Shared memory and Message passing revisited in the many-core era

Aram Santogidis
CERN

Inverted CERN School of Computing, 29 February – 2 March 2016
The pioneers of concurrent programming

**Edsger Dijkstra**
- Mutual exclusion
- Cooperating Sequential Processes
- Semaphores

**Per Brinch Hansen**
- Concurrent Pascal
- Shared Classes
- The Solo OS
- Distributed Processes

**C.A.R Hoare**
- Communicating Sequential Processes (CSP)
- Monitors
Communication is important

Process 1

Process 2

Process 3

Time

Communication/Synchronization

Shared memory

VS

Message passing
Agenda of the talk

- Concurrency and communication
- Two basic examples of the two models
- Conventional wisdom for the two models
- Cache coherence and manycore processors
- Emerging paradigm shift in OS architectures
- The future perspective
The Shared Memory model

- Threads communicate **implicitly** with each other via shared data structures

- Synchronization primitives (locks, semaphores, etc.)
The message passing model

- Threads communicate **explicitly** with each other by exchanging messages
- Is the more **fundamental** class from the two
- Synchronous or asynchronous communication
Lets see an example for each model

1. Image processing (shared memory)
2. Simple GUI (message passing)
A shared memory-based example: Convert from colour to grayscale

\[
\begin{align*}
R &: 204 \\
G &: 46 \\
B &: 10 \\
\end{align*}
\]

\[
\frac{(R+G+B)}{3} = 130
\]
A shared memory-based example: Convert from colour to grayscale

We parallelize the computation by assigning tiles (pieces) of the image to threads which execute the conversion in parallel.
A message passing-based example

- A GUI with 3 widgets
  - Text Area
  - Up scroll button
  - Down scroll button

- Must be interactive (Immediate feedback)

Many operating system designs can be placed into one of two very rough categories, depending upon how they…
GUI example implementation: Message passing solution

Many operating system designs can be placed into one of two very rough categories, depending upon how they...
Conventional wisdom about the characteristics of the two models

1. Performance

2. Programmability
## Performance comparison

<table>
<thead>
<tr>
<th></th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware support</strong></td>
<td><em>Extensive</em> (All popular architectures)</td>
<td><em>Limited</em> (Only special purpose architectures)</td>
</tr>
<tr>
<td><strong>Data transfer Overhead</strong></td>
<td><em>Low</em> (Cache block management in HW)</td>
<td><em>High</em> (Data replication)</td>
</tr>
<tr>
<td><strong>Access/Sync overhead</strong></td>
<td><em>Sometimes high</em> (Critical section contention, NUMA effects)</td>
<td><em>Low</em> (Local private memory access)</td>
</tr>
</tbody>
</table>
# Programmability comparison

<table>
<thead>
<tr>
<th></th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Communication</strong></td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td><strong>Synchronization</strong></td>
<td>Explicit (locks etc.)</td>
<td>Implicit (side-effect)</td>
</tr>
<tr>
<td><strong>Interface (API)</strong></td>
<td>Read/write shared data structures, mutex primitives</td>
<td>Send/Receive messages, Multicast</td>
</tr>
<tr>
<td><strong>Hazards</strong></td>
<td>Race conditions, Deadlocks, Starvation</td>
<td>Deadlocks, Starvation</td>
</tr>
</tbody>
</table>
Towards the manycore architectures

http://www.wired.com/images_blogs/gadgetlab/2009/10/tilera-wafer-1.jpg
The manycore era

Manycore systems design space

http://image.slideserve.com/277797/manycore-systems-design-space-n.jpg

- Power limits the frequency increase of the processor.
- Moore’s law: The transistors keep doubling every two years
- Replication: Increasing number of cores


MIT/UCB
On the duality of operating systems structures

- Operating Systems are generally classified as:
  - Message passing oriented
  - Procedure-oriented (shared memory)

- Each system from one category has the other category.

- Neither model is inherently better than the other (depends on the machine architecture).

Cache coherence

j = i;

i: 0

Core 0

L1

Shared

L1

Core 1

Cache Controller

LLC

i: 0

Cache Controller

BUS
Cache coherence

j = i;

Core 0

Cache Controller

L1

Modified

i: 0

Invalid

Core 1

Cache Controller

L1

i: 1

Message

LLC

i: 0

Bus/ Invalidate

BUS
Cache coherence

j = i;

Core 0

L1

i: 1

Shared

L1

i: 1

Core 1

Cache Controller

BUS

Write back

LLC

i: 1

Cache Controller

i++;

Bus Read

j = i;

i: 0

0

1
A key question

When updating shared state, which approach is more expensive (in terms of latency), Shared memory or Message passing?
An experiment of shared memory vs message passing performance

Updating shared state of size [1,8] cachelines, relying on cache coherent shared memory on 4x4 AMD system
An experiment of shared memory vs message passing performance

CPU in Socket → Hyper → Transport → Server

Server, updating the shared state on behalf of the threads

Updating shared state of size [1,8] cachelines, relying on synchronous Lightweight Remote Procedure Calls (message passing)
Messages scale better than shared memory

Message passing scales better than shared memory when increasing the core count and the size of the shared state.

...some other hints that may lead to further fragmentation of coherency domains

Increasing Heterogeneity of computing platforms

- **Message passing**: Fundamental for communication in a heterogeneous environment
- **Shared memory**: Hard to implement in a heterogeneous environment

**Shared memory and message passing revisited in the many-core era**

**Increasing Heterogeneity**

- Time
  - Multi socket
  - Manycores, GPU
  - Coprocessor
  - FPGA
  - Dual cores, pthreads
  - Single cores, Concurrent OS, Coroutines
Message passing OS vs Shared memory OS

**Barrelfish OS** (Message passing)

- OSnode
  - State replica
  - x86
  - ARM
- Async messages
- Architecture dependant code

**Linux OS** (Shared memory)

- Linux Kernel
  - App
  - Driver
  - Single arch(x86, etc.)
  - GPU

What to expect?

Emerging concurrency paradigms

New high level paradigms are being developed, based on shared memory and/or message passing constructs.

- Asynchronous tasks (Futures/Promises)
- Partitioned Global Address Space (PGAS) languages/libraries
- Actor Model
- Functional Concurrency
The future perspective

- Communication is the key
  - For energy efficiency
  - For runtime performance
  - To manage software complexity
  - To manage hardware heterogeneity

- Innovation in the hardware sector pressures to systems software engineers to develop appropriate support
  - At the operating system level
  - Concurrent programming frameworks level
  - Communication-oriented tools and techniques to design, implement, analyse concurrent programs
Shared memory and message passing revisited in the many-core era

http://globe-views.com/dcim/dreams/surprise/surprise-05.jpg
References


- A Primer on Memory Consistency and Cache Coherence Daniel J. Sorin, Mark D. Hill, and David A. Wood


Thank you for your attention

Many thanks to my supporters and mentors for this presentation:

Sebastian Lopienski, Sebastian.Lopienski@cern.ch, CERN
Andreas Joachim Peters, Andreas.Joachim.Peters@cern.ch, CERN
Andreas Hirstius, andreas.hirstius@intel.com, Intel GmbH
Spyros Lalis, lalis@inf.uth.gr, University of Thessaly

This work is support by the Marie Curie Early European Industrial Doctorates Fellowship of the European Community’s Seventh Framework Programme under contract number (PITN-GA-2012-316596-ICE-DIP).