Running MPI applications on Toro unikernel

www.torokernel.io

Matias Vara Larsen
matiasevara@gmail.com
Who am I?

- I am passionate about operating system development and virtualization technologies.
- I have worked at Citrix, Tttech, Huawei and currently at Vates.
- matiasevara@gmail.com
- https://github.com/MatiasVara
Outline

• Toro unikernel
• MPI over Toro
• OSU benchmarks
Virtual Machine / Baremetal

Operating System

Kernel

User Application

Device Model

KVM

This is your MPI application!

Hardware

Memory

CPUs
Virtual Machine / Baremetal

- User Application
- Operating System
- Kernel

Ring 3:
Ring 0:

Device Model

Ring -1:

KVM

Hardware
Memory
CPUs
Virtual Machine / Baremetal

User Application

Operating System

Scheduler
Filesystem
Drivers

Kernel

Device Model

KVM

Hardware
Memory
CPUs
Virtual Machine / Baremetal

Operating System

Kernel

Device Model

Scheduler

Filesystem

User Application

KVM

Too general for a single purpose application!

Hardware

Memory

CPUs
Virtual Machine / Baremetal

User Application
e.g., Osv, MirageOS, Unikraft, NanoVMs

Kernel

Device Model

Device Model

KVM

Hardware

Memory

CPUs

Unikernel [1]

User Application

[1] “Unikernels: library operating systems for the cloud”, Madhavapeddy et al., 2013
Toro is an application-oriented unikernel to efficiently deploy parallel applications
How does Toro leverage multicore?

- Memory per core
- Cooperative Scheduler
- Core to Core communication based on VirtIO
Dedicated Memory

Memory space in Toro

Memory Region 1
Memory Region 2

Toro reserves the same amount of memory for each core

Core 1

Core 2
Dedicated Memory

The memory allocator keeps separated structures for each chunk.
Dedicated Memory

Memory space in Toro

- Memory Region 1
- Memory Region 2

TORO Memory allocator

ToroGetMem()

Thread 1
- Core 1

Thread 2
- Core 2

Allocations from Core 1 always get memory from Region 1
The programmer decides for each thread on which core to execute it. In Toro, there are only threads.

BeginThread(DataBase, Thread1, Core1)

BeginThread(Microservice, Thread2, Core2)
Scheduler

Each thread decides when to yield the CPU aka cooperative thread scheduling, eg: Disk I/O, FS, Socket
Each core has its own scheduler

Non-preemptive scheduler
Core-to-Core communication

• Each core can communicate with any other core by using dedicated queues

• It is based on two primitives:
  – procedure SendTo(Core: DWORD; Buffer: Pointer; Len: DWORD);
  – procedure RecvFrom(Core: DWORD; Buffer: Pointer);

• These are the ingredients to implement MPI_Gather(), MPI_Bcast() and MPI_Scatter()
Core-to-Core communication

Queue to send from Core 1 to Core 2

RX virtqueue

buffer ring
avail ring
used ring

TX virtqueue

buffer ring
avail ring
used ring

TX virtqueue

buffer ring
avail ring
used ring

RX virtqueue

buffer ring
avail ring
used ring

Queue to send from Core 2 to Core 1
Core-to-Core communication

Produced by core 1 and consumed by core 2

Produced by core 2 and consumed by core 1

RX virtqueue

TX virtqueue

Core 1

Core 2
Core-to-Core communication

“It’s all talk until the code runs.” - Ward Cunningham
The generated binary is **immutable**[1], i.e., the generated image can be used across different hypervisors without the need to recompile it.
How a MPI application is deployed?
Benchmarking

- I benchmark it by using the OSU MPI_Barrier (see OSU microbenchmarks[1]) that measures the latency of the MPI_Barrier() function for a given number of nodes
- I deploy it by using a single VM (QEMU microvm/KVM) with 4, 8, 16 and 32 cores
- I run it on a 1 x Intel Xeon Gold 6314U, 32 cores @ 2.3 GHz

[1] https://mvapich.cse.ohio-state.edu/benchmarks/
Note that [1] reports between 20ns to 30ns for 16 nodes in the Cray XC40 Xeon Phi Systems.

Questions?
Thanks!

> Toro kernel is open source on GitHub: https://github.com/torokernel/torokernel
> Follow me on Twitter: https://twitter.com/ToroKernel
> Sponsor me on GitHub: https://github.com/sponsors/MatiasVara
> Watch me on Youtube: https://www.youtube.com/@torokernel3078