Multiprocessors and Coherent Memory

Erik Hagersten
Uppsala University

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!

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_chreates 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:
Adding Caches: More Concurrency

Shared Memory

Caches: Automatic Replication of Data

Shared Memory

The Cache Coherent Memory System

The Cache Coherent $2$

Summing up Coherence

Too strong definition!

Implementation options for memory coherence

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

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

There is a single global order of value changes to each datum.
**Snoop-based Protocol Implementation**

"BUS" Shared Memory

Cache

CPU access

CPU access

**Example: Bus Snoop MOSI**

BUSrtw: 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 caches copies

**Example: CPU access MOSI**

CPUwrite: Caused by a store miss
CPUread: Caused by a load miss
CPUrep: Caused by a replacement

**“Upgrade” in snoop-based**

A: 

B: 

| Read A |
| Read B |

BusINV

Have to INV

My INV

Have to INV

Thread

Thread

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

Directory Protocol

Thread

Thread

Thread
"Upgrade" in dir-based

Thread $A$

Who has a copy

Thread $B$

Who has a copy

Cache-to-cache in dir-based

Thread $A$

Who has a copy

Thread $B$

Who has a copy

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\(\rightarrow\)shared)
    "A store to data I had in state M, but now it's in state S" 😊
  - Caused my invalidation (shared\(\rightarrow\)invalid)
    "A load to data I had in state S, but now it's been invalidated" 😊

Why directory-based

- P2P messages \(\rightarrow\) 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!

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 😊
**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: SPARC Center 2000, Xerox Dragon

**Update in MOSI snoop-based**

- 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: SPARC Center 2000, Xerox Dragon

**Implementing Coherence (and Memory Models...)**

Erik Hagersten
Uppsala University
Sweden

**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

**Snoop-based Protocol Implementation**
The Cache Coherent Memory System

Upgrade – the requesting CPU

Upgrade – the other CPUs

Modern snoop-based architecture -- dual tags

"Upgrade" in snooped-based

The Cache Coherent Cache-to-cache
All the three RISC CPUs in a MOSI shared-memory sequentially consistent multiprocessor executes the following code almost at the same time:

```c
while(A := my_id{}); /* this is a primitive kind of lock*/
B := B + A * 2;
while (A != 4) {}; /* this is a primitive kind of barrier*/
A := A + 1;         /* this is a primitive kind of unlock */
B := B + A * 2;
while(A != my_id){};   /* this is a primitive kind of lock */
```

Initially, CPU1 has its local variable my_id=1, CPU2 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. CPU1 and CPU2 are starting slightly ahead of CPU3 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:

- **BUSrts**: 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 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).

### Example of a state transition sheet:

<table>
<thead>
<tr>
<th>CPU action</th>
<th>Data transferred</th>
<th>Instructions after the CPU action</th>
<th>Data provided by</th>
<th>State of the CPU</th>
<th>From CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU1</td>
<td></td>
<td>CPU2</td>
<td></td>
<td>CPU1</td>
<td>CPU2</td>
</tr>
<tr>
<td>CPU2</td>
<td></td>
<td>CPU3</td>
<td></td>
<td>CPU2</td>
<td>CPU3</td>
</tr>
<tr>
<td>CPU3</td>
<td></td>
<td>CPU1</td>
<td></td>
<td>CPU3</td>
<td>CPU1</td>
</tr>
</tbody>
</table>

Here is the state transition sheet for one possible execution order: CPU1 -> CPU2 -> CPU3.
False sharing

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

Memory Ordering
(aka Memory Consistency)
-- tricky but important stuff

Erik Hagersten
Uppsala University
Sweden

Memory Ordering

- Coherence defines a per-datum value change order
- Memory model defines the value change 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

Thread 2

LD A
ST B
LD C
ST D
LD E
....

LD A happened before ST A’

ST A’
LD B’
ST C’
LD D
ST E’
....

....
....
One possible observed order

Thread 1
- LD A
- ST A'
- LD C
- ST C'
- LD D
- ST D'
- LD E
- ST E'

Thread 2
- LD B'
- ST C'
- LD D'
- ST E'

Another possible observed order

Thread 1
- LD A
- ST B'
- LD C
- ST C'
- LD D
- ST D'
- LD E
- ST E'

Thread 2
- LD B'
- ST C'
- LD D'
- ST E'
- ST A'

"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 ==> 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
if (B == 0) print("Left wins")
B := 1
if (A == 0) print("Right wins")
```

- Both Left and Right wins
- SC violation
- Not cyclic graph => SC
- One global order:
  - STB < LDA < STA < LDB

SC is OK if one thread wins

- Only Right wins => SC is OK
- Not cyclic graph => SC

SC is OK if no thread wins

- No thread wins => SC is OK
- Four Partial Orders, still SC
  - STB < LDA; STA < LDA; STB < LDB; STA < LDA
One implementation of SC in dir-based (...without speculation)

```
Thread
Read A
... READ X ...
Read X must complete before starting Read A
Read A
Must receive all ACKs before continuing
Thread
Thread
```

“Almost intuitive memory model”
Total Store Ordering (P. Sindhu)

- Global order achieved by interleaving all store accesses from different threads
- “Programmer’s intuition is maintained”
  - Store causality? Yes
  - Does Dekker work? No
- Unnecessarily restrictive => performance penalty

TSO HW Model

Stores are moved off the critical path
Coherence implementation can be the same as for SC

Dekker’s Algorithm

```
Initially A = B = 0

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

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

Dekker’s Algorithm for TSO

```
Initially A = B = 0

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

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

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

Left: The read (i.e., test if B==0) can bypass the store (A=1)
Right: The read (i.e., test if A==0) can bypass the store (B=1)
Both loads can be performed before any of the stores
Yes, it is possible that both wins
Dekker’s algorithm breaks

Q: Is it possible that both A and B wins?
Membar: The read is stared after all previous stores have been “globally ordered”
Behaves like SC
Dekker’s algorithm works!
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!!

Weak/Release consistency

- New flag synchronization needed
- Dekker's: same as TSO
- Causal correctness provided for this code

Q: What value will get printed? Answer: 1

Example 1: Causal Correctness Issues

Initially A = B = 0

\[\text{membarrier; membarrier; flag := 1; \{X := A; \}}\]

\[\text{membarrier; membarrier; flag := 0; \{Y := A; \}}\]

\[\text{Print A; Print B; }\]

Example 2: Causal Correctness Issues

Initially A = B = 0

\[\text{membarrier; membarrier; flag := 1; \{X := A; \}}\]

\[\text{membarrier; membarrier; flag := 0; \{Y := A; \}}\]

\[\text{Print A; Print B; }\]
Example 1: Causal Correctness Issues

Shared Memory

Thread

Thread

Thread

A:

Thread

A:=1

...

While (A==0) {}

B := 1

B:

While (B==0) {}

Print A

INV

What is the value of A?

It depends...

A: if store causality → "1" will be 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?

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 support different memory models.)

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

Synchronization

Erik Hagersten
Uppsala University
Sweden
Execution on a sequentially consistent shared-memory machine:

```
while (sum < N)
  sum := sum + 1
```

What value will be printed?
- A: any value between N and N + 3
- B: any value between N and N * 4

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"

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"

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
}

Much less coherence traffic!! -- still lots of traffic at lock handover!

It could still get messy!

```
L==1
0 0 0 0 0 0 0 0 0 0
Interconnect
L==0
X X X X X X X X X
Interconnect
N reads
L==0 0 0 0 0 0 0 0 0 0
Interconnect
```

...messy (part 2)

```
N-1 Test&Set (i.e., N writes)
T&S T&S T&S T&S T&S ...
Interconnect
L==1
CS 0 0 0 0 0 0 0 0 0
Interconnect
L==0
0 0 0 0 0 0 0 0 0
Interconnect
```

potentially: ~N*N/2 reads :-(

Problem1: Contention on the interconnect slows down the CS proc
Problem2: The lock hand-over time is N*read_throughput
Fix1: Some back-off strategy, bad news for hand-over latency
Fix2: Queue-based locks

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
    => N-2 loads queue up in the protocol
  - the winner’s atomic TAS will bypass the loads
  - the loads will return “busy”

Ticket-based queue locks: "ticket"

Proc lock(istruct) {
    int my_num;
    my_num := INC(istruct.ticket) /* get your unique number*/
    while(my_num > istruct.nowserving) {} /* wait here for your turn */
}
Proc unlock(istruct) {
    istruct.nowserving++ /* next in line please */
}

Less traffic at lock handover!
Ticket-based back-off "TBO"

```
proc lock(lstruct) {
    my_num := INC(lstruct.ticket) /* get your number*/
    while(my_num != lstruct.nowserving) /* my turn?*/
        idle_wait(lstruct.nowserving - my_num) /* do other shopping*/
    }

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

Even less traffic at lock handover!

---

Queue-based lock: CLH-lock -- a variation of the MCS lock

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

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

```
proc unlock(int **I, **P) {
    **I =0; /* release the lock */
    **P =*P; } /* next time *I to reuse the previous guys flag*/
```

---

CLH lock

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

In CS

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

---
proc 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 */

proc 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 ....

proc unlock(int **I, **P) {
    **I = 0;
    *I = *P;
}

In CS

E6800 locks 12 CPUs

NUMA: Non-uniform Comm Arch.

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

E6800 locks (excluding POUND)

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

Introducing RH locks

Benchmark:

for i = 1 to 10000 {
    lock(AL)
    A:= A + 1;
    unlock(AL)
}
**RH locks: encourages unfairness**

- **Time per lock handover**
- **Node migration (%)**

**Ex: Splash Raytrace Application Speedup**

- **HBO@HPCA 2003**
- **RH@SC 2002**

**Performance under contention**

- **TestAndSet**
- **ExpBackoff**
- **Queuebased**
- **RH**

**A Centralized Barrier**

```
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);
    if (bar_name.counter == p) { /* everybody here yet? */
        bar_name.flag = local_sense; /* release waiters */
        bar_name.counter++;
        UNLOCK(bar_name.lock);
    } else { /* wait for the last guy */
        while (bar_name.flag != local_sense) {
            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

**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
New kind of synchronization: Transactional Memory

- OLD: lock(ID); unlock(ID) around critical sections
- NEW: start_transaction; end_transaction around "critical sections" (note: no ID!!)
- Underlying mechanism to guarantee atomic behavior
- This is not the same as guaranteeing that only one thread is in the critical action!!
- Supported in HW (soon) or in SW (normally very inefficient)
- Suggested by Maurice Herlihy in 1993
- HW support announced by Sun and Intel (??)

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

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
**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, ...

- Vectory Processors
  - LD/ST operate on vectors of data
  - ALU Ops operate on vectors of data
- Example:
  - 8 "vector register" contains 64 vector entries 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**

- **Message-passing**
  - **Fine-grained**
  - **Coarse-grained**
- **Shared Memory**
  - **UMA**
  - **NUMA**
  - **COMA**

**Message-passing Arch MIMD**

Explicit Messages

Program...
Program...
Program...
**Message-Passing HW**

- Programmed in MPI or PVM (or HPF Fortran...)
- 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, but 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:**

```
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 Elektornik
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

The server market 1995

<table>
<thead>
<tr>
<th>Server Size</th>
<th>High-Perf Computing</th>
<th>Commercial Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;$10k</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>&lt;$50k</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>&lt;$250k</td>
<td>5%</td>
<td>2%</td>
</tr>
<tr>
<td>&lt;$1M</td>
<td>2%</td>
<td>3%</td>
</tr>
<tr>
<td>&gt;$1M</td>
<td>3%</td>
<td>3%</td>
</tr>
</tbody>
</table>

The target of the rocket science supercomputers