Extending Mozart 2 to support multicore processors

Benoit Daloze
Supervisor
Peter Van Roy
Readers
Per Brand
Sébastien Doeraene
Université Catholique de Louvain
Introduction

An example

Communication

Conclusion
The convenient expression of parallelism is an industry-wide challenge

Thomas Würthinger et al. One VM to Rule Them All

This thesis aims to:

- better support multicore processors in Oz
- bring the ability to run efficiently multiple tasks in parallel
- provide a strong foundation in Oz for parallel computing
Oz

- Multi-paradigm programming language
- Created at the Saarland University in 1991
- Mainly supported by the Swedish Institute of Computer Science and UCL

{Show 'Hello World!'}
Mozart and Mozart 2

Mozart is the *de facto* implementation of Oz. It has now been deprecated in favor of its successor, Mozart 2.

Mozart 2 is a new implementation of Oz which focuses on

- Portability (not architecture dependent, few runtime dependencies)
  - Mozart 1 can only run on 32-bit systems

- Extensibility through clarity of the design and the code
  - The huge number of extensions made to Mozart 1 made its codebase very hard to modify
Concurrency and Parallelism

**Concurrency** exists when at least two threads are making progress.

**Parallelism** arises when at least two threads are executing simultaneously.

We may achieve better performance by *parallelizing* a sequential program.

*Concurrency* alone does not improve performance.
Concurrent in Oz

- Oz supports multiple concurrent paradigms
- Some are even deterministic like *declarative concurrency*

- The *concurrency* primitive is the (lightweight) *thread*
  
  \[
  A = \texttt{thread 2 * 3 end}
  \]

  \[
  B = \texttt{thread 3 * 4 end}
  \]

- They never execute simultaneously
Parallelism with multicore processors

- The operating system primitive for parallel execution is the (heavy) operating system thread.

- The programming model of OS threads is shared-state concurrency, which is very hard to reason about!

- Any operations from different OS threads involving mutable memory must be synchronized.
How to achieve parallelism?

There are many ways, assuming we have a multicore processor

- Using multiple computers, that is distributed computing
- Running multiple Mozart processes and communicating through shared memory (Mozart 1)
- Running multiple virtual machines in the same process, each in its own operating system thread and communicating through the same process memory
- Making the thread construct above parallel
The idea

- Multiple virtual machines (VMs) in a single process
- All VMs may run simultaneously
- Each VM is as independent as possible from others
- They only share a couple structures not worth duplicating
  - The asynchronous IO thread
  - The backup memory space used during garbage collection
- They can easily and efficiently communicate
Communication

- Each VM has its own memory and entities
  - We must *copy* entities across VMs
  - We can only send *stateless* entities

- Either direct copy and we need to synchronize both VMs
- Or dump to some temporary memory on *send* and then load in the receiver on *receive*
  - We need an intermediate format
  - We chose the existing serialization of values: Pickle
Handling failure
What happens when a VM terminates?

- When a VM is created, it is also monitored by its parent
- The parent receives termination information when the VM terminates
- The termination information is a record terminated(VMIdentifier reason:Reason)
- The Reason is normal, exception, kill or outOfMemory
A robust API

If using the VM identifiers properly (not trying to guess them), no procedure of the VM module will ever throw an error!

Sending to a dead VM is simply ignored (monitors can detect failure and we keep send asynchronous)
Introduction

An example

Communication

Conclusion
An example

Estimating $\pi$ with a definite integral

$$\tan \left( \frac{\pi}{4} \right) = 1 \iff \frac{\pi}{4} = \arctan(1)$$

$$\pi = 4 \arctan(1) = 4 \int_0^1 \frac{1}{1 + x^2} \, dx$$
The rectangle method

\[
\int_{0}^{1} \frac{1}{1+x^2} \, dx
\]
The rectangle method

With $n = 10$

\[\frac{1}{1+x^2}\]
A sequential implementation for \(4 \int_0^1 \frac{1}{1+x^2} \, dx\)

\[\begin{align*}
N &= 2000000 \\
Dx &= 1.0 \div \{\text{IntToFloat } N}\n\end{align*}\]

\[
\text{fun } \{\text{EstimatePi}\}
\]

\[
\text{fun } \{\text{Loop X Sum}\}
\]

\[
\text{if } X \geq 1.0 \text{ then }
\]

\[
\text{Sum}
\]

\[
\text{else}
\]

\[
\{\text{Loop } X+Dx \text{ Sum+Dx}/(1.0 + X*X)\}
\]

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{in}
\]

\[
4.0 \times \{\text{Loop 0.0 0.0}\}
\]

\[
\text{end}
\]

\[
\{\text{Show } \{\text{EstimatePi}\}\}
\]
Design of a parallel implementation

Splitting the interval in subintervals

With $n = 20$

$$y = \frac{1}{1 + x^2}$$
Design of a parallel implementation

Splitting the interval in subintervals

With \( n = 20 \)

\[
\frac{1}{1+x^2}
\]
We need EstimatePi to only consider a subinterval.

```plaintext
fun {EstimatePi From To}
    fun {Loop X Sum}
        if X >= 1.0To then
            Sum
        else
            {Loop X+Dx Sum+Dx/(1.0 + X*X)}
        end
    end
in
    4.0 * {Loop 0.0From 0.0}
end
```
Distributing the work over 2 cores

\begin{verbatim}
Master={VM.current}

{VM.new functor define
 Part={{EstimatePi 0.5 1.0}
   {Send {VM.getPort Master} Part}
 end _}

PI={EstimatePi 0.0 0.5} + {VM.getStream}.1
\end{verbatim}
Distributing the work over $N$ cores

```
Master = {VM.current}
NVMs = {VM.ncores}
ToF = IntToFloat

for I in 1..NVMs do
  From = {ToF I - 1} / {ToF NVMs}
  To = {ToF I} / {ToF NVMs}
  in
    {VM.new functor define
      Part = part({EstimatePi From To})
      {Send {VM.getPort Master} Part}
    end _}
end
```
Assembling the work of $N$ cores

Parts = thread

\{Filter \{VM.getStream\} fun \{\$ E\}
\{Label E\} == part
end\}
end

PI = \{FoldL \{List.take Parts NVMs\}
fun \{\$ Sum part(X)\} Sum + X end 0.0\}
Speedup in the example

![Bar graph showing speedup vs number of tasks]

- The y-axis represents speedup.
- The x-axis represents the number of tasks.
- The graph shows a significant increase in speedup as the number of tasks increases.
- With 1 task, the speedup is approximately 1.
- With 2 tasks, the speedup is approximately 2.
- With 3 tasks, the speedup is approximately 3.
- With 4 tasks, the speedup is approximately 4.
Introduction

An example

Communication

Conclusion
Sending values on ports

- Sending values on VMPorts should be *fast*, it is a major reason for having many VMs in the *same* process

- Actual results for sending *unit*

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Port</td>
<td>0.302 ± 0.016 µs</td>
</tr>
<tr>
<td>VMPort</td>
<td>120 ± 1 µs</td>
</tr>
</tbody>
</table>

- 3 orders of magnitude (±400 times) slower!
Sending unit on a port

Graph showing time in microseconds (μs) on the y-axis and ports (Port, VMPort) on the x-axis. The bar for VMPort reaches up to 120 μs.
Sending unit on a port and pickling

- Port
- VMPort

- Sending
- Pickling

-bar chart with x-axis labeled 'Port' and 'VMPort', y-axis labeled 'μs'
When *Future work* is done: a faster pickler
Sending unit on a port and pickling

- Port
- newVMPort
- VMPort

µs

- sending
- pickling

Graph showing the comparison between sending and pickling times for different ports.
Sending values on ports with a faster pickler

- Actual results for sending `unit`

<table>
<thead>
<tr>
<th>Port Type</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Port</td>
<td>0.302 ± 0.016 µs</td>
</tr>
<tr>
<td>new VMPort</td>
<td>5.44 ± 0.68 µs</td>
</tr>
<tr>
<td>VMPort</td>
<td>120 ± 1 µs</td>
</tr>
</tbody>
</table>

- Only 1 order of magnitude slower!
Effect on a realistic example: the heat equation

At each time step, a task gives the values of its border cells to the neighbor tasks.

- tasks: 4
- time steps: 100
- height: 200
- width: 200

$$100 \times (1 + 2 + 2 + 1) = 600$$
messages, each being a tuple of 200 integers.
Effect on a realistic example: the heat equation

![Graph showing speedup for sequential and parallel (4 cores) of old pickler and new pickler.]

- **old pickler**
- **new pickler**
Introduction

An example

Communication

Conclusion
Conclusion

We contributed to Mozart 2:

- The ability to run multiple VMs in the same process
- A programming model to achieve parallel computing safely with message-passing concurrency behavior
- A serializer with superior performance
- A much improved memory management
- A better test runner and tests for the new features
- A couple bug and memory leak fixes
Memory management

Before: always using 768 MB (configurable)

After: using a reasonable amount of memory, computed from the actual memory usage after garbage collection


Any questions?
Memory management: evolution of the heap

**Between GCs**
- Actual heap size
- Wished heap size
- Threshold
  - Tolerance: 10% of wished
  - Free: < 75% of threshold
  - Allocated: > 25% of threshold

**Before GC**
- Actual heap size
- Wished heap size
- Threshold
  - Tolerance: 10% of wished
  - Allocated: ≥ threshold
Memory management: shrinking the heap

After GC

old heap size

old heap size/2
wished heap size
threshold

tolerance

free
= 75% of threshold

allocated = active

Shrink

new heap size
wished heap size
threshold

tolerance

new heap size/2

free
= 75% of threshold

allocated = active
Memory management: growing the heap

**After GC**

- wished heap size threshold
- old heap size
- new heap size = 75% of threshold
- allocated = active

**Grow**

- wished heap size threshold
- old heap size/2
- new heap size = 75% of threshold
- allocated = active
An abstraction

% import VMUtils

fun {Work From #To}
    {EstimatePi From To}
end

Parts = {VMUtils.distribute Work
        VMUtils.floatChunk}

PI = {FoldL Parts Number. '+', 0.0}