Non uniform memory architectures

Master in computer science of IP Paris
Master CHPS of Paris Saclay
Gaël Thomas
We need computing power

- To analyze large datasets
- To perform large computations
- To handle many clients
The computing power is in the CPU
(Old) computing power trends

Moore’s low: #transistors x2 each 1.5 year

# transistors (thousands)

Single step Perf (SpecINT)

Frequency (MHz)

⇒ processing power

Typical (electrical) power
(Watts)

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Multicore Programming  Non-Uniform Memory Architectures
But frequency increases $\Rightarrow$ electrical power increases

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore
Fortunately, the Moore’s law still hold

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Multicore Programming
Non-Uniform Memory Architectures
Today: we increase power by increasing the number of cores

# transistors (thousands)

Single step Perf (SpecINT)

Frequency (MHz)

Typical power (Watts)

# cores

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore
But programming a multicore is hard

```c
#include <stdlib.h>
#define N 100000000
int main(int argc, char **argv) {
    int* a = malloc(sizeof(int) * N);

    for(int i=1; i<N; i++) {
        a[i] = a[i] * a[i-1];
    }
}
```

On my laptop at 2k€ (2 cores at 2.2GHz)
$ time ./bip
real 0m0.474s

On my server at 15k€ (48 cores at 2.2GHz)
$ time ./bip
real 0m1.142s

Not really what we can expect
Multicores radically change the way we design applications

We have to parallelize our applications
Multicores radically change the way we design applications

- We have to parallelize our applications
- And our parallel algorithms have to scale
Multicores radically change the way we design applications

- We have to parallelize our applications
- And our parallel algorithms have to scale

But that’s not enough…

We have to handle complex memory architectures
But memory access latency varies a lot

Core 0 allocates, core 0 accesses => ~5 cycles
Core 3 allocates, core 0 accesses => ~50 cycles
Core 15 allocates, core 0 accesses => ~275 cycles
Core 20 allocates, core 0 accesses => ~380 cycles

Benchmark : memal on a 48 cores/4 sockets with 128GB (AMD)
We have cache effects

\[
\begin{align*}
& \sim 5 \text{ cycles} \\
& L1 \\
& \sim 15 \text{ cycles} \\
& L2 \\
& \sim 50 \text{ cycles} \\
& L3
\end{align*}
\]
And, since a single bus does not scale...
...we have complex architectures

Total: 64 cores/128 hyperthreads, 256GB
(32x the power of my macbook for 15k€)

Supermicro X11QPH+

4 x Intel Xeon GOLD 6130
16 cores/32 hyperthreads

16 x 8GB
...we have complex architectures
and non uniform memory accesses

Fast access

Slow access

Node

Bus

Core

Interconnect

Multicore Programming

Non-Uniform Memory Architectures
and non uniform memory accesses

On our 48-core AMD with 8 nodes (6 cores per node)
- Local memory access : 155 cycles
- One hop = 275 cycles
- Two hops = 380 cycles x2.5
Memory access latency can collapse

When all the cores access the same node (but different cache lines)

870 cycles (6 times a local access)
On a NUMA architecture, we need memory placement policies

- To avoid the overload of a single NUMA domain
- To avoid the overload of interconnect links
- To enforce memory access locality
HowTo: NUMA placement policy

Step 1: choose the physical address of a data

because the physical address space is partitioned among the domains

```
<table>
<thead>
<tr>
<th>0</th>
<th>n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>RAM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>2n - 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>RAM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2n</th>
<th>3n - 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>RAM</td>
</tr>
</tbody>
</table>
```
HowTo: NUMA placement policy

Step 2: leverage the page table

Maps a virtual address to a specific node by mapping the virtual address to a page that belongs to the node.
// reserve a virtual address space
struct x* x = mmap(0, sizeof(*x), ...);
HowTo: NUMA placement policy

// reserve a virtual address space
struct x* x = mmap(0, sizeof(*x), ...);

// and requires pages from node 2
mbind(x, sizeof(*x), 2);
Main questions

- Does NUMA effect matters in practice?
- If yes, can we mitigate this effect?
In theory, NUMA matters

Abstract cache-unfriendly application

- 50% of the instructions access memory
- 30% of the accesses in L1 cache
- 30% of the accesses in L2 cache
- 30% of the accesses in L3 cache

Comparison between best and worst NUMA placements

- Best: all accesses to local node $\Rightarrow \sim 32$ cycles/insn
- Worst: all accesses to an overloaded node $\Rightarrow \sim 156$ cycles/insn

$\Rightarrow$ overhead of 385% in the worst case
In theory, NUMA matters

Abstract cache-friendly application

- 50% of the instructions access memory
- 70% of the accesses in L1 cache
- 70% of the accesses in L2 cache
- 70% of the accesses in L3 cache

Comparison between best and worst NUMA placements

- Best: all accesses to local node $\Rightarrow \sim 7$ cycles/insn
- Worst: all accesses to an overloaded node $\Rightarrow \sim 17$ cycles/insn

$\Rightarrow$ overhead of 137% in the worst case
First study

Goal:
- Understand how Linux manages NUMA
- Understand how applications react to NUMA

How:
- Study a panel of 29 applications from 5 benchmarks (NPB, Parsec, Mosbench, X-stream, YCSB)
- Evaluate various NUMA management policies
The hand-tuned policy

Manually place the memory address ranges on the nodes

- Memory range 1
- Memory range 2

Virtual address space of a process
The hand-tuned policy

Manually place the memory address ranges on the nodes

+ Tune the memory placement for an application
- A lot of engineering effort for only a single application/hardware
The hand-tuned policy on Linux

Hand-tuned thread placement

- `setaffinity(set of cores)`: for all the threads of a process
- `pthread_setaffinity(set of cores)`: for a single thread

Hand-tuned memory placement

- `mbind(virtual address range, set of nodes)`
  *(granularity of a 4k-page)*
The interleaved policy

Round-robin from all the nodes
The interleaved policy

- Round-robin from all the nodes
  - Balance the load on all the nodes ⇒ no overloaded node
  - Many remote accesses ⇒ interconnect can saturate

![Diagram of virtual address space and page table]

- Physical Address Space
- Page Table
- Virtual address space of a process

Node 0  Node 1  Node 2

Multicore Programming  Non-Uniform Memory Architectures
The first-touch policy

- From the node that triggers the first access
  - Relies on the lazy mapping used in Linux

Thread running on node 1

![Diagram showing physical address space, page table, and virtual address space of a process.](image-url)
The first-touch policy

From the node that triggers the first access

- Relies on the lazy mapping used in Linux

Thread running on node 1

Memory access

Virtual address space of a process

Page Table

Physical Address Space

Node 0  Node 1  Node 2
The first-touch policy

From the node that triggers the first access
- Relies on the lazy mapping used in Linux
The first-touch policy

- From the node that triggers the first access
  + Perfect locality and no saturation if a thread accesses its memory
  - Overloaded nodes if some threads allocate for the others
The Carrefour policy

Proposed by Dashti et al. (ASPLOS’15)
  • Rebalance the load on all the nodes
  • Prevents the contention of the interconnect

Dynamically migrate a page
  • From contended to uncontended nodes in case of contented node
  • On the node that uses the page in case of contended interconnect
The Carrefour policy

- Proposed by Dashti et al. (ASPLOS’15)
  - Rebalance the load on all the nodes
  - Prevents the contention of the interconnect

- Dynamically migrate a page
  - From contended to uncontended nodes in case of contended node
  - On the node that uses the page in case of contended interconnect

+ Improves locality and avoid contention in many cases

- Can lead to inefficient placements for applications with different access patterns during the run
Evaluated policies

Four combinations

- First-touch (Linux FT)
- First-touch with Carrefour (Linux FT/Carrefour)
- Interleaved (Linux 4K)
- Interleaved with Carrefour (Linux 4K/Carrefour)

Only considers pages of 4KiB
Evaluation of the NUMA policies

Speedup relative to Linux FT

[presented at Eurosys’17]
Evaluation of the NUMA policies

First conclusion

All the NUMA policies are important
Each application needs its own NUMA policy

Speedup relative to Linux FT
Second study

▪ Predict which NUMA policy is the best for an application

▪ Goal:
  • Select the most efficient NUMA policy
  • Understand the memory access behavior
Predict the NUMA policy

- Measure the memory access imbalance with first-touch
  Relative standard deviation around the average #accesses per node

<table>
<thead>
<tr>
<th>Imbalance</th>
<th>0%</th>
<th>40%</th>
<th>62%</th>
<th>83%</th>
<th>107%</th>
<th>138%</th>
<th>185%</th>
<th>283%</th>
</tr>
</thead>
<tbody>
<tr>
<td># of accessed nodes</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Perfect balance

All the accesses go to a single node
Predict the NUMA policy

- Measure the memory access imbalance with first-touch
  Relative standard deviation around the average \#accesses per node

<table>
<thead>
<tr>
<th>Imbalance</th>
<th>0%</th>
<th>40%</th>
<th>62%</th>
<th>83%</th>
<th>107%</th>
<th>138%</th>
<th>185%</th>
<th>283%</th>
</tr>
</thead>
<tbody>
<tr>
<td># of accessed nodes</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Low imbalance

High imbalance

Moderate imbalance
### Predict the NUMA policy

**Low imbalance with first-touch**  
Often because we already have a good locality

<table>
<thead>
<tr>
<th>Imbalance</th>
<th>0%</th>
<th>40%</th>
<th>62%</th>
<th>83%</th>
<th>107%</th>
<th>138%</th>
<th>185%</th>
<th>283%</th>
</tr>
</thead>
<tbody>
<tr>
<td># of accessed nodes</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Low imbalance  
Moderate imbalance  
High imbalance

---

Multicore Programming  
Non-Uniform Memory Architectures
Predict the NUMA policy

Low imbalance with first-touch
Often because we already have a good locality
=> keep first-touch
(1% slower than best in average)

<table>
<thead>
<tr>
<th>Imbalance</th>
<th>0%</th>
<th>40%</th>
<th>62%</th>
<th>83%</th>
<th>107%</th>
<th>138%</th>
<th>185%</th>
<th>283%</th>
</tr>
</thead>
<tbody>
<tr>
<td># of accessed nodes</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

First-touch

High imbalance

Moderate imbalance
Predict the NUMA policy

Moderate imbalance with first-touch
First-touch **roughly balances the load but locality is not perfect**

<table>
<thead>
<tr>
<th>Imbalance</th>
<th>0%</th>
<th>40%</th>
<th>62%</th>
<th>83%</th>
<th>107%</th>
<th>138%</th>
<th>185%</th>
<th>283%</th>
</tr>
</thead>
<tbody>
<tr>
<td># of accessed nodes</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Predict the NUMA policy

Moderate imbalance with first-touch
First-touch roughly balances the load but locality is not perfect
⇒ use First-touch/Carrefour
(2% slower than best in average)

<table>
<thead>
<tr>
<th>Imbalance</th>
<th>0%</th>
<th>40%</th>
<th>62%</th>
<th>83%</th>
<th>107%</th>
<th>138%</th>
<th>185%</th>
<th>283%</th>
</tr>
</thead>
<tbody>
<tr>
<td># of accessed nodes</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

First-touch

High imbalance

First-touch/Carrefour
Predict the NUMA policy

High imbalance with first-touch
Interleaved balances the load and Carrefour improves locality
⇒ use Interleaved/Carrefour

<table>
<thead>
<tr>
<th>Imbalance</th>
<th>0%</th>
<th>40%</th>
<th>62%</th>
<th>83%</th>
<th>107%</th>
<th>138%</th>
<th>185%</th>
<th>283%</th>
</tr>
</thead>
<tbody>
<tr>
<td># of accessed nodes</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

First-touch

Interleaved/Carrefour

First-touch/Carrefour
Predict the NUMA policy

High imbalance with first-touch
Interleaved balances the load and Carrefour improves locality
⇒ use Interleaved/Carrefour
(2% slower than best in average)

<table>
<thead>
<tr>
<th>Imbalance</th>
<th>0%</th>
<th>40%</th>
<th>62%</th>
<th>83%</th>
<th>107%</th>
<th>138%</th>
<th>185%</th>
<th>283%</th>
</tr>
</thead>
<tbody>
<tr>
<td># of accessed nodes</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

First-touch
Interleaved/Carrefour
First-touch/Carrefour
Predict the NUMA policy

High imbalance with first-touch
Interleaved balances the load and Carrefour improves locality

Second conclusion

We can reasonably predict the best NUMA policy of an application

<table>
<thead>
<tr>
<th>Imbalance</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td># of accessed nodes</td>
<td>8</td>
</tr>
</tbody>
</table>

First-touch

Interleaved/Carrefour

First-touch/Carrefour
Third study

- How a data analytic application behaves?
  - Page rank query on the friendster dataset with Spark
  - Heap of 40GB, JVM with the Parallel Scavenge (PS) GC
Application scalability of Spark

Speedup in term of completion time

Bad scalability after 12 cores

Ideal scalability

Performance of Spark (40GB of heap)
A bottleneck in the garbage collector

Time spent in the application
Roughly scales with the number of cores

Completion Time (s)

Time spent in the garbage collector
Does not seem to scale after 12 cores

#cores = #threads
A bottleneck in the garbage collector

The garbage collector does not scale

GC throughput
(GB collected per second)

Bad scalability after 12 cores

Performance of the GC in Spark (40GB of heap)
First optimizations: synchronizations

- Remove useless synchronizations in the garbage collector
  - Trades the genericity of the code for better performance
- Optimize the locks
  - Futex instead of hand tuned
- Optimized lock-free queue for the work stealing
First optimizations: synchronizations

- Remove useless synchronizations in the garbage collector
  - Trades the genericity of the code for better performance
- Optimize the locks
  - Futex instead of hand tuned
- Optimized lock-free queue for the work stealing

GC throughput
(GB collected per second)

Better performance
but does not solve the scalability issue
Second optimizations: NUMAGiC

- The problem: a GC thread accesses any node
Second optimizations: NUMAGiC

- The problem: a GC thread accesses any node
Second optimizations: NUMAGiC

- The problem: a GC thread accesses any node

![Diagram showing remote accesses between Node 0 and Node 1]
Second optimizations: NUMAGiC

- Idea: distributed memory => distributed GC design
Second optimizations: NUMAGiC

- Idea: distributed memory => distributed GC design
  - Trade remote accesses for messages

![Diagram illustrating distributed memory and GC design](image-url)
Second optimizations: NUMAGiC

- Idea: distributed memory => distributed GC design
  - Trade remote accesses for messages
Second optimizations: NUMAGiC

- Idea: distributed memory => distributed GC design
  - Trade remote accesses for messages
Second optimizations: NUMAGiC

- Idea: distributed memory => distributed GC design
  - Trade remote accesses for messages

![Diagram of memory and GC threads between nodes 0 and 1]
Second optimizations: NUMAGiC

- Idea: distributed memory => distributed GC design
  - Trade remote accesses for messages
Second optimizations: NUMAGiC

- Idea: distributed memory => distributed GC design
  - Trade remote accesses for messages

![Diagram of distributed memory and GC threads](image)

- Node 0
- Node 1
- Memory
- GC Thread 0
- GC Thread 1

**Memory**

**Node 0**

**Node 1**

**GC Thread 0**

**GC Thread 1**
As is, messages degrades performance

- Problem: a message is more costly than a remote access

Node 0

Too many messages

Node 1
As is, messages degrades performance 😞

- Problem: a message is more costly than a remote access

⇒ Inter-node references must be minimized

Too many messages
As is, messages degrades performance

- Problem: a message is more costly than a remote access

⇒ Inter-node references must be minimized
  - Observation: a thread mostly connects objects it has allocated

Only 1% of references between objects allocated by different threads in Spark
As is, messages degrades performance 😞

- Problem: a message is more costly than a remote access

⇒ Inter-node references must be minimized

- Observation: a thread mostly connects objects it has allocated
- Heuristics: allocate and let the objects on their allocation nodes

Too many messages
But few inter-node references degrade the parallelism! 😞

Node 1 idles while node 0 collects its memory
But few inter-node references degrade the parallelism! 😞

Solution: adaptive algorithm

- Local mode: send messages when not idling
- Thief mode: steal and access remote objects when idling

Node 1 idles while node 0 collects its memory
Performance of NumaGiC

GC throughput
(GB collected per second)

Performance of the GC with Spark (40GB of heap)
Performance of the application

Application speedup in term of completion time

Performance of Spark (40GB of heap)

Completion time divided by two

#cores = #threads

NumaGiC
synchroPS
synchroPS + interl.
PS

gidra@asplos15
Third lessons

- NUMA can have a large impact on performance
  - On data analytic applications written in Java

- We can design better NUMA policies than the ones proposed by default in Linux
  - Technically inspired by distributed systems
Fourth study

How a hypervisor behaves on a NUMA machine?

- Study of a set of 29 parallel applications
  - Parsec, NPB, MosBench, X-stream, YCSB (Cassandra, MangoDB)
- Hypervisor overhead when we increase the #cores
Fourth study

How a hypervisor behaves on a NUMA machine?

- Study of a set of 29 parallel applications
  Parsec, NPB, MosBench, X-stream, YCSB (Cassandra, MangoDB)
- Hypervisor overhead when we increase the #cores

Up to a 9.5 time slowdown in Xen with 48 cores while overhead is negligible with 1 core
Memory access latency causes the overhead

![Chart showing memory access latency for different benchmarks across different core counts.](chart.png)

- **Xen Overhead**
  - With 1 core/vCPU/thread
  - With 48 cores/vCPUs/threads

- **Memory access latency (cycles per access)**
  - Linux 48 cores
  - Xen 48 cores
Solution: XenNUMA

- Implement generic NUMA policies in Xen
  - Interleaved: roughly randomize memory access
  - First-touch: allocate from the node that triggers the first access
  - Carrefour: dynamic policies proposed by Dashti et al.

- Add a new interface between Linux and Xen
  - To select a NUMA policy for a process
  - To know which pages are allocated to a process
    In order to allocate a page from the node that triggers the first access

- Rewrite the memory sub-system of Xen
Overhead of Xen with 48 cores/vCPUs

- Settings: 48 vCPUs (pinned) on the 48 pCPUs
  - Xen uses the default (nonexistent) NUMA policy
  - XenNUMA uses the best possible NUMA policy

Results:
- Performance improvement of up to 700%
- Virtualization costs less than 50% for
  - 12/29 applications with Xen
  - 23/29 applications with XenNUMA
XenNUMA is not a satisfactory solution because XenNUMA hides the topology

Prevents the use of System Runtime Libraries (SLR) optimizations:

- Impossible to use NumaGiC or other application-specific NUMA policies
- Impossible to use NUMA-aware allocators
  - TCMalloc, JEMalloc

=> Bad performance for many applications
Exposing the topology is not more efficient

vNUMA exposes the initial NUMA topology
- But the hypervisor may change the NUMA topology at runtime
=> makes SLR and OS work with a stale topology
Exposing the topology is not more efficient

vNUMA exposes the initial NUMA topology
- But the hypervisor may change the NUMA topology at runtime

=> makes SLR and OS work with a stale topology
**XPV: eXtended ParaVirtualization**

- Expose the initial NUMA topology
- Add notifications when the NUMA topology changes
  - Used by the OS and the SLR to update the topology
  - Few lines of code changed

<table>
<thead>
<tr>
<th>System</th>
<th># files</th>
<th># LOC changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xen 4.9</td>
<td>8</td>
<td>117</td>
</tr>
<tr>
<td>KVM from Linux 4.14</td>
<td>6</td>
<td>218</td>
</tr>
<tr>
<td>Linux 4.14</td>
<td>26</td>
<td>670</td>
</tr>
<tr>
<td>FreeBSD 11.0</td>
<td>23</td>
<td>708</td>
</tr>
<tr>
<td>HotSpot 8</td>
<td>3</td>
<td>53</td>
</tr>
<tr>
<td>TCMalloc 2.6.90</td>
<td>3</td>
<td>65</td>
</tr>
<tr>
<td>jemalloc 5.0.1</td>
<td>9</td>
<td>86</td>
</tr>
</tbody>
</table>
XPV versus XenNUMA (fixed NUMA topology)

By exposing the NUMA topology: up to 130% improvement

![Bar chart comparing XPV with and without NUMA SRL against different XenNUMA policies.]
XPV facing topology changes

- Xen migrates vCPUs to balance the load
  - Three identical VMs
  - 48 vCPUs/42 pCPUs

Improvement: up to 127%
To take away

- NUMA can have a large impact on performance
  - On many parallel applications (both native and Java)

- We can already significantly improve performances with generic NUMA policies
  - We can predict which generic policy can give the best performance

- For some applications/SLRs, we need specific policies
  - JVM, Databases, locks, NUMA-aware allocators…

- We can mitigate NUMA effects even in hypervisors