Extending OpenSHMEM's Footprint for Rust-on-RISCV, Python, Nim, and HPX

Christopher Taylor
Tactical Computing Labs

tactcomplabs.com

Summary

- Introduction
- Rust
- Python
- Nim
- HPX

Introduction

- Research & Development Firm
  - HPC Software Specialization
    - Compilers
    - Runtime Systems
    - Scientific Computing (Machine Learning, AI)
    - Modeling and Simulation (Structural Simulation Toolkit - SST)
  - Hardware Specialization
    - RISC-V

Rust

Rust

- Rusty2 - OpenSHMEM bindings for Rust
  - Stonybrook, Rebecca Hassett, and Tony Curtis
- Successful Rust-on-RISCV bring up!
  - Verified compiler has a self-hosting, multi-stage, bring up process
- Successfully deployed Rusty2 over O3S-UCX!

Rust

- Successfully implemented/deployed to a Slurm cluster
  - Infiniband support on RISCV works* (Connect X3, X4, IPoIB)
- Hardware
  - SiFive Hifive Unmatched Development Boards
  - Pine64 Star64 (8GB) Single Board Computers

Python

Python

- Codon is an LLVM compiler for Python (Exaloop)
  - Python programs compile to machine executable code!
  - Boehm’s garbage collection
  - Advanced developers can inline LLVM-IR into Python applications
  - Foreign Function Interface (FFI) only supports C (LLVM)
  - Pre-Mojo technology
  - 2 year open source licensing “age off” to Apache 2 License
- OpenMP is supported natively in Codon
  - Users apply Python annotations (@) to loops

Python

- Codon provides support for `Static[T]` variables
  - Variables are placed into the compiled programs data segment
  - Accessible to the Partitioned Global Address Space
- OpenSHMEM integrates into Codon w/LLVM FFI
- Bindings provide basic support for OpenSHMEM operations
- 2 new types are introduced: Runtime and a `SymmetricArray[T]`

Python

- Runtime is a wrapper type for the standard OpenSHMEM runtime functionality
- The type exists to provide scoped initialization and finalization of the OpenSHMEM runtime using the Python `with` expression
- The idea of scoped management of the runtime is derived from C++’s std::scoped_lock<T> type for handling mutex and locks.

Python

- `SymmetricArray[T]` folds OpenSHMEM operations into a container data type
- Supports
  - localized-slice
  - copy, deepcopy
  - put, get
  - max, min, sum, prod
  - all2all, broadcast

Python

```
from openshmem import Runtime, SymmetricArray
with Runtime() as rt:
    print("PE", rt.my_pe(), rt.n_pes())
    a = SymmetricArray[int](10)
    print("symmetric array allocated", len(a))
    print("HERE")
```

Nim

Nim

- Nim is a ~10 year old programming language
  - Syntax is *very* similar to Python (iterators, generators, etc)
  - Compiles to C and C++ and (ECMAScript/Javascript, WASM, etc)
  - Boasts memory management features similar to Rust
- Metaprogramming features
  - Compile-time logic manipulates the compiler's abstract syntax tree representation of the program.
  - Macros can emit C/C++ code during compilation
- Templates and generic support similar to C++
  - Users can define 'sets' of types
  - Consider a `typedef` that references a list of types or `std::variant<>` from C++. `SomeNumber`

Nim

- All the fundamental OpenSHMEM functionality is exposed as a direct binding
- Convenience functions and datatypes are provided to improve productivity and experiment with the language
    - Symmetric Arrays and Symmetric Scalars
    - Heavy use of the macro and template system
    - All output code targets the C code generator

Nim

- Users can create symmetric variables (scalars) and arrays; Nim's macro system generates the appropriate C code
  - Static variables and sequences are compiled into the .data segment of the compiled C executable, which places them into the partitioned global address space
  - Nim symmetric variables and sequences, with a known-fixed size, become static C arrays
  - Symmetric arrays with dynamic sizes are represented in C using pointers and are forwarded into shmem_malloc calls

Nim

- Symmetric Arrays
  - Arrays implement Nim’s Sequence interface
- Supports
  - min, max, sum, prod
  - broadcast, alltoall, reduce
  - put, get

Nim

```
import ../sos/sos
import ../sos/bindings
import std/macros

# template function that handles initialization and finalization of
# the OpenSHMEM runtime
#
SymmetricMain:

   var apple : symint                          # => symscalar[int]
   var orange : symsarray[2, int]        # => fixed sized symmetric array
   orange[0] = 1

   # dynamic allocation
   #
   var a : symarrayint = newSymArray[int]([1,2,3,4,5])     # => fixed size sequence
   var b : symarrayint = newSymArray[int](5)                   # => can accept a literal or variable

   let mrmin = minop.reduce(WORLD, b, a)                     # `minop` is an enum of type `ReductionKind`
   echo(pe, ' ', mrmin)

   let mmrmin = min(WORLD, b, a)                                 # performs the same operation as above
   echo(pe, ' ', mmrmin)
```

HPX

HPX

- STE||AR Group (LSU CCT, Swiss Supercomputing Center, etc)
    - Asynchronous Many-Task Runtime System
- Implements ISO C++ standard for data parallelism and concurrency
    - hpx::async => std::async, hpx::future => std::future, coroutines (coawait), etc
    - User-land thread library, 64K thread stack
    - HPX scales exceptionally well to large problem sizes

HPX

- Additional features
  - Uses APEX for performance counters and adaptive runtime features (Active Harmony)
  - Has a communication subsystem called “Parcelport”
  - Distributed container type support for an “Asynchronous Global Address Space”

HPX

- Parcelport is the HPX communication subsystem
    - Implemented using MPI, libfabric, sockets (tcp/ip, udp), LCI
- GASNet & OpenSHMEM support added to HPX Parcelport
    - HPX can now “AGAS over PGAS”
- OpenSHMEM Processing Element (PE) maps to an HPX locality

HPX

- The GASNet Parcelport creates a shared segment (memory page) on each locality, for each locality
    - Example: 4 localities; each locality has 4 shared segments
- The shared segments operate as “communication lanes” where gets/puts can operate and minimize contention between localities

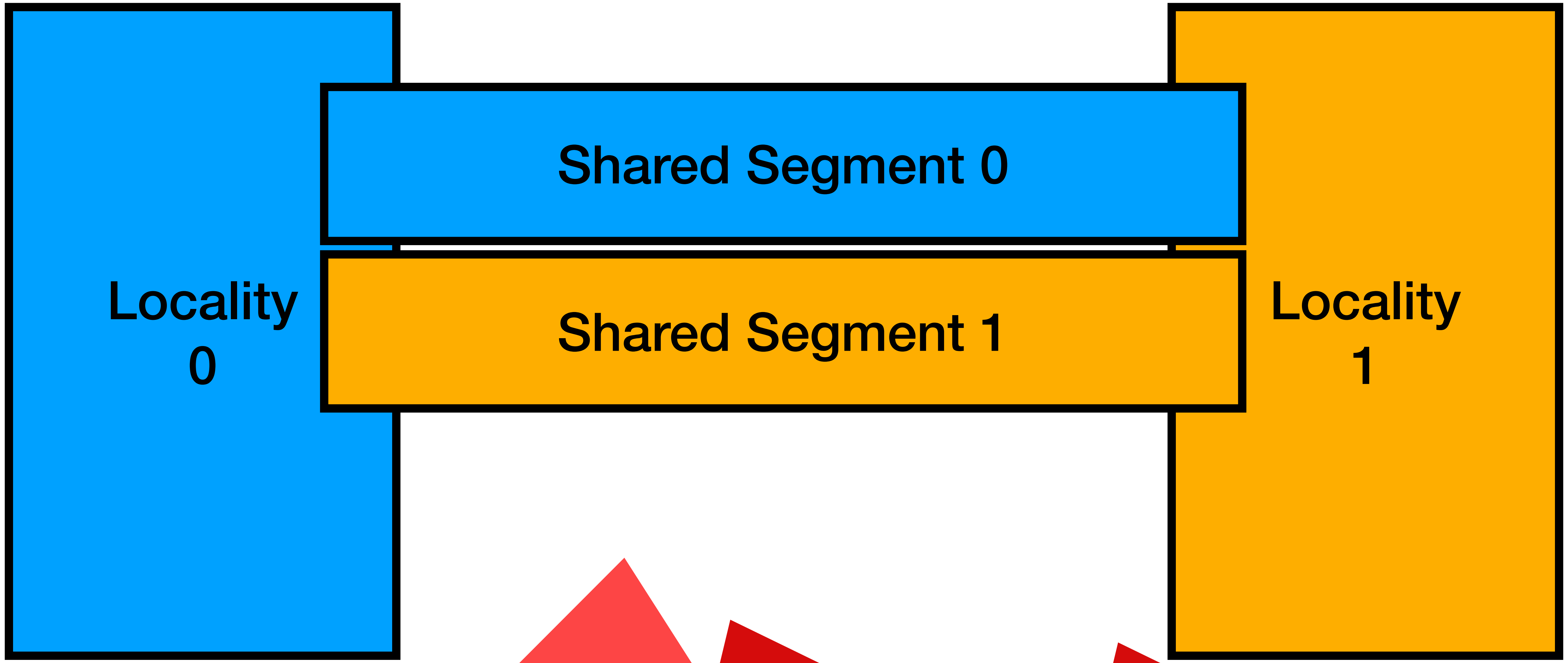
HPX

- GASNet’s active message support performs the heavy lifting of moving data out of the shared segments when data arrives
  - Implementation heavily inspired by Chapel’s GASNet support

HPX

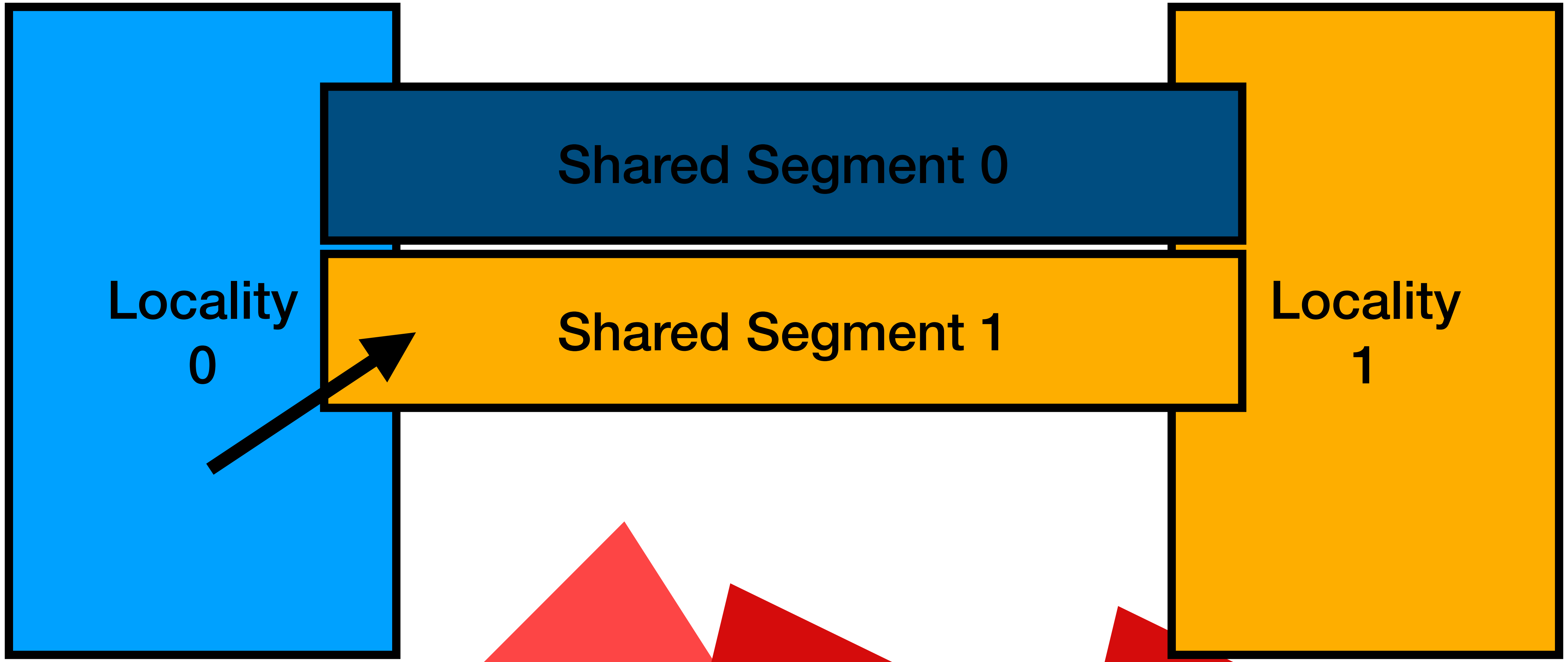
- OpenSHMEM Parcelport operates in a similar fashion w/o using Active messages
- HPX's OpenSHMEM Parcelport creates a shared segment and a symmetric variable for each locality
- OpenSHMEM `put_signal` is used to move data between localities in their exclusive segment on a remote locality
  - `put_signal`, along with the symmetric variable for a locality, is used to indicate a communication event has occurred on the remote locality
  - The remote locality uses an OpenSHMEM `wait` to detect when the expected `put_signal` completes
- `put_signal` enables message passing over OpenSHMEM
  - decomposes to a `put` + `put_atomic`

HPX



HPX

Locality 0 to Locality 1



Shared Segment 0

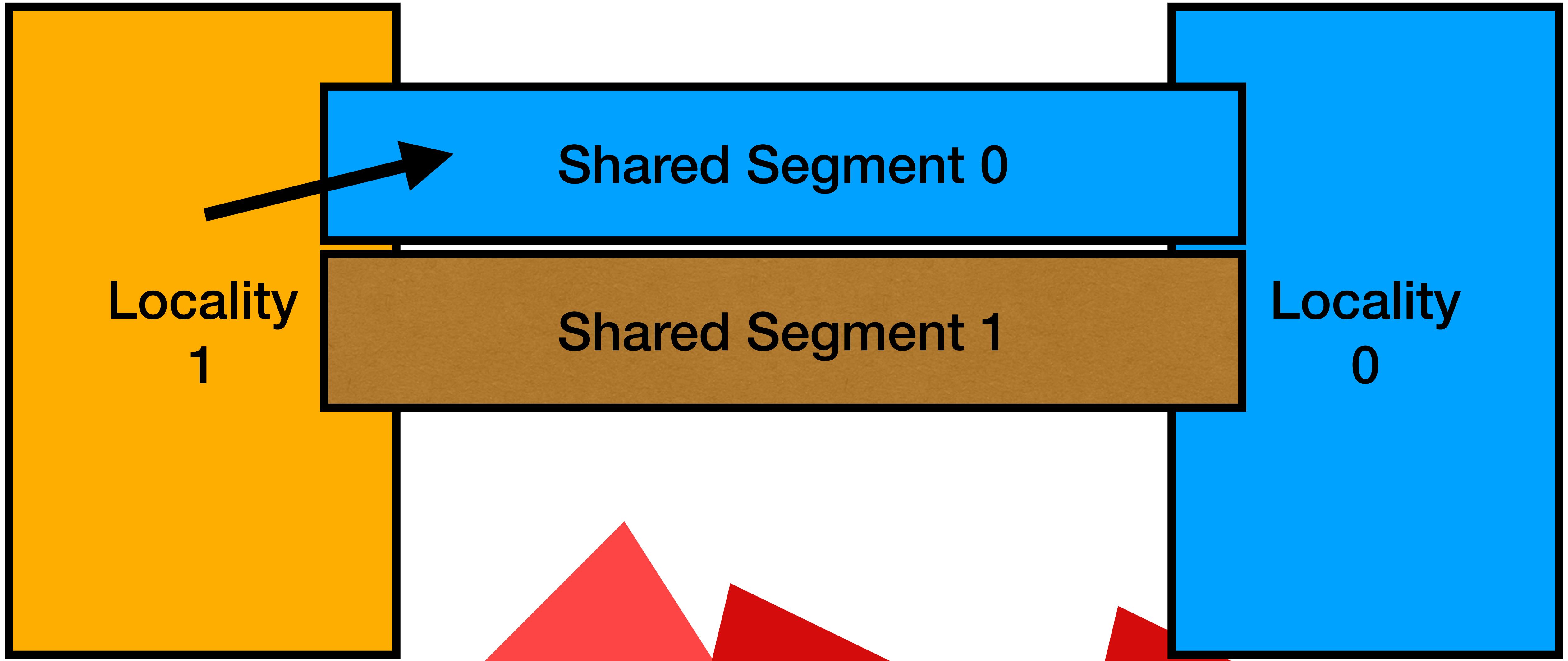
Shared Segment 1

Locality 0

Locality 1

HPX

Locality 1 to Locality 0



Thanks!

TCL