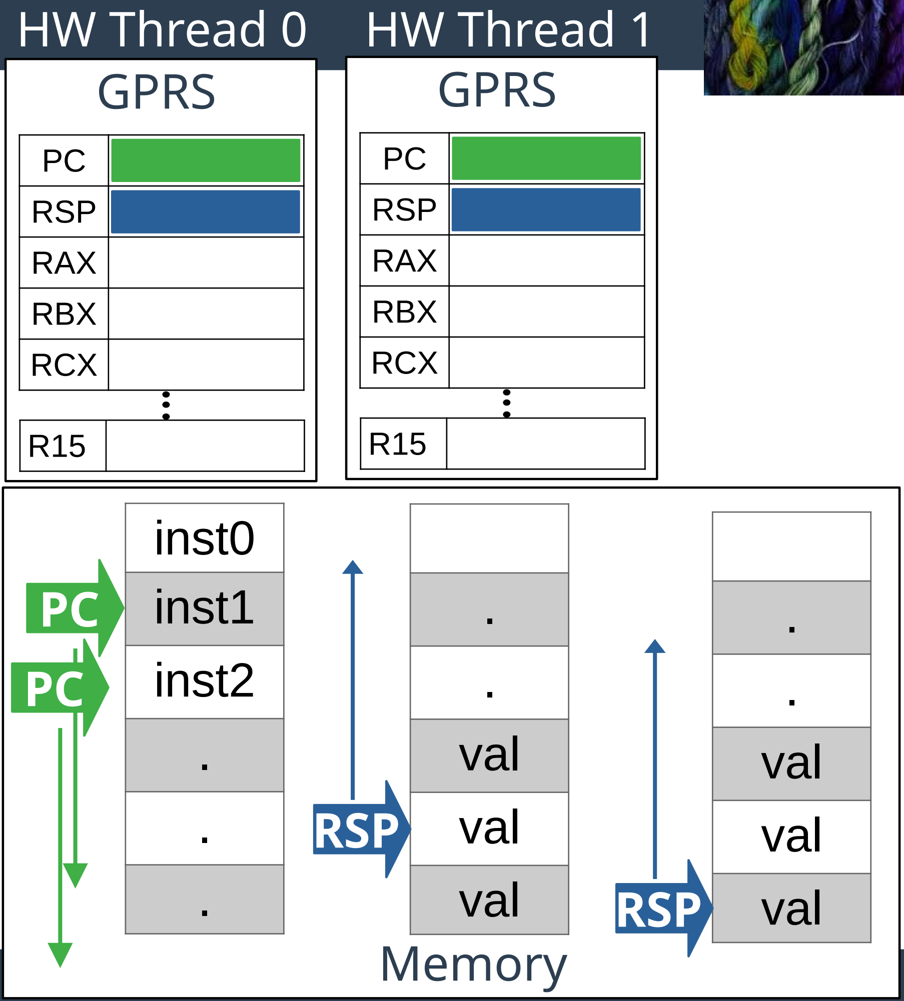
- **HW**
  - Independent set of General Purpose Registers (GPRS)
  - Can execute a sequential stream of instructions (albeit potentially using ILP techniques)
  - Typically has lots of caches to improve / hide latency of memory accesses (L3, L2, L1)
  - Each executes **independently**



Explicit Multi-processing with Threads: Thread -level concurrency

- **SW**
  - Software ensures that each HW thread is configured “correctly”
    - PC can be initialized independently (but you are free to use same starting point)
    - Each has their own independent stacks. (**why is this important?**)
  - All threads in a process share memory but NOT registers!
  - POSIX threads are an example of an Application Programmer Interface (API) for threads
  - OS / Libraries map

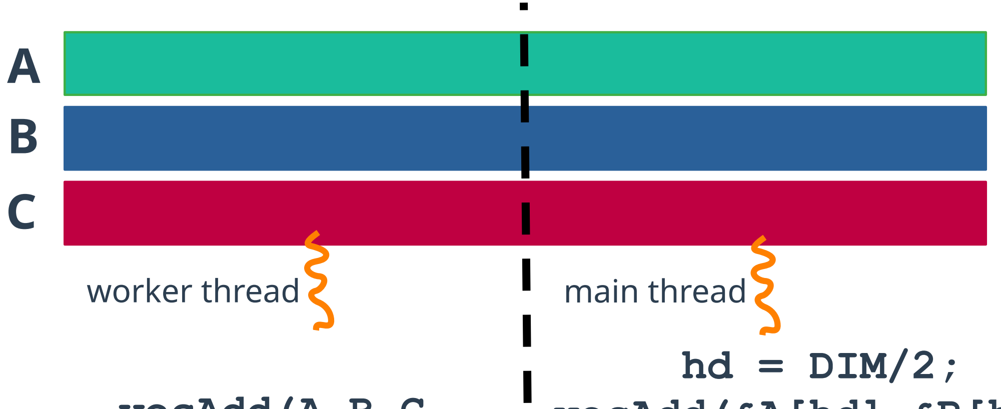
    SW thread → HW threads



Explicit Multi-processing with Threads: Shared memory: good, bad and the ugly

- **All threads can access all instructions/code**
- **Can pass pointers around**
- **Any thread can access an data structure:**
  - Share simple variables, arrays, lists, queues, trees, etc!
- **Makes things feel familiar**
  - No need for message passing
  - Explicit communication
  - No need for Remote Procedure Calls (RPC)

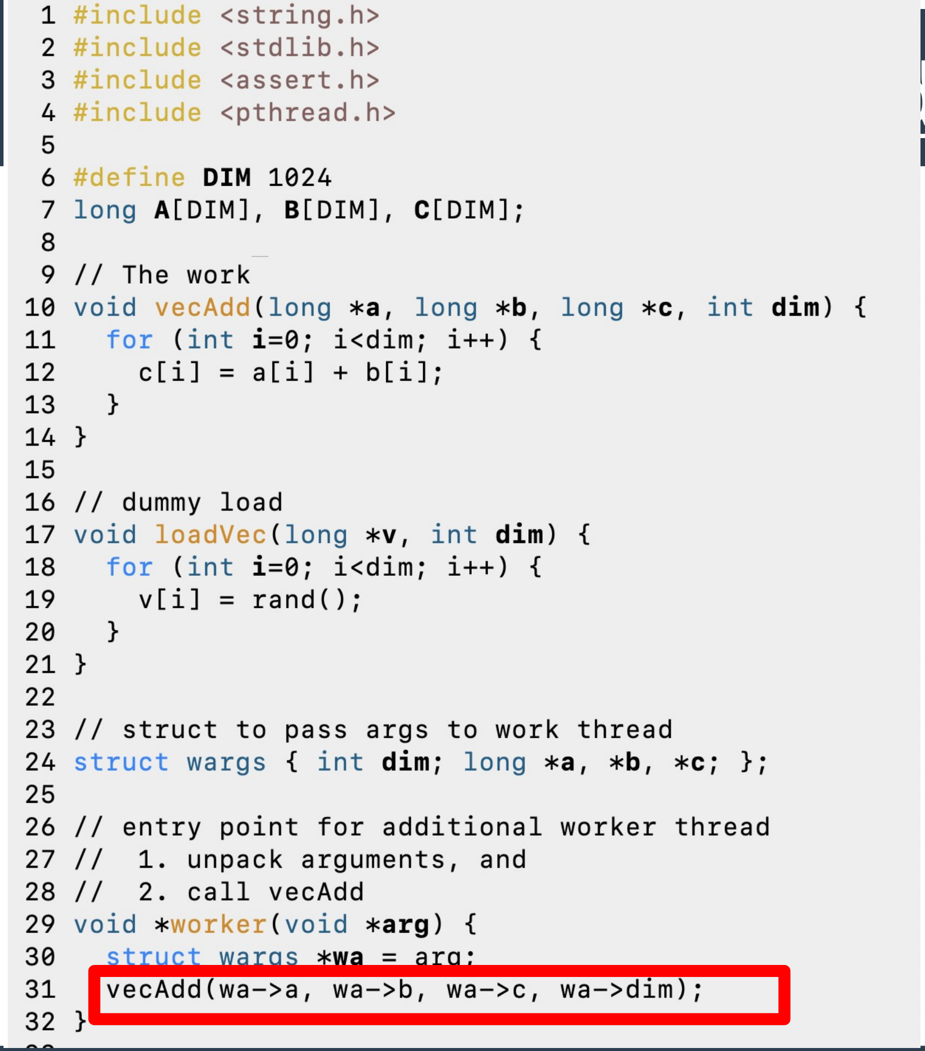
```
void
vecAdd(long *a,long *b,long *c,int dim)
{
   for (int i=0;i<dim;i++){
      c[i] = a[i] + b[i];
   }
}
```



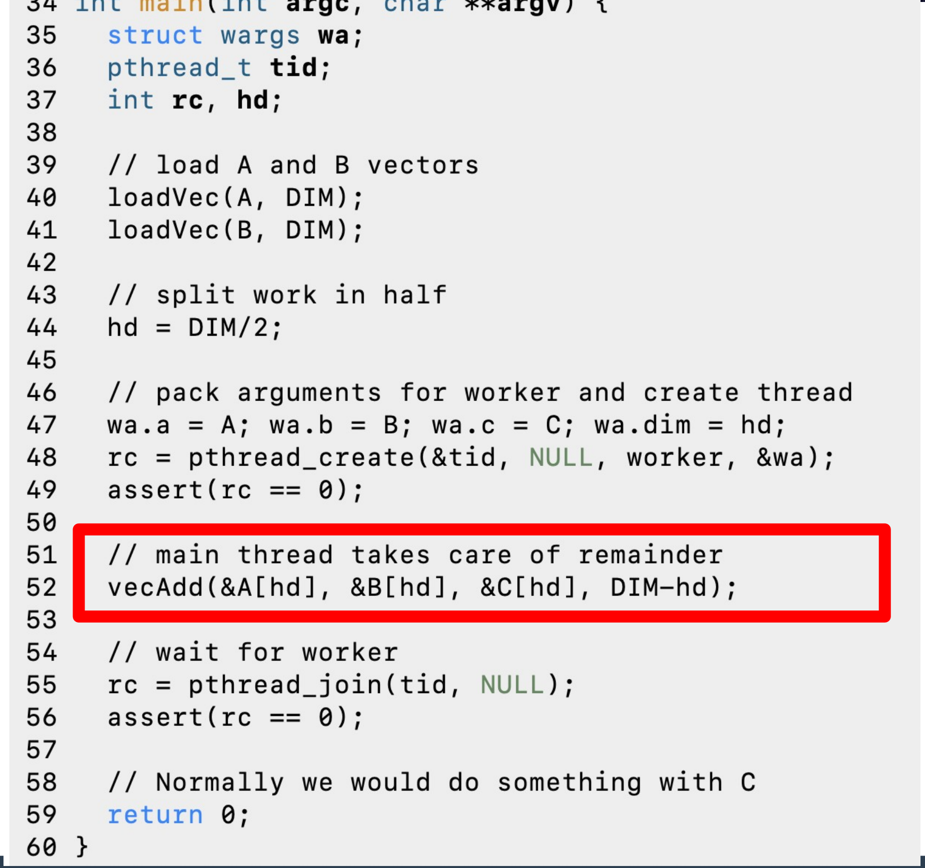
```
vecAdd(A,B,C,
   DIM/2)
```

```
hd = DIM/2;
vecAdd(&A[hd],&B[hd],
   &C[hd],DIM-hd);
```

Explicit Multi-processing with Threads:
d, bad and the ugly

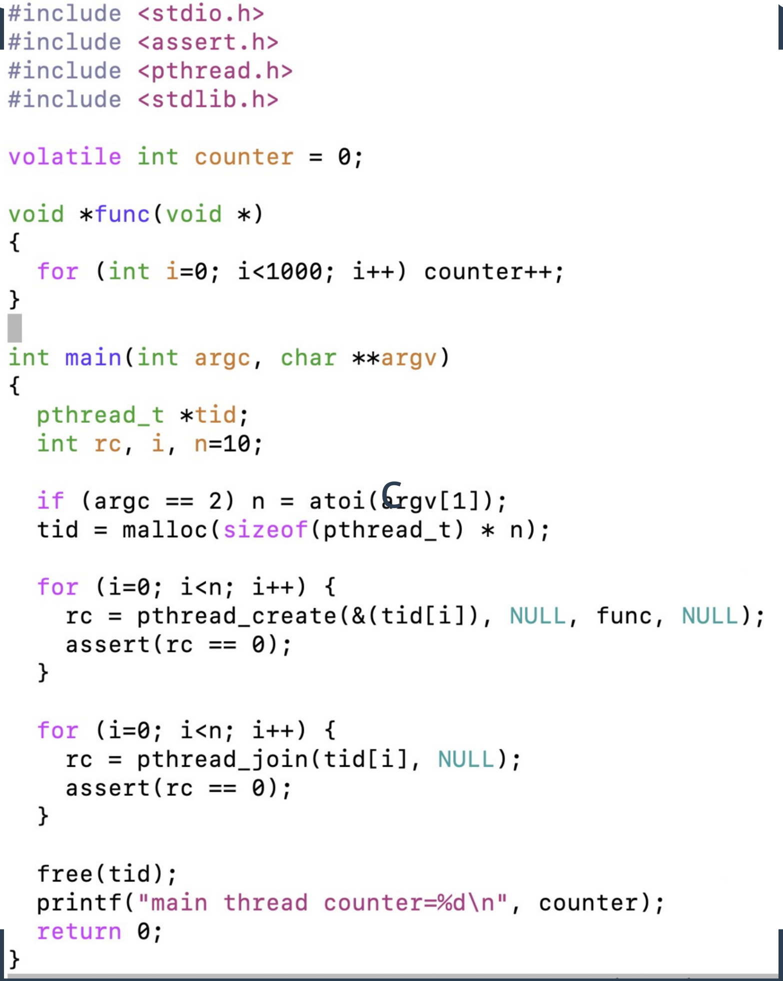


```c
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <pthread.h>

#define DIM 1024
long A[DIM], B[DIM], C[DIM];

// The work
void vecAdd(long *a, long *b, long *c, int dim) {
  for (int i=0; i<dim; i++) {
    c[i] = a[i] + b[i];
  }
}

// dummy load
void loadVec(long *v, int dim) {
  for (int i=0; i<dim; i++) {
    v[i] = rand();
  }
}

// struct to pass args to work thread
struct wargs { int dim; long *a, *b, *c; };

// entry point for additional worker thread
//   1. unpack arguments, and
//   2. call vecAdd
void *worker(void *arg) {
  struct wargs *wa = arg;
  vecAdd(wa->a, wa->b, wa->c, wa->dim);
}
```

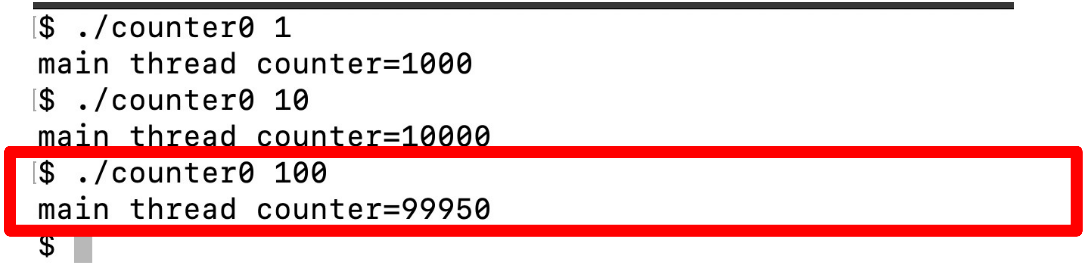


```c
int main(int argc, char **argv) {
  struct wargs wa;
  pthread_t tid;
  int rc, hd;

  // load A and B vectors
  loadVec(A, DIM);
  loadVec(B, DIM);

  // split work in half
  hd = DIM/2;

  // pack arguments for worker and create thread
  wa.a = A; wa.b = B; wa.c = C; wa.dim = hd;
  rc = pthread_create(&tid, NULL, worker, &wa);
  assert(rc == 0);

  // main thread takes care of remainder
  vecAdd(&A[hd], &B[hd], &C[hd], DIM-hd);

  // wait for worker
  rc = pthread_join(tid, NULL);
  assert(rc == 0);

  // Normally we would do something with C
  return 0;
}
```

Explicit Multi-processing with Threads: Shared memory: good, bad and the ugly

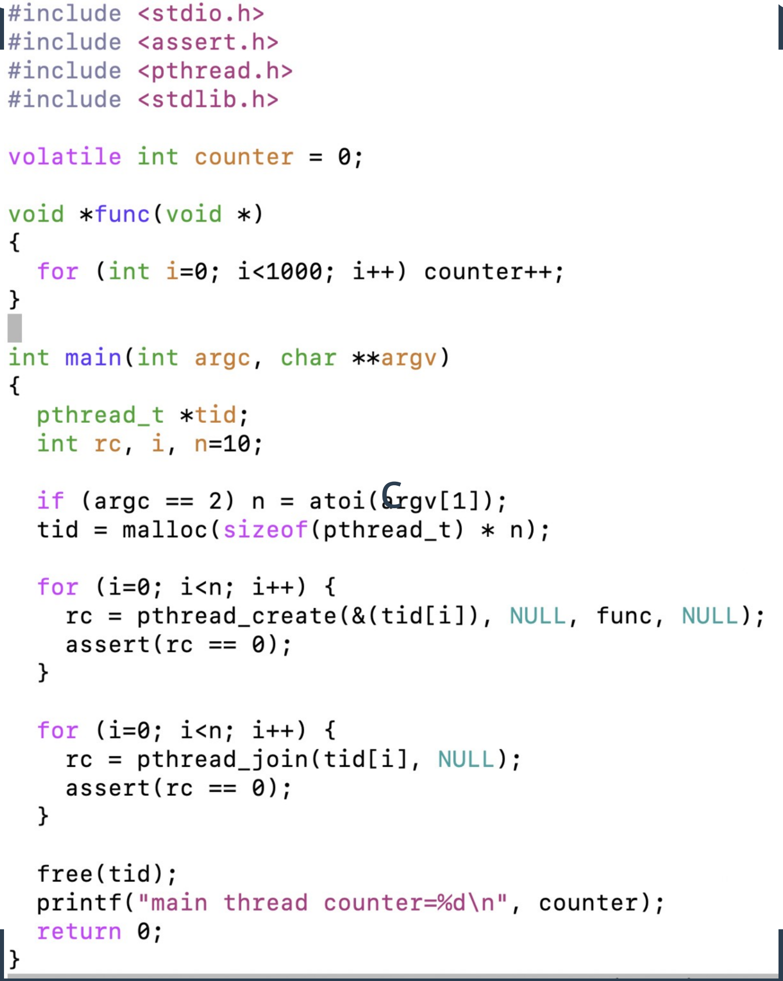


```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <stdlib.h>

volatile int counter = 0;

void *func(void *)
{
  for (int i=0; i<1000; i++) counter++;
}

int main(int argc, char **argv)
{
  pthread_t *tid;
  int rc, i, n=10;

  if (argc == 2) n = atoi(argv[1]);
  tid = malloc(sizeof(pthread_t) * n);

  for (i=0; i<n; i++) {
    rc = pthread_create(&(tid[i]), NULL, func, NULL);
    assert(rc == 0);
  }

  for (i=0; i<n; i++) {
    rc = pthread_join(tid[i], NULL);
    assert(rc == 0);
  }

  free(tid);
  printf("main thread counter=%d\n", counter);
  return 0;
}
```

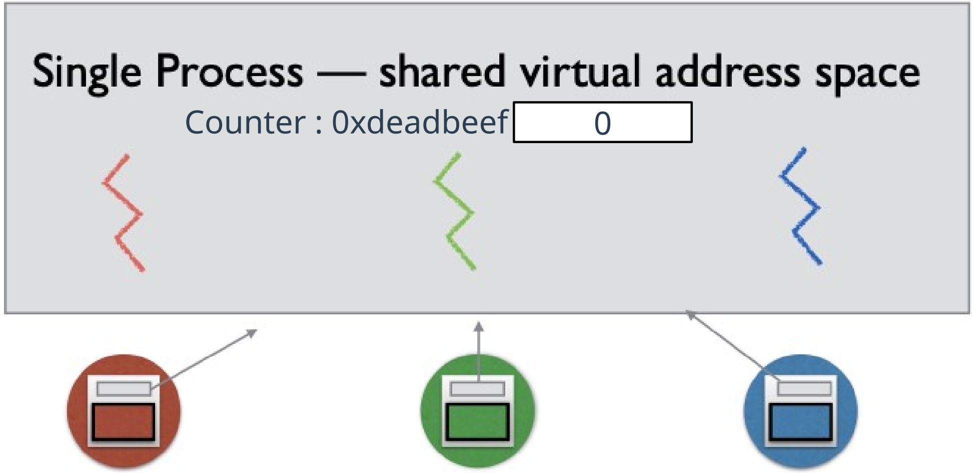


```
$ ./counter0 1
main thread counter=1000
$ ./counter0 10
main thread counter=10000
$ ./counter0 100
main thread counter=99950
$
```

Explicit Multi-processing with Threads: Shared memory: good, bad and the ugly

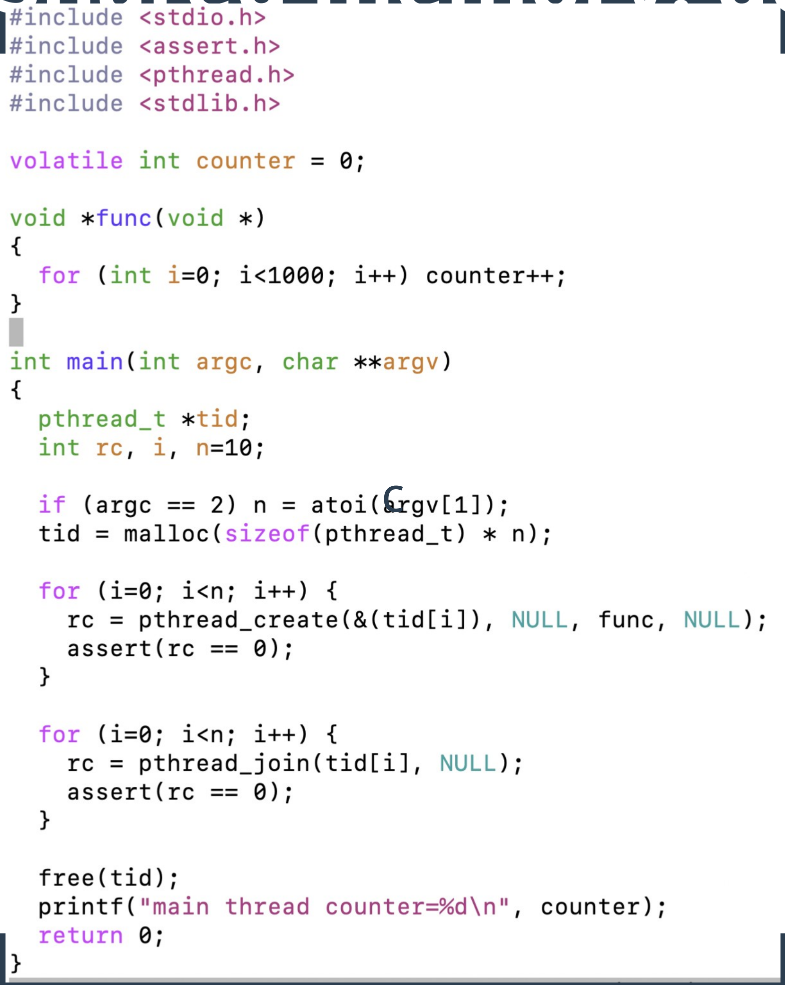


```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <stdlib.h>

volatile int counter = 0;

void *func(void *)
{
  for (int i=0; i<1000; i++) counter++;
}

int main(int argc, char **argv)
{
  pthread_t *tid;
  int rc, i, n=10;

  if (argc == 2) n = atoi(argv[1]);
  tid = malloc(sizeof(pthread_t) * n);

  for (i=0; i<n; i++) {
    rc = pthread_create(&(tid[i]), NULL, func, NULL);
    assert(rc == 0);
  }

  for (i=0; i<n; i++) {
    rc = pthread_join(tid[i], NULL);
    assert(rc == 0);
  }

  free(tid);
  printf("main thread counter=%d\n", counter);
  return 0;
}
```



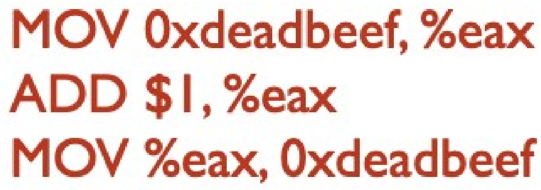
Physical Cores/Processors : common page table pointer, but GPRS can be arbitrary eg. PC is different

Explicit Multi-processing with Threads: Shared memory: good, bad and the ugly

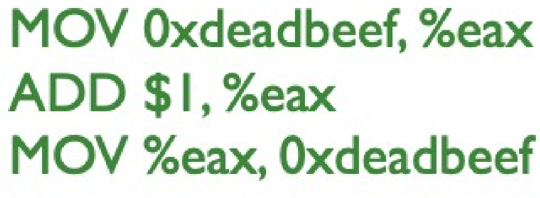


```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <stdlib.h>

volatile int counter = 0;

void *func(void *)
{
  for (int i=0; i<1000; i++) counter++;
}

int main(int argc, char **argv)
{
  pthread_t *tid;
  int rc, i, n=10;

  if (argc == 2) n = atoi(argv[1]);
  tid = malloc(sizeof(pthread_t) * n);

  for (i=0; i<n; i++) {
    rc = pthread_create(&(tid[i]), NULL, func, NULL);
    assert(rc == 0);
  }

  for (i=0; i<n; i++) {
    rc = pthread_join(tid[i], NULL);
    assert(rc == 0);
  }

  free(tid);
  printf("main thread counter=%d\n", counter);
  return 0;
}
```

Basic problem: updating shared data structures concurrently



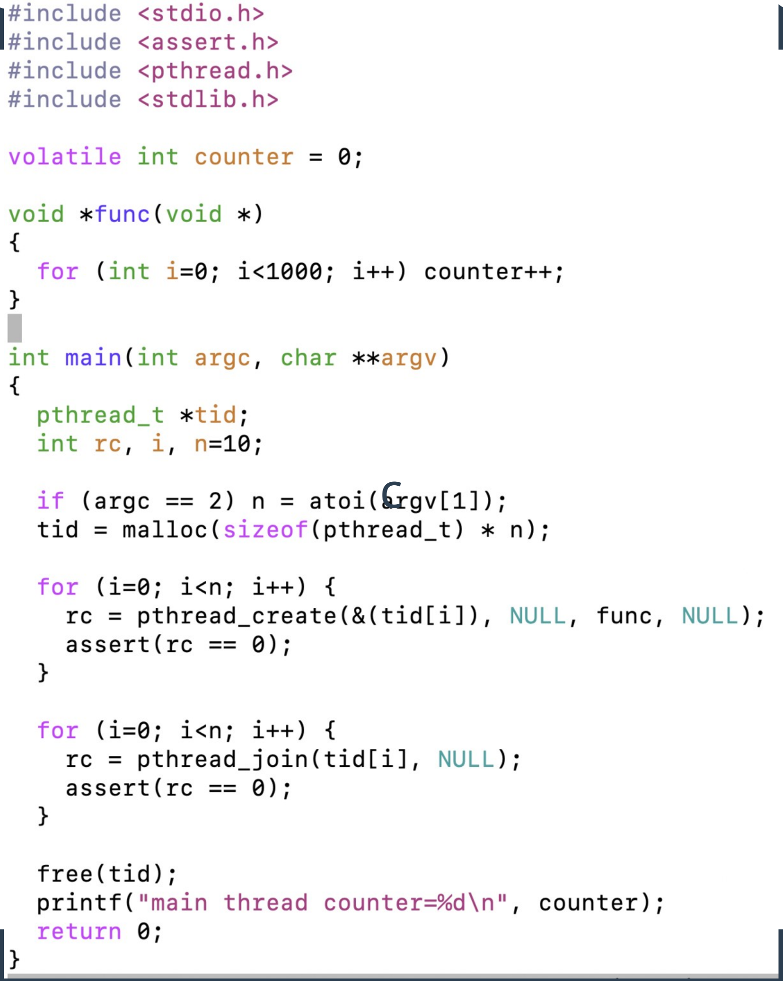
```
MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef
```



```
MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef
```

OK

Explicit Multi-processing with Threads: Shared memory: good, bad and the ugly



```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <stdlib.h>

volatile int counter = 0;

void *func(void *)
{
  for (int i=0; i<1000; i++) counter++;
}

int main(int argc, char **argv)
{
  pthread_t *tid;
  int rc, i, n=10;

  if (argc == 2) n = atoi(argv[1]);
  tid = malloc(sizeof(pthread_t) * n);

  for (i=0; i<n; i++) {
    rc = pthread_create(&(tid[i]), NULL, func, NULL);
    assert(rc == 0);
  }

  for (i=0; i<n; i++) {
    rc = pthread_join(tid[i], NULL);
    assert(rc == 0);
  }

  free(tid);
  printf("main thread counter=%d\n", counter);
  return 0;
}
```

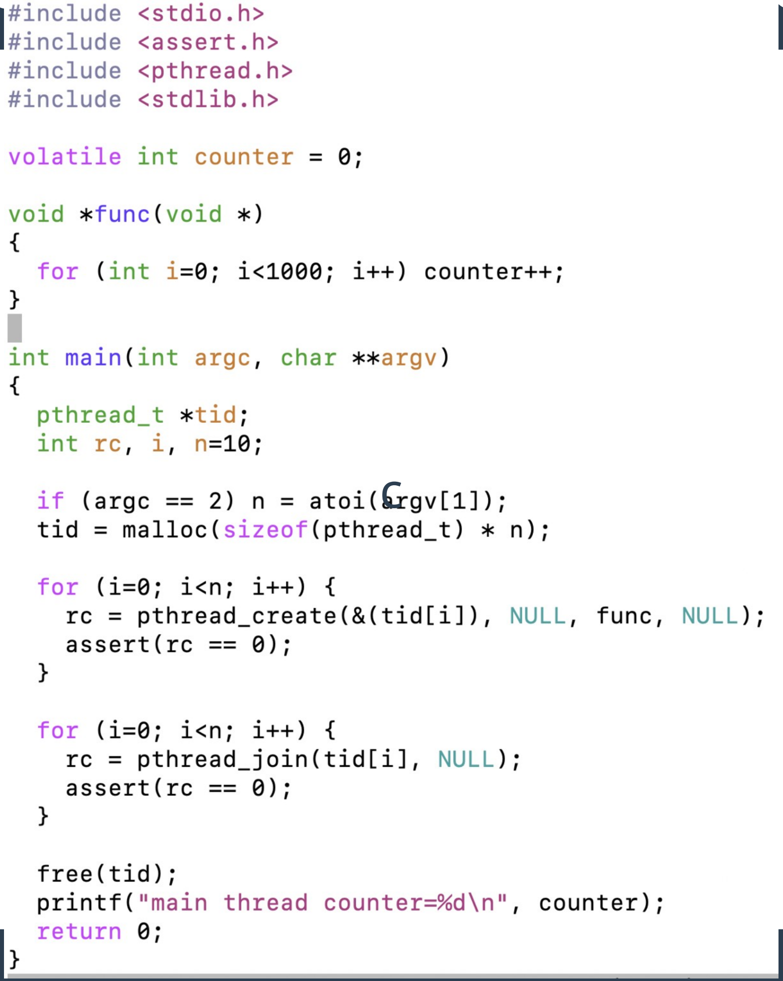
Basic problem: updating shared data structures concurrently

```
MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef
```

```
MOV 0xdeadbeef, %eax
ADD $1, %eax
MOV %eax, 0xdeadbeef
```

OK

Explicit Multi-processing with Threads: Shared memory: good, bad and the ugly



```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <stdlib.h>

volatile int counter = 0;

void *func(void *)
{
  for (int i=0; i<1000; i++) counter++;
}

int main(int argc, char **argv)
{
  pthread_t *tid;
  int rc, i, n=10;

  if (argc == 2) n = atoi(argv[1]);
  tid = malloc(sizeof(pthread_t) * n);

  for (i=0; i<n; i++) {
    rc = pthread_create(&(tid[i]), NULL, func, NULL);
    assert(rc == 0);
  }

  for (i=0; i<n; i++) {
    rc = pthread_join(tid[i], NULL);
    assert(rc == 0);
  }

  free(tid);
  printf("main thread counter=%d\n", counter);
  return 0;
}
```

Basic problem: updating shared data structures concurrently

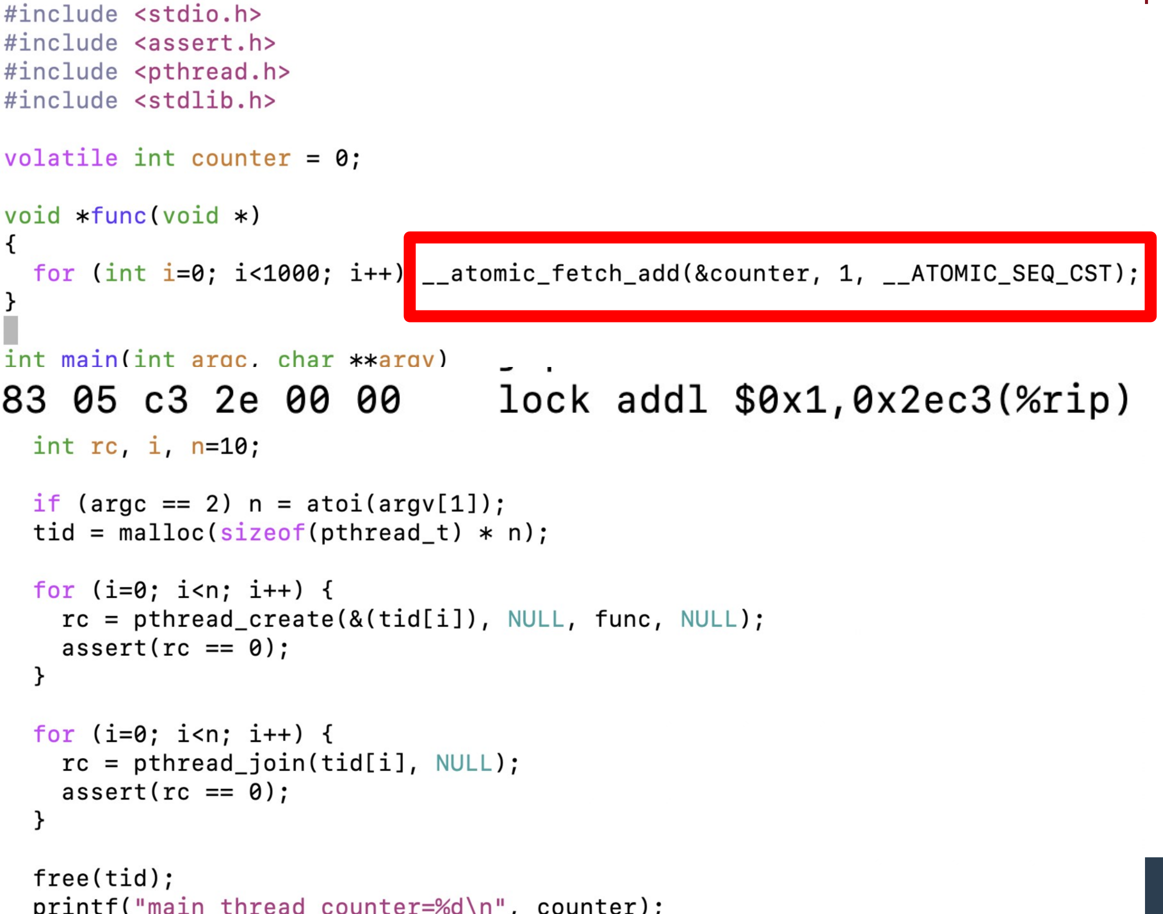
```
MOV 0xdeadbeef, %eax        MOV 0xdeadbeef, %eax
ADD $1, %eax                ADD $1, %eax
MOV %eax, 0xdeadbeef        MOV %eax, 0xdeadbeef
```

BAD : RACE

What would be the observed behaviour?

Explicit Multi-processing with Threads: Shared memory: good, bad and the ugly

- **Need to control concurrency when touching shared data structures**
- **Mutual Exclusion:**
  - HW atomic instructions
  - SW Locks and MUTEX's



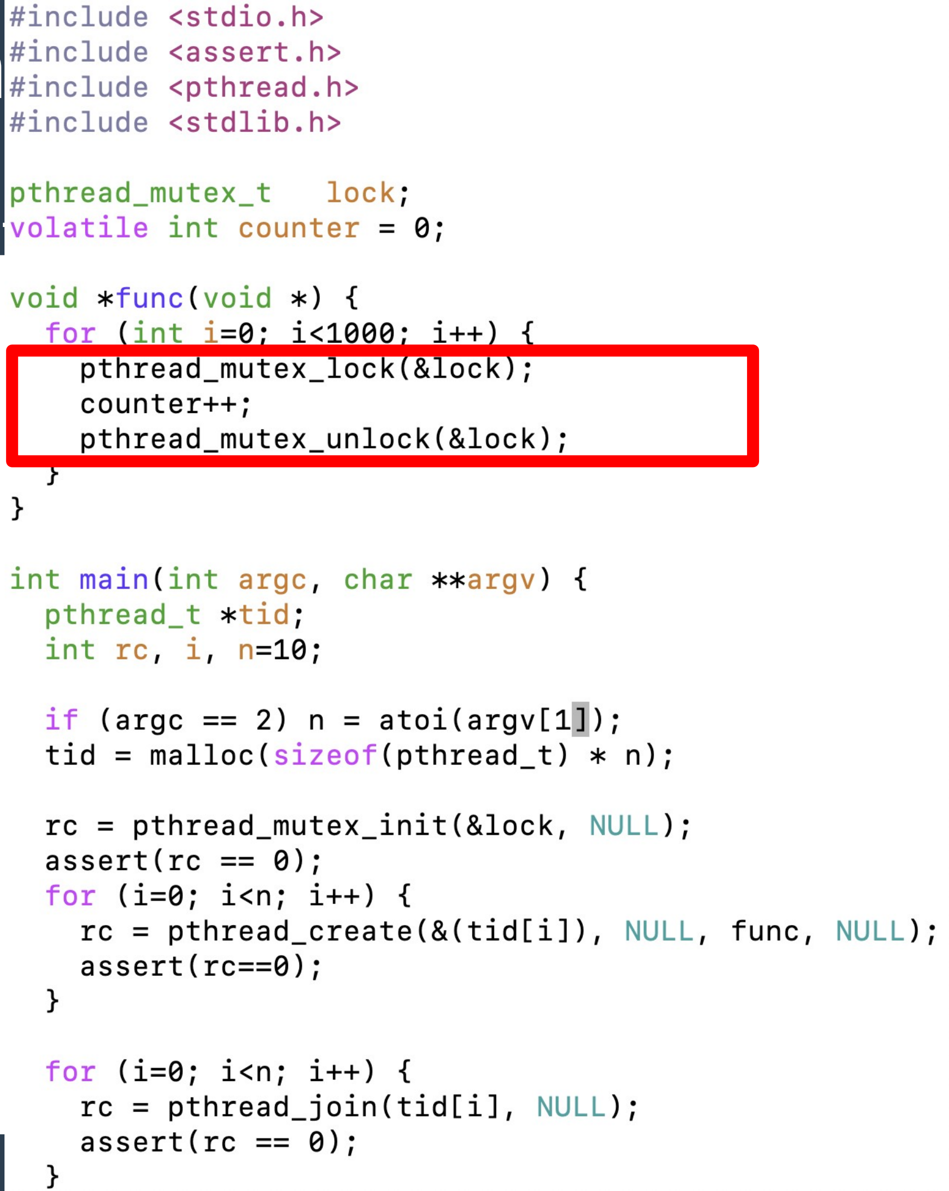
```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <stdlib.h>

volatile int counter = 0;

void *func(void *)
{
  for (int i=0; i<1000; i++) __atomic_fetch_add(&counter, 1, __ATOMIC_SEQ_CST);
}

int main(int argc, char **argv)
```

```
4011f5:      f0 83 05 c3 2e 00 00     lock addl $0x1,0x2ec3(%rip)
```

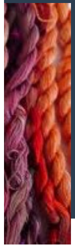
```
  int rc, i, n=10;

  if (argc == 2) n = atoi(argv[1]);
  tid = malloc(sizeof(pthread_t) * n);

  for (i=0; i<n; i++) {
    rc = pthread_create(&(tid[i]), NULL, func, NULL);
    assert(rc == 0);
  }

  for (i=0; i<n; i++) {
    rc = pthread_join(tid[i], NULL);
    assert(rc == 0);
  }

  free(tid);
  printf("main thread counter=%d\n", counter);
```

Explicit Multi-processin
Shared memory: good,

- **Need to control concurrency when touching shared data structures**
- **Mutual Exclusion:**
  - HW atomic instructions
  - SW Locks and MUTEX’s



```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <stdlib.h>

pthread_mutex_t   lock;
volatile int counter = 0;

void *func(void *) {
  for (int i=0; i<1000; i++) {
    pthread_mutex_lock(&lock);
    counter++;
    pthread_mutex_unlock(&lock);
  }
}

int main(int argc, char **argv) {
  pthread_t *tid;
  int rc, i, n=10;

  if (argc == 2) n = atoi(argv[1]);
  tid = malloc(sizeof(pthread_t) * n);

  rc = pthread_mutex_init(&lock, NULL);
  assert(rc == 0);
  for (i=0; i<n; i++) {
    rc = pthread_create(&(tid[i]), NULL, func, NULL);
    assert(rc==0);
  }

  for (i=0; i<n; i++) {
    rc = pthread_join(tid[i], NULL);
    assert(rc == 0);
  }
```



Explicit Multi-processing with Threads: Shared memory: good, bad and the ugly

- **But concurrency control is hard!**
  - Deadlocks
  - Live-locks
  - Bugs
  - Bad performance
    - Even when you don’t realize it

```
pthread_mutex_t lockA;
pthread_mutex_t lockB;
```

```
pthread_mutex_lock(lockA);
pthread_mutex_lock(lockB);
```

```
pthread_mutex_lock(lockB);
pthread_mutex_lock(lockA);
```

Explicit Multi-processing with Threads: Shared memory: good, bad and the ugly

- **But concurrency control is hard!**
  - Deadloc
  - Live-lock
  - Bugs
  - Bad performance
    - Even when you don't realize it

```
pthread_mutex_t lockA;
pthread_mutex_t lockB;
```

REMEMBER: Amdhal's Law

Locking/Mutex→ sequential execution → reduces parallelism → **limits scalability**

But correctness does matter ;-)

```
ck(lockB);
ck(lockA);
```

Memory Coherency Models are subtle

- **One there is only one thread what you expect is easy**
  - Read of a memory location should return value of last write
- **But when there are multiple threads what does last mean?**
  - A global total sequential order is expensive to implement
  - Many memory models have been proposed that weaken semantics to provide better performance:
    - Sequential, Processor, Weak, Release, Lazy Release, …
- **https://dl.acm.org/doi/pdf/10.1145/325096.325102**

Shared Memory Multiprocessors and Caches

Cache Coherency

Switch to keynote