Multiprocessors and Coherent Memory

Erik Hagersten
Uppsala University

DARK2 in a nutshell
1. Memory Systems (caches, VM, DRAM, microbenchmarks, ...)
2. Multiprocessors (TLP, coherence, memory models, interconnects, scalability, ...)
3. CPUs (pipelines, ILP, scheduling, Superscalars, VLIWs, embedded, ...)
4. Future: (physical limitations, TLP+ILP in the CPU,...)

DARK2 On the web
www.it.uu.se/edu/course/homepage/dark2/ht09

DARK2, Fall 09
Welcome!
News
FAQ
Schedule
Slides
New Papers
Assignments
Reading instr 4:ed
Exam

The era of the "Rocket Science Supercomputers" 1980-1995
- The one with the most blinking lights wins
- The one with the niftiest language wins
- The more different the better!

MP Taxonomy (more later...)

SIMD
MIMD

Message-passing
Shared Memory

Fine-grained
Coarse-grained

UMA
NUMA
COMA For now!

Models of parallelism
- Processes (fork or & in UNIX)
  - A parallel execution, where each process has its own process state, e.g., memory mapping
- Threads (thread_chreate in POSIX)
  - Parallel threads of control inside a process
  - There are some thread-shared state, e.g., memory mappings.
- Sverker will tell you more...
**Programming Model:**

- **Shared Memory**

**Adding Caches: More Concurrency**

- **Shared Memory**

**A more common MP: Multicore**

- **Multicore: Need to go parallel!**
  or do throughput computing, more later

**Caches: Automatic Replication of Data**

- **The Cache Coherent Memory System**
**The Cache Coherent $2$**

Shared Memory

Thread

Read A

Read A

Thread

Read A

Read A

Thread

Read A

Write A

Read A

**Summing up Coherence**

There can be many copies of a datum, but only one value.

Too strong definition!

There is a single global order of value changes to each datum.

**Implementation options for memory coherence**

- Two coherence options
  - Snoop-based ("broadcast")
  - Directory-based ("point to point")
- Different memory models
- Varying scalability

**Snoop-based Protocol Implementation**

"BUS"

Shared Memory

Cache

A-tag

State

Data

BUS snoops

CPU access

CPU

Example: Bus Snoop MOSI

BUSrta: ReadToShare (reading the data with the intention to read it)

BUSrtw, ReadToWrite (reading the data with the intention to modify it)

BUSwb: Writing data back to memory

BUSinv: Invalidating other copies

**Snoop-based Protocol Implementation**

BUS

Shared Memory

Cache

A-tag

State

Data

BUS snoops

CPU access

CPU
Example: CPU access MOSI

- CPUwrite: Caused by a store miss
- CPUread: Caused by a load miss
- CPUrepl: Caused by a replacement

"Upgrade" in snoop-based

Directory-based coherence: per-cacheline info in the memory

Cache-to-cache in dir-based

Cache-to-cache in snoop-based

Directory Protocol

Read Request

Read Demand

Forward

Bus RTS

My RTS

Gotta answer

Wait for data

Thread

Thread

Thread

Thread

Thread

Thread

Thread

Thread

Thread
A New Kind of Cache Miss

- Capacity – too small cache
- Conflict – limited associativity
- Compulsory – accessing data the first time
- Communication (or “Coherence”) [Jouppi]
  - Caused by downgrade (modified→shared)
    “A store to data I had in state M, but now it’s in state S”
  - Caused my invalidation (shared→invalid)
    “A load to data I had in state S, but now it’s been invalidated”

Why snoop?

- A ”bus”: a serialization point helps coherence and memory ordering
- Upgrade is faster [producer/ consumer and migratory sharing]
- Cache-to-cache is much faster [i.e., communication...]
- Synchronization, a combination of both
  - ...but it is hard to scale the bandwidth

Why directory-based

- P2P messages → high bandwidth
- Suits out-of-the-box coherence
- Note:
  - Dir-based can be used to build a uniform-memory architecture (UMA)
  - Bandwidth will be great!!
  - Memory latency will be OK
  - Cache-to-cache latency will not!

Update Instead of Invalidate?

- Write the new value to the other caches holding a shared copy (instead of invalidating...)
- Will avoid coherence misses
- Consumes a large amount of bandwidth
- Hard to implement strong coherence
- Few implementations: SPARCCenter2000, Xerox Dragon

Update in MOSI snoop-based

![Diagram showing update in MOSI snoop-based]

Implementing Coherence (and Memory Models...)

Erik Hagersten
Uppsala University
Sweden
Snoop-based Protocol Implementation

Shared Memory

Common Cache States
- M – Modified
  - My dirty copy is the only cached copy
- E – Exclusive
  - My clean copy is the only cached copy
- O – Owner
  - I have a dirty copy, others may also have a copy
- S – Shared
  - I have a clean copy, others may also have a copy
- I – Invalid
  - I have no valid copy in my cache

Some Coherence Alternative
- MSI
  - Writeback to memory on a cache2cache.
- MOSI
  - Leave one dirty copy in a cache on a cache2cache.
- MOESI
  - The first reader will go to E and can later write cheaply

The Cache Coherent Memory System

Upgrade – the requesting CPU

Upgrade – the other CPUs

Some Coherence Alternative
- MSI
  - Writeback to memory on a cache2cache.
- MOSI
  - Leave one dirty copy in a cache on a cache2cache.
- MOESI
  - The first reader will go to E and can later write cheaply

Upgrade – the requesting CPU

Upgrade – the other CPUs
Modern snoop-based architecture -- dual tags

Modern snoop-based architecture -- dual tags

"Upgrade" in snoop-based

The Cache Coherent Cache-to-cache

Cache-to-cache – the requesting CPU

Cache-to-cache – the other CPU

Cache-to-cache in snoop-based
While (A != my_id) {} /* this is a primitive kind of lock */
B := B + A * 2;
A := A + 1; /* this is a primitive kind of unlock */

Initially, CPU1 has its local variable my_id=1, CPU has my_id=2 and CPU3 has my_id=3 and the globally shared variables A is equal to 1 and B is equal to 0. CPU2 and 3 are starting slightly ahead of CPU1 and will execute the first while statement before CPU1. Initially, both A and B only reside in memory.

The following four bus transaction types can be seen on the snooping bus connecting the CPUs:
- RTS: ReadToShare (reading the data with the intention to read it)
- RTW, ReadToWrite (reading the data with the intention to modify it)
- WB: Writing data back to memory
- INV: Invalidating other caches copies

Show every state change and/or value change of A and B in each CPU's cache according to one possible interleaving of the memory accesses. After the parallel execution is done for all of the CPUs, the cache lines still in the caches will be replaced. These actions should also be shown. For each line, also state what bus transaction occurs on the bus (if any) as well as which device is providing the corresponding data (if any).

False sharing

Communication misses even though the threads do not share data "the cache line is too large"

Memory Ordering

- Coherence defines a per-datum valuechange order
- Memory model defines the valuechange order for all the data.

Q: What value will get printed?
Dekker’s Algorithm

Initially \( A = B = 0 \)

“fork”

\[ A := 1 \]

if \((B == 0)\) print("A won")

\[ B := 1 \]

if \((A == 0)\) print("B won")

Q: Is it possible that both A and B win?

Memory Ordering

- Defines the guaranteed memory ordering
- Is a “contract” between the HW and SW guys
- Without it, you can not say much about the result of a parallel execution

In which order were these threads executed?

Thread 1

1. LD A
2. ST B
3. LD C
4. ST D
5. LD E
...

Thread 2

1. LD B
2. ST C
3. LD D
4. ST E
...

“fork”

One possible observed order

Thread 1

1. LD A
2. ST B
3. LD C
4. ST D
5. LD E
...

Thread 2

1. ST A
2. ST B
3. ST C
4. ST D
5. ST E
...

Another possible observed order

Thread 1

1. LD A
2. ST B
3. LD C
4. ST D
5. LD E
...

Thread 2

1. ST A
2. ST B
3. ST C
4. ST D
5. ST E
...

“The intuitive memory order”

Sequential Consistency (Lamport)

- Global order achieved by interleaving all memory accesses from different threads
- “Programmer’s intuition is maintained”
  - Store causality? Yes
  - Does Dekker work? Yes
  - Unnecessarily restrictive \( \Rightarrow \) performance penalty

Dekker’s Algorithm

Initially \( A = B = 0 \)

“fork”

\[ A := 1 \]

if \((B == 0)\) print("A won")

\[ B := 1 \]

if \((A == 0)\) print("B won")

Q: Is it possible that both A and B win?
### Sequential Consistency (SC) Violation

**Dekker: both wins**

\[ A := B := 0 \]

\[ A := 1 \]
\[ \text{If } (B == 0) \]
\[ \text{print } "Left wins" \]

\[ B := 1 \]
\[ \text{If } (A == 0) \]
\[ \text{print } "Right wins" \]

Both Left and Right wins \( \Rightarrow \) SC violation

**Cyclic access graph \( \Rightarrow \) Not SC (there is no global order)**

\[ A := B := 0 \]

\[ ST A, 1 \]
\[ LD B \]
\[ \Rightarrow 0 \]

\[ ST B, 1 \]
\[ LD A \]
\[ \Rightarrow 0 \]

SC violation

**SC is OK if one thread wins**

Only Right wins \( \Rightarrow \) SC is OK

\[ A := B := 0 \]

\[ A := 1 \]
\[ \text{If } (B == 0) \]
\[ \text{print } "Left wins" \]

\[ B := 1 \]
\[ \text{If } (A == 0) \]
\[ \text{print } "Right wins" \]

Not cyclic graph \( \Rightarrow \) SC

**SC is OK if no thread wins**

No thread wins \( \Rightarrow \) SC is OK

\[ A := B := 0 \]

\[ A := 1 \]
\[ \text{If } (B == 0) \]
\[ \text{print } "Left wins" \]

\[ B := 1 \]
\[ \text{If } (A == 0) \]
\[ \text{print } "Right wins" \]

Not cyclic graph \( \Rightarrow \) SC

**Four Partial Orders, still SC**

STB < LDA ; STA < LDA ; STB < LDB ; STA < LDA

One global order:

STB < LDA < STA < LDB

**“Almost intuitive memory model”**

Total Store Ordering [TSO] (P. Sindhu)

- Global *interleaving* [order] for all stores from different threads (own stores excepted)
- “Programmer’s intuition is maintained”
  - Store causality? Yes
  - Does Dekker work? No
- Unnecessarily restrictive \( \Rightarrow \) performance penalty

**TSO HW Model**

Stores are moved off the critical path

Coherence implementation can be the same as for SC
### TSO
- Flag synchronization works
  \[ A := \text{data} \quad \text{while} \ (\text{flag} \neq 1) \ \{ \}; \]
- Provides causal correctness

Q: What value will get printed? Answer: 1

### Dekker’s Algorithm, TSO

Initially \( A = B = 0 \)

\[ \text{fork} \]

\[ A = 1 \]

\[ \text{membarrier} \]

\[ B := 1 \]

\[ \text{membarrier} \]

\[ \text{print}("A won") \]

\[ \text{if} \ (B = 0) \ \text{print}("B won") \]

Q: Is it possible that both A and B win?

Answer: Yes, it can happen.

### Weak/Release Consistency

(M. Dubois, K. Gharachorloo)

- Most accesses are unordered
- "Programmer’s intuition is not maintained"
  - Store causality: No
  - Does Dekker work? No
- Global order only established when the programmer explicitly inserts memory barrier instructions
- Better performance!!
- Interesting bugs!!

### Example 1: Causal Correctness Issues

Q: What value will get printed? Answer: 1
Example 1: Causal Correctness Issues

Shared Memory

Thread
Read A
Write B
Read A

A: if store causality \(\Rightarrow\) “1” will be printed

B: if store causality \(\Rightarrow\) “1” will be printed

What is the value of A? It depends...
Dekker’s Algorithm

Initially \( A = B = 0 \)

```plaintext
fork

A := 1
if (B == 0) print("A won")

B := 1
if (A == 0) print("B won")
```

Q: Is it possible that both A and B win?

A: Only known if you know the memory model

Learning more about memory models

Shared Memory Consistency Models: A Tutorial
by Sarita Adve, Kourosh Gharachorloo
in IEEE Computer 1996 (in the “Papers” directory)

RFM: Read the F*****n Manual of the system you are working on!
(Different microprocessors and systems supports different memory models.)

Issue to think about:
What code reordering may compilers really do?
Have to use “volatile” declarations in C.

X86’s new memory model

- Processor consistency with causal correctness for non-atomic memory ops
- TSO for atomic memory ops

Video presentation:
http://www.youtube.com/watch?v=WlFvVDsA8hI

See section 8.2 in this manual:

Processor Consistency [PC] (J. Goodman)

- PC: The stores from a processor appears to others in program order
- Causal correctness (often added to PC): if a processor observes a store before performing a new store, the observed store must be observed before the new store by all processors
- Flag synchronization works.
- No causal correctness issues

Synchronization

Erik Hagersten
Uppsala University
Sweden

Execution on a sequentially consistent shared-memory machine:

Sum := 0

while (sum < N)

sum := sum + 1

What value will be printed?

A: any value between \( N \) and \( N + 3 \)

How many addition will get executed?

A: any value between \( N \) and \( N + 4 \)
Need to introduce synchronization

- Locking primitives are needed to ensure that only one process can be in the critical section:

```plaintext
LOCK(lock_variable) /* wait for your turn */
if (sum > threshold) {
    sum := my_sum + sum
} UNLOCK(lock_variable) /* release the lock*/
```

if (sum > threshold) {
    LOCK(lock_variable) /* wait for your turn */
    sum := my_sum + sum
    UNLOCK(lock_variable) /* release the lock*/
}

Components of a Synchronization Event

- Acquire method
  - Acquire right to the synch (enter critical section, go past event)

- Waiting algorithm
  - Wait for synch to become available when it isn't

- Release method
  - Enable other processors to acquire right to the synch

Atomic Instruction to Acquire

Atomic example: test&set “TAS” (SPARC: LDSTB)
- The value at Mem(lock_addr) loaded into the specified register
- Constant “1” atomically stored into Mem(lock_addr) (SPARC: “FF”)
- Software can determine if won (i.e., set changed the value from 0 to 1)
- Other constants could be used instead of 1 and 0

Looks like a store instruction to the caches/memory system
Implementation:
1. Get an exclusive copy of the cache line
2. Make the atomic modification to the cached copy

Other read-modify-write primitives can be used too
- Swap (SWAP): atomically swap the value of REG with Mem(lock_addr)
- Compare&swap (CAS): SWAP if Mem(lock_addr)==REG2

Waiting Algorithms

- Blocking
  - Waiting processes/threads are de-scheduled
  - High overhead
  - Allows processor to do other things

- Busy-waiting
  - Waiting processes repeatedly test a lock_variable until it changes value
  - Releasing process sets the lock_variable
  - Lower overhead, but consumes processor resources
  - Can cause network traffic

Hybrid methods: busy-wait a while, then block

Release Algorithm

- Typically just a store “0”
- More complicated locks may require a conditional store or a “wake-up”.

A Bad Example: “POUNDING”

```plaintext
proc lock(lock_variable) {
    while (TAS(lock_variable)==1) {} /* bang on the lock until free */
}
proc unlock(lock_variable) {
    lock_variable := 0
}
Assume: The function TAS (test and set)
-- returns the current memory value and atomically writes the busy pattern “1” to the memory
Generates too much traffic!!
-- spinning threads produce traffic!
```
**Optimistic Test&Set Lock "spinlock"**

```c
proc lock(lock_variable) {
    while true {
        if (TAS[lock_variable] == 0) break; /* bang on the lock once, done if TAS==0 */
        while(lock_variable != 0) {} /* spin locally in your cache until "0" observed*/
    }
}

proc unlock(lock_variable) {
    lock_variable := 0
}
```

---

**It could still get messy!**

```plaintext
..messy (part 2)

N-1 Test&Set
(i.e., N writes)

```

---

**Could Get Even Worse on a NUMA**

- Poor communication latency
- Serialization of accesses to the same cache line
- WF: added hardware optimization:
  - TAS can bypass loads in the coherence protocol
  - The winner’s atomic TAS will bypass the loads
  - The loads will return “busy”

---

**Ticket-based queue locks: "ticket"**

```c
proc lock(lock_struct) {
    int my_num;
    my_num := INC(lock_struct.ticket) /* get your unique number*/
    while(my_num > lock_struct.nowserving) {} /* wait here for your turn */
}

proc unlock(lock_struct) {
    lock_struct.nowserving++ /* next in line please */
}
```

---

**Ticket-based back-off "TBO"**

```c
proc lock(lock_struct) {
    int my_num;
    my_num := INC(lock_struct.ticket) /* get your number*/
    while(my_num > lock_struct.nowserving) {} /* my turn */
    idel_wait(lock_struct.nowserving - my_num) /* do other shopping */
}

proc unlock(lock_struct) {
    lock_struct.nowserving++ /* next in line please */
}
```

---
Queue-based lock: CLH-lock

"Initially, each process owns one global cell, pointed to by private *I and *P. Another global cell is pointed to by global *L "lock variable"
1) Initialize the *I flag to busy (= "1")
2) Atomically, make *L point to "our" cell and make "our" *P point where *L's cell
3) Wait until *P points to a "0"

proc lock(int **L, **I, **P)
{  **I = 1; /* initialized "our" cell as "busy"*/
    atomic_swap {*P =*L; *L=*P} /* P now stores a pointer to the cell L pointed to*/
    /* L now stores a pointer to our cell */
    while (**P != 0){} /* keep spinning until prev owner releases lock*/
}

proc unlock(int **I, **P)
{ **I = 0; /* release the lock */
    *I =*P; /* next time *I to reuse the previous guy's cell*/
}

In CS
lock(int **L, **I, **P) {
    **I = 1; /* init to "busy"*/
    atomic_swap (**P = *L; *L = *P;)
    /* *L now point to our I */
    while (**P != 0) {} } /* spin unit prev is done */

unlock(int **I, **P) {
    **I = 0; /* release the lock */
    *I = *P; } /* reuse the previous guy's *P*/

Minimizes traffic at lock handover!
May be too fair for NUMAs ....
E6800 locks 12 CPUs

```
0  2  4  6  8  10  12  14  16  18  20  22  24  26  28  30  32
Processors

0,00  0,05  0,10  0,15  0,20  0,25  0,30  0,35  0,40
Time / Processors

TATAS  TATAS_EXP  MCS  CLH  RH  spin  spin_exp  MCS-queue  CLH-queue  RH-locks
```

E6800 locks (excluding POUND)

```
0  2  4  6  8  10  11  12
# Contenders

0  500  1,000  1,500  2,000  2,500  3,000
Avg. time per CS job

SPIN  TICKET  TBO  CLH
```

NUMA: Non-uniform Comm Arch.

```
CPU  CPU  Mem  CPU  CPU  Mem  CPU  CPU  Switch  CPU  CPU  Mem  CPU  CPU  Mem  CPU  CPU  I/F  CPU  CPU  Mem  CPU  CPU  Mem  CPU  CPU  I/F  …  Switch  Snoop
```

Directory-latency = 6x snoop i.e., roughly CMP NUCA-ness

Trad. chart over lock performance on a hierarchical NUMA (round robin scheduling)

```
0,00  0,05  0,10  0,15  0,20  0,25  0,30  0,35  0,40
Time / Processors

TATAS  TATAS_EXP  MCS  CLH  RH  spin  spin_exp  MCS-queue  CLH-queue  RH-locks
```

Introducing RH locks

```
for i = 1 to 10000 {
    lock(AL)
    A := A + 1;
    unlock(AL)
}
```

RH locks: encourages unfairness

```
0  2  4  6  8  10  11  12  14  16  18  20  22  24  26  28  30  32
Processors

0,00  0,05  0,10  0,15  0,20  0,25  0,30  0,35  0,40
Time per lock handover

spin  spin_exp  MCS-queue  CLH-queue  RH-locks

0  2  4  6  8  10  11  12  14  16  18  20  22  24  26  28  30  32
Processors

0,00  0,05  0,10  0,15  0,20  0,25  0,30  0,35  0,40
Node migration (%)

spin  spin_exp  MCS-queue  CLH-queue  RH-locks
```
Barriers: Make the first threads wait for the last thread to reach a point in the program

1. Software algorithms implemented using locks, flags, counters
2. Hardware barriers
   - Wired-AND line separate from address/data bus
   - Set input high when arrive, wait for output to be high to leave
   - (In practice, multiple wires to allow reuse)
   - Difficult to support arbitrary subset of processors
     - Even harder with multiple processes per processor
   - Difficult to dynamically change number and identity of participants
     - E.g. latter due to process migration

A Centralized Barrier

```c
BARRIER (bar_name, p) {
    int loops;
    loops = 0;
    local_sense = !(local_sense);
    /* toggle private sense variable each time the barrier is used */

    LOCK(bar_name.lock);
    bar_name.counter++;
    /* globally increment the barrier count */
    if (bar_name.counter == p) { /* everybody here yet? */
        bar_name.flag = local_sense;
        /* release waiters*/
        UNLOCK(bar_name.lock)
    } else {
        UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) { /* wait for the last guy */
            if (loops++ > UNREASONABLE) report_warning(pid)
        }
    }
}
```

Centralized Barrier Performance

- Latency
  - Want short critical path in barrier
  - Centralized has critical path length at least proportional to $p$
- Traffic
  - Barriers likely to be highly contended, so want traffic to scale well
  - About $3p$ bus transactions in centralized
- Storage Cost
  - Very low: centralized counter and flag
- Key problems for centralized barrier are latency and traffic
  - Especially with distributed memory, traffic goes to same node
- Hierarchical barriers

New kind of synchronization: Transactional Memory (TM)

- OLD SYNC: lock(ID); unlock(ID) around critical sections
- TM: start_transaction; end_transaction around “critical sections” (note: no ID!)
  - Underlying mechanism to guarantee atomic behavior often by rollback mechanisms
  - This is not the same as guaranteeing that only one thread is in the critical action!!
  - Supported in HW or in SW (normally very inefficient)
- Suggested by Maurice Herlihy in 1993
- HW support announced by Sun (??)
Support for TM

- Start_transaction:
  - Save original state to allow for rollback (i.e., save register values)
- In critical section:
  - Do not make any global state change
  - Detect “atomic violations” (others writing data you’ve read in CS or reading data you have written)
  - At atomic violation: roll-back to original state
  - Forward progress must be guaranteed
- End_transaction
  - Atomically commit all changes performed in the critical section.

Advantage of TM

- Do not have to “name” CS
- Less risk for deadlocks
- Performance:
  - Several thread can be in “the same” CS as long as they do not mess with each other
  - CS can often be large with a small performance penalty

Introduction to Multiprocessors

Erik Hagersten
Uppsala University

Flynn’s Taxonomy

{Single, Multiple}Instruction + {Single, Multiple}Data

- SISD - Our good old “simple” CPUs
- SIMD - Vectors, “MMX”, DSPs, CM-2,...
- MIMD – TLP, cluster, shared-mem MP,...
- MISD – Can’t think of any...

MP Taxonomy

SIMD

MIMD

Message-passing

Shared Memory

Fine-grained

Coarse-grained

UMA

NUMA

COMA

SIMD = “Dataparallelism”

Program:
**SIMD: Thinking Machine**
- Connection Machine: CM1, CM2, CM200 (at KTH ~1990: CM200 “Bellman”)
- One-bit ALU and a small local memory
- FP accelerator available
- Programmed in “ASM”, *C and *Lisp
- Hard to program (in my opinion...)

**Other Flavors of SIMD**
- MMX/AltiVec/VIS instructions/SSE...
  - Divide register content into smaller items (e.g., bytes)
  - Special instructions operate on all items in parallel, e.g., BYTE-COMPARE...
- Some DSPs (Digital Signal Processors)
- Some Image Processors

**Vector architectures**
- CRAY, NEC, Fujitsu, Also x86 extensions: e.g., SSE instruction
- Vectory Processors
  - LD/ST operate on vectors of data
  - ALU Ops operate on vectors of data
- Example:
  - 8 “vector register” contain 64 vector “words” each
  - A single LD/ST instr loads/stores entire vectors
  - A single ALU instr V1 ← V2 op V3
  - 64 bit mask vectors make execution conditional
  - Overlaps Mem and ALU ops
  - One form of “SIMD” — Single Instruction Multiple Data

**MIMD: Message-passing**
- Program: send
- Program: receive
- Program:...
- Program:...

**Message-Passing HW**
- Programmed in MPI or PVM (or HPFortran...)
  - Thinking Machines: CM5
  - Intel: Paragon
  - IBM: SP2
  - Meiko (Bristol, UK!!): CS2
- Today: Clusters with high-speed interconnect (Important today, but not covered in this course)
- Clusters can be used as message-passing HW, put is most often used as capacity computing (i.e., throughput computing)
**Dataflow**

- Often programmed in functional languages (e.g., ID)
- Compile program to Dataflow graph
- Operands + graph = executable
- Operation ready when the source operands are available

**Dataflow Example:**

```plaintext
X := A + B
Y := C + D
If (X > Y)
    output X
else
    output Y
```

**Static Dataflow (Dennis)**

- Each operand executed exactly once per program
- Location assigned for each input data

**Dynamic Dataflow (Arvind)**

- Allows for recursion and loops
- Each invocation is assigned a “color”
- Pairs of operands are matched dynamically
  - Based on {Color, Operation}
  - In the Waiting-Matching Section (i.e., a cache)
- One problem: too much parallelism in the wrong place

**Carlstedts Elektronik**

Gunnar Carlstedt, Staffan Truve' et al

- Processor “8601”
- Gothenburg 1990-1997
- Functional language "H"
- Execution performed by a reduction a CAM memory
- ALU rarely used
- Many parallel processors on a wafer (Wafer-scale integration)
- CRT (Carlstedt Research Technology)
Today’s Topic

SIMD  MIMD

Message-passing  Shared Memory

Fine-grained  Coarse-grained

UMA  NUMA  COMA

The server market 1995

The target of the rocket science supercomputers

UNIX  shared-mem servers

- $10k: 1%
- $50k: 5%
- $250k: 5%
- $1M: 2%
- >$1M: 3%