Architectures

A high level exploration of the HPC world
What kind of parallel architecture

– Shared memory or distributed memory
– What kind of network? Which topology?
– What tools can we use?
– How the user level programs interact with the hardware?
Asynchronous vs. synchronous

- Allow overlapping communication computation
- Hiding latencies
- Additional cost for management

- No overlapping
- No additional cost
- All latencies included in the final time
Architectures

• Vector architecture
• Multi flow architectures
• Shared memory
• Distributed memory
Vector architecture

• Specialized on computation on arrays
• One instruction can be applied to several data from the same arrays (loops)
• Load/Store through vector registers

For $i = 0$ to 64 do
  $a[i] = c[i] + d[i]$
  $b[i] = a[i] * f[i]$

Correct sequential semantic: $a[0]$, $b[0]$, $a[1]$, $b[1]$, …
Vector architecture

- Specialized on computation on arrays
- One instruction can be applied to several data from the same arrays (loops)
- Load/Store through vector registers

For $i = 0$ to $64$ do
\[ a[i] = c[i] + d[i] \]
\[ b[i] = a[i] \times f[i] \]

$a[0], a[1], a[2], \ldots, a[63], b[0], b[1], b[64]$
Vector architecture

- Vector operation = pipeline
- Operation applied directly on the vector registers
- Instruction with strong semantics: one instruction applied on the whole vector register
Vector architecture - example

For $i = 0$ to $64$ do
\[ c[i] = a[i] + b[i] \]

LoadV A, RV1
LoadV B, RV2
AddV RV1, RV2, RV0
StoreV C, RV0

Cost:
- sequential 64 * 4 cycles
- Vector 63 + 4 cycles
Vector architecture - example

• Total cost of the vectorized loop:

\[ T_{\text{init}} + 63 \quad (\text{LoadV A || LoadV B}) \]
\[ + 4 + 63 \quad (\text{AddV}) \]
\[ + T_{\text{init}} + 63 \quad (\text{StoreV}) \]
\[ = 2 \times T_{\text{init}} + 193 \text{ ticks} \]

• Chaining = linking the pipelines together

\[ T_{\text{init}} + 4 + T_{\text{init}} + 63 = 2 \times T_{\text{init}} + 67 \]

• How about the memory access?
Vector architecture

• Several bancs to sustain the high bandwidth
• Components “state of the art” from the technology point of view
• First vector processor: Cray1 (12 vector units + chain MAC)
• Vector multiprocessor: CrayT90 32 procs (1024 memory bancs)
• Vendors: SGI/Cray, Fujitsu, NEC, Hitachi
Multiflow architecture

• Hyper-Threading it’s a new idea?
• Basic idea: **do something else while waiting for memory latency or how to deal with cache misses and data dependencies**

• When to switch?
  – On every load operation
  – On cache miss
  – On every instruction (no cache locality)
  – On instruction block

• How to switch?
  – Context switch too expensive: thread approach
Multi flow architectures

Super scalar

Alternate flows

Simultaneous flows

Shared resources
Multiflow architecture

• TERA MultiThreaded Architecture
  – Heavily alternate multi threaded
  – No caches (direct access to the memory)
  – Change the flow after each load
  – One memory access ~ 100 cycles
  – 16 protection domains (register, status, CP) sharing 128 threads by processor ...

• Up to 256 processors !!!
Shared Memory

- Each processor has its own cache (one or several levels)
- They can access the whole shared memory

- How about consistency? How can a data be on several processors at the same time.
Shared memory

- Allow fine grain resources sharing
- Communications are implicit in load/store on shared addresses
- Synchronization is performed by operations on shared addresses
Shared memory - ShMem

Uniform Memory Access
Non Uniform Memory Access
Cache Coherent – Non Uniform Memory Access
Cache Only Memory Access
ShMem – Shared Cache

• Alliant FX-8 (8x68020 512KB); Encore & Sequent (2xN32032)

• Advantages
  – Identical to uni processor systems
  – Only one copy of any cached block
    • Smaller storage size
  – Fine-grain sharing
  – Potential for positive interference
    • One proc prefetch data for another
  – Can share data within a line without “ping-pong”
  – No false sharing for long data
ShMem – Shared Cache

• Drawbacks
  – Sharing cache bandwidth between processors
  – Increase latencies for ALL accesses
  – Potential for negative interference
    • One proc flush data for another

Many L2 caches are shared today
ShMem – Bus based approach

- Cheap, usual components => dominate the market
- Attractive as servers and convenience parallel computers
  - Fine grain resource sharing
  - Uniform access using Load/Store
  - Automatic data movement and coherent cache replication
  - Cheap and powerful extension

- **Sequential access**

Normal uni processor mechanism to access data
ShMem - caches

• Caches become critical
  – Reduce average latency (replication closer to proc)
  – Reduce average bandwidth
  – Manage consistency

• Data goes from producer to consumer to memory
• Many processors can share the memory efficiently
• Concomitant read accesses to the same location
ShMem – cache coherence example

A = 1

Shared Memory

A = 1

L2

L2

L2
ShMem – cache coherence example

A = 1

Shared Memory

A = 1

L2

A = 1

L2

L2
ShMem – cache coherence example

Processors have different values for A
ShMem – cache coherence example

Processors have different values for A
Write back caches depend on the happenstance of which caches flushes (?)

Intolerable from the programmer point of view
ShMeme – cache coherence example2

Shared Memory

A = 1
L2
A = 0
L2
L2
A = 10

Still intolerable ....
ShMem – Caches and coherence

• Caches play an important role in all cases
  – Reduce average access time
  – Reduce bandwidth on shared interconnection

• Private caches create a coherence problem
  – Copies of the same data on several caches
  – A write may not become visible to other processors

• Solutions
  – Another memory organization
  – Detect and take actions to avoid this problem
ShMem – Cache coherence protocols

• 2 main categories:
  – Invalidation
    • Any write preceded by a block invalidation for all others processors
  – Broadcast (diffusion)
    • Before any write all caches containing the same data will be invalidated
ShMem – Snoop protocols

• Snooping (or monitoring) the bus
• set of states
• state-transition diagram
• actions

Cache line representation

<table>
<thead>
<tr>
<th>state</th>
<th>TAG</th>
<th>data</th>
</tr>
</thead>
</table>
Ordering (memory consistency)

- What’s the intuition?
- Whatever it is, we need an ordering model for clear semantics
  - across different locations as well
  - so programmers can reason about what results are possible

Lamport give the definition of a multi processor sequentially consistent:
- The result of all executions is the same as the sequential atomic execution of each instruction
- The operations of each processor appear in the sequential order as specified by the program.

```plaintext
/*Assume initial values of A and B are 0 */

(P1)
(1a) A = 1;
(1b) B = 2;

(P2)
(2a) print B;
(2b) print A;
```
Ordering (memory consistency)

• What matters is order in which operations *appear to execute*, not the chronological order of events
• Possible outcomes for (A,B): (0,0), (1,0), (1,2)
• What about (0,2)?
  – program order => 1a->1b and 2a->2b
  – A = 0 implies 2b->1a, which implies 2a->1b
  – B = 2 implies 1b->2a, which leads to a contradiction
• What is actual execution 1b->1a->2b->2a?
  – appears just like 1a->1b->2a->2b as visible from results
  – actual execution 1b->2a->2b->1a is not
ShMem – Cache & Directories

• **Centralized**
  – Keep the state and the tag of each block of data for all caches
  – For each memory access the controller check the tag and state of all blocks

• **Distributed**
  – Each processor keep a directory for the data in his cache
  – Update this data depending on the information on the bus.

• **Strongly depend on the interconnection network.** (broadcast)
POSIX Threads & RPC: 2 parallel programming models

George Bosilca
bosilca@cs.utk.edu
Process vs. Thread

- A process is a collection of virtual memory space, code, data, and system resources.
- A thread (lightweight process) is code that is to be serially executed within a process.
- A process can have several threads.

Threads executing the same block of code maintain separate stacks. Each thread in a process shares that process's global variables and resources.

Possible to create more efficient applications?
Process vs. Thread

- Multithreaded applications must avoid two threading problems: deadlocks and races.
- A deadlock occurs when each thread is waiting for the other to do something.
- A race condition occurs when one thread finishes before another on which it depends, causing the former to use a bogus value because the latter has not yet supplied a valid one.
The key is synchronization

- Synchronization = gaining access to a shared resource.
- Synchronization REQUIRE cooperation.
POSIX Thread

• What’s POSIX?
  – Widely used UNIX specification
  – Most of the UNIX flavor operating systems

POSIX is the Portable Operating System Interface, the open operating interface standard accepted world-wide. It is produced by IEEE and recognized by ISO and ANSI.
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive
Mutual exclusion

• Simple lock primitive with 2 states: lock and unlock
• Only one thread can lock the mutex.
• Several politics: FIFO, random, recursive
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive
Mutual exclusion

• Simple lock primitive with 2 states: lock and unlock
• Only one thread can lock the mutex.
• Several politics: FIFO, random, recursive
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive
Mutual exclusion

- Simple lock primitive with 2 states: lock and unlock
- Only one thread can lock the mutex.
- Several politics: FIFO, random, recursive
Mutual exclusion

• Spin vs. sleep?
• What’ s the desired lock grain?
  – Fine grain – spin mutex
  – Coarse grain – sleep mutex
• Spin mutex: use CPU cycles and increase the memory bandwidth, but when the mutex is unlock the thread continue his execution immediately.
Shared/Exclusive Locks

- **ReadWrite Mutual exclusion**
- Extension used by the reader/writer model
- 4 states: write_lock, write_unlock, read_lock and read_unlock.
- multiple threads may hold a shared lock simultaneously, but only one thread may hold an exclusive lock.
- if one thread holds an exclusive lock, no threads may hold a shared lock.
**Shared/Exclusive Locks**

Legend
- **Active thread**
- **Sleeping thread**

Step 1

- **Writer 1**
  - rw_lock
  - rw_unlock

- **Writer 2**
  - rw_lock
  - rw_unlock

- **Reader 1**
  - rd_lock
  - rd_unlock

- **Reader 2**
  - rd_lock
  - rd_unlock

Step 2

- **Writer 1**
  - rw_lock
  - rw_unlock

- **Writer 2**
  - rw_lock
  - rw_unlock

- **Reader 1**
  - rd_lock
  - rd_unlock

- **Reader 2**
  - rd_lock
  - rd_unlock
Shared/Exclusive Locks

Legend
- Active thread
- Sleeping thread

Writer 1
... 
**rw_lock**
**rw_unlock**
...

Writer 2
... 
**rw_lock**
**rw_unlock**
...

Reader 1
... 
**rd_lock**
**rd_unlock**
...

Reader 2
... 
**rd_lock**
**rd_unlock**
...

Step 3

Step 4
Shared/Exclusive Locks

Legend
- Active thread
- Sleeping thread

Writer 1
- `rw_lock`
- `rw_unlock`

Writer 2
- `rw_lock`
- `rw_unlock`

Reader 1
- `rd_lock`
- `rd_unlock`

Reader 2
- `rd_lock`
- `rd_unlock`

Step 5

Writer 1
- `rw_lock`
- `rw_unlock`

Writer 2
- `rw_lock`
- `rw_unlock`

Reader 1
- `rd_lock`
- `rd_unlock`

Reader 2
- `rd_lock`
- `rd_unlock`

Step 6
Condition Variable

- Block a thread while waiting for a condition
- `Condition_wait / condition_signal`
- Several thread can wait for the same condition, they all get the signal

```
Thread 1
... | ... | ...
signal | wait | ...
... | ... | ...
Thread 2
... | ... | ...
wait | wait | ...
... | ... | ...
Thread 3
... | ... | ...
wait | wait | ...
... | ... | ...
```

Active threads

Sleeping threads
Condition Variable

- Block a thread while waiting for a condition
- Condition_wait / condition_signal
- Several threads can wait for the same condition, they all get the signal
Condition Variable

- Block a thread while waiting for a condition
- `Condition_wait / condition_signal`
- Several threads can wait for the same condition, they all get the signal

![Diagram showing active and sleeping threads with conditions and signals.](image)
Condition Variable

- Block a thread while waiting for a condition
- `Condition_wait / condition_signal`
- Several threads can wait for the same condition, they all get the signal

Active threads

Thread 1

sleep

Thread 2

wait

Thread 3

wait

Sleeping threads
Condition Variable

- Block a thread while waiting for a condition
- Condition_wait / condition_signal
- Several thread can wait for the same condition, they all get the signal
Semaphores

- simple counting mutexes
- The semaphore can be hold by as many threads as the initial value of the semaphore.
- When a thread get the semaphore it decrease the internal value by 1.
- When a thread release the semaphore it increase the internal value by 1.
Semaphores

Thread 1
...
get
release
...

Thread 2
...
get
release
...

Thread 3
...
get
release
...

Semaphore (2)

Semaphore (1)

Thread 1
...
get
release
...

Thread 2
...
get
release
...

Thread 3
...
get
release
...
Semaphores

Thread 1

...  

get

release

...

Thread 2

...  

get

release

...

Thread 3

...  

get

release

...

Semaphore (0)

Semaphore (0)
Semaphores

Thread 1
...
get
release
...

Thread 2
...
get
release
...

Thread 3
...
get
release
...

Semaphore (1)

Thread 1
...
get
release
...

Thread 2
...
get
release
...

Thread 3
...
get
release
...

Semaphore (1)
Atomic instruction

- Is any operation that a CPU can perform such that all results will be made visible to each CPU at the same time and whose operation is safe from interference by other CPUs
  - TestAndSet
  - CompareAndSwap
  - DoubleCompareAndSwap
  - Atomic increment
  - Atomic decrement