Parallel computing models and their performances

A high level exploration of the parallel computing world

George Bosilca
bosilca@icl.utk.edu
Overview

- Definition of parallel application
- Architectures taxonomy
- What is quantifiable? Laws managing the parallel applications field
- Modeling performance of parallel applications
Formal definition of parallelism

The **Bernstein Conditions** Let’s define:

- I(P) all variables, registers and memory locations used by P
- O(P) all variables, registers and memory locations written by P

Then P1; P2 is equivalent to P1 || P2 if and only if

\[
\{ \text{I}(P1) \cap \text{O}(P2) = \emptyset \land \text{I}(P2) \cap \text{O}(P1) = \emptyset \land \text{O}(P1) \cap \text{O}(P2) = \emptyset \}\]

**General case:** P1… Pn are parallel if and only if each for each pair Pi, Pj we have Pi || Pj.

3 limit to the parallel applications:

1. Data dependencies
2. Flow dependencies
3. Resources dependencies

\[\begin{array}{c}
\text{I1} \\
\downarrow \\
P1 \\
\downarrow \\
\text{O1}
\end{array}
\quad
\begin{array}{c}
\text{I2} \\
\downarrow \\
P2 \\
\downarrow \\
\text{O2}
\end{array}\]
Data dependencies

I1:   A = B + C
I2:   E = D + A
I3:   A = F + G

— Flow dependency (RAW): a variable assigned in a statement is used in a later statement
— Anti-dependency (WAR): a variable used in a statement is assigned in a subsequent statement
— Output dependency (WAW): a variable assigned in a statement is subsequently re-assigned

How to avoid them?
Which type of data dependency can be avoided?
Flow dependencies

I1: \( A = B + C \)
I2: if( A ) {
I3: \( D = E + F \) }
I4: \( G = D + H \)

— Data dependency
— Control dependency

How to avoid?
Resources dependencies

I1: \[ A = B + C \]
I2: \[ G = D + H \]

How to avoid?
A more complicated example (loop)

for $i = 0$ to $9$
\[ A[i] = B[i] \]

All statements are independent, as they relate to different data. They are concurrent.

for $i = 1$ to $9$
\[ A[i] = A[i-1] \]


All statements are dependent, as every 2 statements are strictly sequential.
Flynn Taxonomy (1966)

• Computers classified by instruction delivery mechanism and data stream(s)
  • I for instruction, P for program. Conceptually similar, technically at a different granularity.
• 4 characters code: 2 for instruction stream and 2 for data stream

<table>
<thead>
<tr>
<th></th>
<th>1 Instruction flow</th>
<th>&gt; 1 Instruction flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 data stream</td>
<td>SISD</td>
<td>MISD pipeline</td>
</tr>
<tr>
<td></td>
<td>Von Neumann</td>
<td></td>
</tr>
<tr>
<td>&gt; 1 data stream</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>
Flynn Taxonomy: Analogy

- SISD: assembly line work (no parallelism)
- SIMD: systolic, GPU computing (vector computing MMX, SSE, AVX)
- MISD: more unusual type. Safety requirements, replication capabilities, think space shuttle.
- MIMD: airport facility, several desks working at their own pace, synchronizing via a central entity (database). Most distributed algorithms, as well as multi-core applications.
Amdahl Law

- First law of parallel applications (1967)
- Limit the speedup for all parallel applications

\[ speedup = \frac{s + p}{s + \frac{p}{N}} \]

\[ speedup = \frac{1}{a + \frac{(1 - a)}{N}} \]

N = number of processors

s = sequential part
p = parallel part
Amdahl Law

Speedup is bound by $1/a$. 

**FIGURE 1.** Speedup under Amdahl’s Law
Amdahl Law

• Bad news for parallel applications

• 2 interesting facts:
  • We should limit the sequential part
  • A parallel computer should be a fast sequential computer to be able to resolve the sequential part quickly

• What about increasing the size of the initial problem?
Gustafson’s Law

• Less constraints than the Amdahl law.
• In a parallel program the quantity of data to be processed increase, so compared with the parallel part the sequential part decrease.

\[
t = s + \frac{P}{n} \\
P = an \\
\begin{align*}
\text{speedup} &= \frac{s + an}{s + a} \\
\end{align*}
\]

\[a \to \infty \Rightarrow \text{speedup} \to n\]
Gustafson’s Law

• The limit of Amdahl Law can be transgressed if the quantity of data to be processed increase.

\[ speedup \leq n + (1 - n)s \]

Rule stating that if the size of most problems is scaled up sufficiently, then any required efficiency can be achieved on any number of processors.
Speedup

• Superlinear speedup?

Sometimes superlinear speedups can be observed!

• Memory/cache effects
• More processors typically also provide more memory/cache.
• Total computation time decreases due to more page/cache hits.

• Search anomalies
  • Parallel search algorithms.
  • Decomposition of search range and/or multiple search strategies.
• One task may be "lucky" to find result early.
Parallel execution models

- Amdahl and Gustafson laws define the limits without taking in account the properties of the computer architecture
- They can only loosely be used to predict (in fact mainly to cap) the real performance of any parallel application
- We should integrate in the same model the architecture of the computer and the architecture of the application
What are models good for?

• Abstracting the computer properties
  • Making programming simple
  • Making programs portable?

• Reflecting essential properties
  • Functionality
  • Costs

• What is the von-Neumann model for parallel architectures?
Parallel Random Access Machine

- World described as a collection of synchronous processors which communicate with a global shared memory unit.
  - A collection of numbered RAM processors ($P_i$)
  - A collection of indexed memory cells ($M[i]$)
  - Each processor $P_i$ has its own unbounded local memory (registers) and knows its index (rank)
  - Each processor can access any shared memory cell in unit time
  - Input and output of a PRAM algorithm consist in $N$ distinct items
  - A PRAM instruction consists of 3 synchronous steps: read (acquire the input data), computation, write (save the data back to a shared memory cell).
  - Exchanging data is realized through the writing and reading of memory cells
Parallel Random Access Machine

- Algorithmic Complexity:
  - Time = the time elapsed for $P_0$ computations
  - Space = the number of memory cells accessed

- Specialized in parallel algorithms
  - Natural: the number of operations per cycle on $N$ processors is at most $N$
  - Strong: all accesses are realized in a single time unit
  - Simple: keep the complexity and correctness overheads low by abstracting all communication or synchronization overheads

The PRAM corresponds intuitively to the programmers' view of a parallel computer: it ignores lower level architectural constraints, and details, such as memory access contention and overhead, synchronization overhead, interconnection network throughput, connectivity, speed limits and link bandwidths, etc.
Bulk Synchronous Parallel – BSP

Valiant 1990

• Differs from PRAM by taking in account communications and synchronizations and by distributing the memory across participants
  • Compute: Components capable of computing or executing local memory transactions
  • Communication: A network routing messages between components
  • Synchronization: A support for synchronization on all or a sub-group of components

• Each processor can access his own memory without overhead and have a uniform slow access to remote memory
BSP - Superstep

- Applications composed by Supersteps separated by global synchronizations.

- A superstep contains:
  - A computation step
  - A communication step
  - A synchronization step

Synchronization used to insure that all processors complete the computation + communication steps in the same amount of time. As communications are remote memory accesses (one sided) there are no synchronizations during the computation + communication step.
BSP – Global View
BSP – The communication step

- BSP consider communication not at the level of individual actions, but as a whole (per step)
- The goal being to define an upper bound on the time necessary to complete all data movements

- \( h = \) the maximum number of messages (incoming or outgoing) per superstep

- \( g = \) the network capability to deliver messages
  - It takes \( hg \) time for a processor to deliver \( h \) messages of size 1
  - A message of size \( m \) takes the same time to send as \( m \) messages of size 1
BSP – The synchronization cost

• The cost of synchronization is noted by \( l \) and is generally determined empirically

• With the increase in scale of the computing resources, the synchronizations are becoming the main bottleneck
  • Removing them might introduce deadlock or livelock
  • Decrease the simplicity of the model
BSP – Compute the cost

\[ T_{\text{superstep}} = \max_{i=1}^{p}(w_i) + g \cdot \max_{i=1}^{p}(h_i) + l \]
\[ = w + g \cdot h + l \]

Where:

- \(w\) = max of computation time
- \(g\) = 1/(network bandwidth)
- \(h\) = max of number of messages
- \(l\) = time for the synchronization

\[ T_{\text{total}} = \sum_{s=1}^{S} T_{\text{superstep}} \]
• An algorithm can be described using only $w$, $h$ and the problem size.

• Collections of algorithms are available depending on the computer characteristics.
  • Small L
  • Small g

• The best algorithm can be selected depending on the computer properties.
• Numerical solution to Laplace’s equation

\[ U_{i,j}^{n+1} = \frac{1}{4} \left( U_{i-1,j}^n + U_{i+1,j}^n + U_{i,j-1}^n + U_{i,j+1}^n \right) \]

for j = 1 to jmax
for i = 1 to imax
   \[ U_{\text{new}}(i,j) = 0.25 \times ( U(i-1,j) + U(i+1,j) \]
   \[ + \ U(i,j-1) + U(i,j+1) ) \]
end for
end for
BSP - example

• The approach to make it parallel is by partitioning the data
• The approach to make it parallel is by partitioning the data

Overlapping the data boundaries allow computation without communication for each superstep.

On the communication step each processor update the corresponding columns on the remote processors.
BSP - example

for j = 1 to jmax
    for i = 1 to imax
        unew(i,j) = 0.25 * ( U(i-1,j) + U(i+1,j) 
                             +   U(i,j-1) + U(i,j+1))
    end for
end for
if me not 0 then
    bsp_put( to the left )
endif
if me not NPROCS - 1 then
    bsp_put( to the right )
Endif
bsp_sync()
BSP - example

\[ T_{superstep} = w + g \times h + l \]

\( h = \text{max number of messages} \)

\( = N \text{ values to the left} + \)

\( N \text{ values to the right} \)

\( = 2 \times N \text{ (ignoring the inverse communication!)} \)

\( w = 4 \times N \times N / p \)

\[ T_{superstep} = 4 \times \frac{N^2}{p} + 2 \times g \times N + l \]
**BSP - example**

- BSP parameters for a wide variety of architectures has been published.

<table>
<thead>
<tr>
<th>Machine</th>
<th>s</th>
<th>p</th>
<th>l</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin 2000</td>
<td>101</td>
<td>4</td>
<td>1789</td>
<td>10.24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td>39057</td>
<td>66.7</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>46.7</td>
<td>4</td>
<td>357</td>
<td>1.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>751</td>
<td>1.66</td>
</tr>
<tr>
<td>Pentium 10Mbit</td>
<td>61</td>
<td>4</td>
<td>139981</td>
<td>1128.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>826054</td>
<td>2436.3</td>
</tr>
<tr>
<td>Pentium II 100Mbit</td>
<td>88</td>
<td>4</td>
<td>27583</td>
<td>39.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>38788</td>
<td>38.7</td>
</tr>
</tbody>
</table>
A more sophisticated model LogP

- More empirical, with a focus on network-related capabilities
  - L – latency of communications
  - o – message overhead
  - g – gap between messages
  - P – number of processors
LogP

• Decompose the communications in 3 elements:
  • **Latency**: small message cross the network
    • Dependent on the physical properties of the underlying network
  • **Overhead**: lost time in communication
    • Depends on the OS. Possibly different on the sender and receiver side.
LogP

- Decompose the communications in 3 elements:
  - **Latency**: small message cross the network
    - Dependent on the physical properties of the underlying network
  - **Overhead**: lost time in communication
    - Depends on the OS. Possibly different on the sender and receiver side.
  - **Gap**: between 2 consecutive messages

- And P the number of processors.

Both $g$ and $o$ matter!

$g > o$

$g < o$
LogP

• The total time for a message to go from the processor A to the processor B is:
  \[ L + 2 \times o \]

• There is no model for the application

• We can describe the application using the same approach as for BSP: supersteps

\[
T_{\text{superstep}} = w + h \times (L + 2o) + l
\]
LogP

• The P parameter does not interfere in the superstep computation?

• When the number of processors is not fixed:
  • The time of the computation change $w(p)$
  • The number of messages change $h(p)$
  • The synchronization time change $l(p)$
LogP

• Allow/encourage the usage of general techniques of designing algorithms for distributed memory machines: exploiting locality, reducing communication complexity and overlapping communication and computation.

• Balanced communication to avoid overloading the processors.
LogP

• Interesting concept: idea of finite capacity of the network. Any attempt to transit more than a certain amount of data will stall the processor.

\[
\begin{bmatrix}
L \\
g
\end{bmatrix}
\]

• This model does not address on the issue of message size, even the worst is the assumption of all messages are of ``small'' size.

• Does not address the global capacity of the network.
Design a LogP program

- Execution time is the time of the slowest process

- Implications for algorithms:
  - Balance computation
  - Balance communications
  Are only sub-goals!

- Remember the capacity constraint $\left\lfloor \frac{L}{g} \right\rfloor$
Analyze an algorithm using LogP

Figure 3: Optimal broadcast tree for $P = 8$, $L = 6$, $g = 4$, $o = 2$ (left) and the activity of each processor over time (right). The number shown for each node is the time at which it has received the datum and can begin sending it on. The last value is received at time 24.
Analyze an algorithm using LogP

Figure 3: Optimal broadcast tree for $P = 8$, $L = 6$, $g = 4$, $o = 2$ (left) and the activity of each processor over time (right). The number shown for each node is the time at which it has received the datum and can begin sending it on. The last value is received at time 24.
## LogP Machines

<table>
<thead>
<tr>
<th>Machine</th>
<th>L</th>
<th>$a$</th>
<th>$g$</th>
<th>$P$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM-5</td>
<td>6</td>
<td>2.2</td>
<td>4</td>
<td>512</td>
</tr>
<tr>
<td>Meiko CS-2</td>
<td>8.6</td>
<td>1.7</td>
<td>$14.2 + 0.03x$</td>
<td>64</td>
</tr>
<tr>
<td>Power Xplorer</td>
<td>21 - 0.82$x$</td>
<td>70 + $x$</td>
<td>$115 + 1.43x$</td>
<td>8</td>
</tr>
<tr>
<td>Para-Station</td>
<td>50 - 0.10$x$</td>
<td>3 + 0.112$x$</td>
<td>3 + 0.119$x$</td>
<td>4</td>
</tr>
<tr>
<td>IBM SP-2</td>
<td>13 - 0.005$x$</td>
<td>8 + 0.008$x$</td>
<td>10 + 0.01$x$</td>
<td>128</td>
</tr>
<tr>
<td>IBM SP-2</td>
<td>17 - 0.005$x$</td>
<td>8 + 0.008$x$</td>
<td>10 + 0.01$x$</td>
<td>256</td>
</tr>
</tbody>
</table>
Improving LogP

• First model to break the synchrony of parallel execution
• LogGP: augments the LogP model with a linear model for long messages
• LogGPC model extends the LogGP model to include contention analysis using queuing model on the $k$-ary $n$-cubes network
• LogPQ model augments the LogP model on the stalling issue of the network constraint by adding buffer queues in the communication lines.
The CCM model

- Collective Computing Model transform the BSP superstep framework to support high-level programming models as MPI and PVM.
- Remove the requirement of global synchronization between supersteps, but combines the message exchanges and synchronization properties into the execution of a collective communication.
- Prediction quality usually high.
How to represent the architecture

• 2 resources have a major impact on the performances:
  • The couple (processor, memory)
  • The communication network.

• The application should be described using those 2 resources.

\[ T_{\text{app}} = T_{\text{comp}} + T_{\text{comm}} \]
Models

- 2 models are often used.
- They represent the whole system as composed by \( n \) identical processors, each of them having his own memory.
- They are interconnected with a predictable network.
- They can realize synchronizations.
Architectures

A high level exploration of the parallel computing 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
  $a[i] = c[i] + d[i]$
  $b[i] = a[i] \times f[i]$

Correct sequential semantic: $a[0], b[0], a[1], b[1], \ldots$
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
- $a[i] = c[i] + d[i]$ $\quad$ for $i = 0$ to $64$ do $a[i] = c[i] + d[i]$
- $b[i] = a[i] * f[i]$ $\quad$ For $I = 0$ to $64$ do $b[i] = a[i] * f[i]$

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

\[
T_{\text{init}} + 63 \quad (\text{LoadV A} \parallel \text{LoadV B})
\]

\[
+ 4 + 63 \quad (\text{AddV})
\]

\[
+ T_{\text{init}} + 63 \quad (\text{StoreV})
\]

\[
= 2 * T_{\text{init}} + 193 \text{ ticks}
\]

• Chaining = linking the pipelines together

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

• 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 in 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
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 ex.

A = 1  
Shared Memory

A = 1

L2

L2

L2
ShMem – cache coherence ex.
ShMem – cache coherence ex.

Processors have different values for A
ShMem – cache coherence ex.

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

Intolerable from the programmer point of view
ShMem – cache coherence ex2

A = 1

Shared Memory

L2
A = 0

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

<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.

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

(1a) A = 1;   (2a) print B;
(1b) B = 2;   (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: a first step toward parallel programming

George Bosilca
bosilca@icl.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 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

- 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

Step 3

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 4

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

Step 5

Step 6
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 7
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
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

Active threads
- Thread 1
  - ... signal ...
  - ...
- Thread 2
  - ... wait ...
  - ...
- Thread 3
  - ... wait ...
  - ...
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  Thread 2  Thread 3

```
  ...  
  signal ...
  ...

  ...  
  wait ...
  ...

  ...  
  wait ...
  ...
```
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 threads and wait states]
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

...  

wait

signal

...  

Thread 2

...  

wait

...  

Thread 3

...  

wait

...
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

...

Semaphore (2)

Thread 2

... 

get

release

...

Semaphore (1)

Thread 3

... 

get

release

...

Semaphore (2)

Semaphore (1)
Semaphores

Thread 1
...
get
release
...

Semaphore (0)

Thread 2
...
get
release
...

Thread 3
...
get
release
...

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)
Semaphores

Thread 1

... get
release ...

Thread 2

... get
release ...

Thread 3

... get
release ...

Semaphore (1)

Semaphore (2)
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