Demonstrating UPC++/Kokkos Interoperability in a Heat Conduction Simulation

Daniel Waters, Colin A. MacLean, Dan Bonachea, Paul H. Hargrove

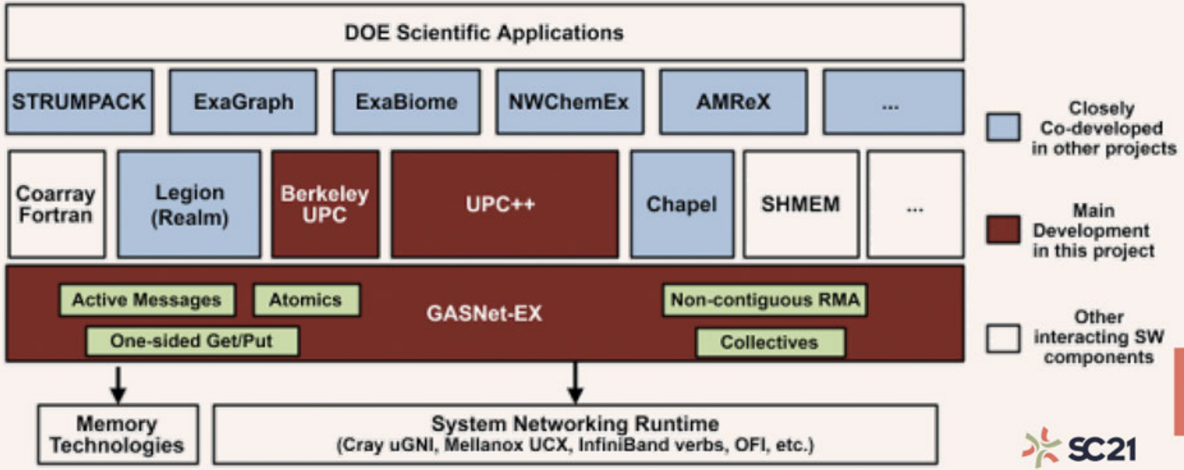
Applied Mathematics and Computational Research Division
Lawrence Berkeley National Laboratory
https://upcxx.lbl.gov/

Paper: doi:10.25344/S4630V Video: doi:10.25344/S4DK5F

https://go.lbl.gov/pagoda

The Pagoda project

- Support for lightweight communication for exascale applications, frameworks and runtimes
  - **GASNet-EX** low-level layer that provides a network-independent interface suitable for Partitioned Global Address Space (PGAS) runtime developers
  - **UPC++** C++ PGAS library for application, framework and library developers, a productivity layer over GASNet-EX



SC21

What does UPC++ offer?

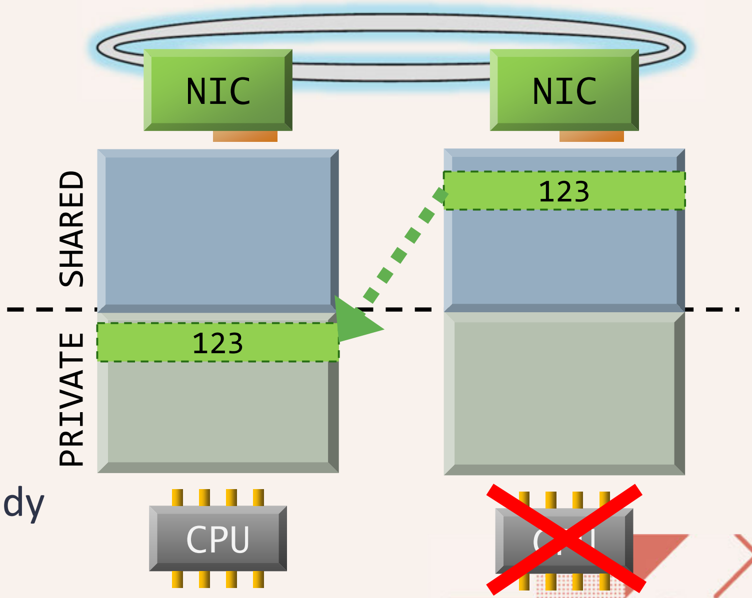
- Communication operations include:
  - **Remote Memory Access (RMA)**:
    - Get/put/atomics on a remote location in another address space
    - One-sided communication leverages low-overhead, zero-copy RDMA
  - **Remote Procedure Call (RPC)**:
    - Moves computation to the data

- Design principles for performance and scalability
  - All communication is syntactically explicit
  - All communication is asynchronous: futures and promises
  - Scalable data structures that avoid unnecessary replication

Asynchronous RMA in UPC++

- By default, all communication operations are split-phased
  - **Initiate** operation
  - **Wait** for completion

```
upcxx::global_ptr<int> gptr1 = ...;
upcxx::future<int> f1 =
                    upcxx::rget(gptr1);
// unrelated work...
int t1 = f1.wait();
```



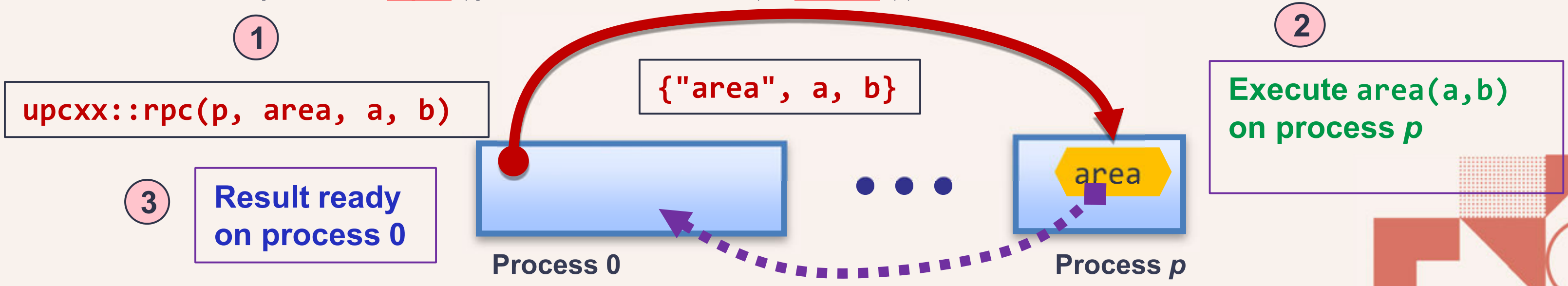
A UPC++ future holds a value and a state: ready/not-ready

Wait returns the result when the rget completes

Remote procedure call (RPC)

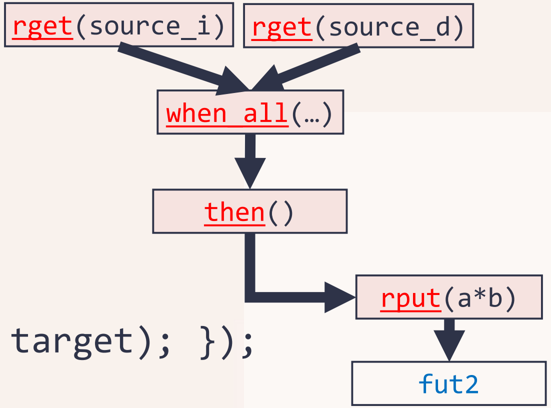
- Execute a function on another process, sending arguments and returning an optional result
  1. Initiator injects the RPC to the *target* process, returning a future
  2. Target process executes `fn(arg1, arg2)` at some later time determined at the target
  3. Result becomes available to the initiator via the future
- Let's imagine that process 0 performs this RPC

```
int area(int a, int b) { return a * b; }
...
... = upcxx::rpc(p, area, a, b).wait();
```



Aggressive Asynchrony via Futures and callbacks

- RMA and RPC both return a *future* object, which represents an operation that may or may not be complete
- Callbacks can be *chained* through calls to then()
- Multiple futures can be *conjoined* with when_all() into a single future that encompasses all their results.
- This code gets two remote values (an int and a double) and puts their product to another location:

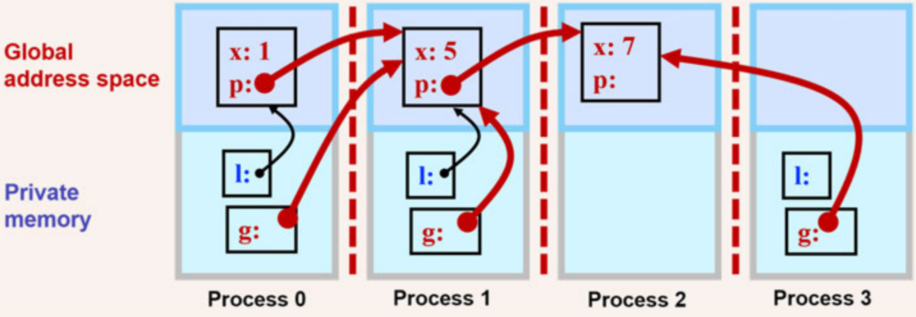



```
global_ptr<int>    source_i = ...;
global_ptr<double> source_d = ...;
global_ptr<double> target   = ...;
...

future<> fut2 =
       when_all(rget(source_i), rget(source_d))
       .then([target](int a, double b) { return rput(a*b, target); });
```

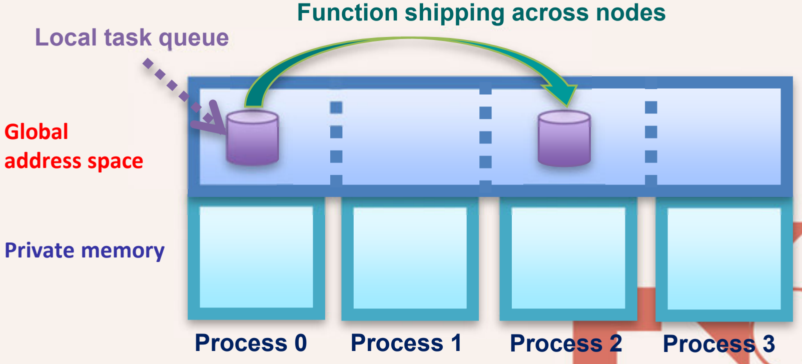
Overview of UPC++ Features

- One-sided RMA and Global Pointers
- Remote Procedure Calls (RPC)
- Future-based asynchrony, continuations
- Remote atomics, non-contiguous RMA
- Serialization, distributed objects
- Teams and non-blocking collectives
- Hierarchical shared mem, node-level bypass
- Memory Kinds for GPU support
- Personas (multi-threading)
- Interoperability with other models
- For more details: https://upcxx.lbl.gov

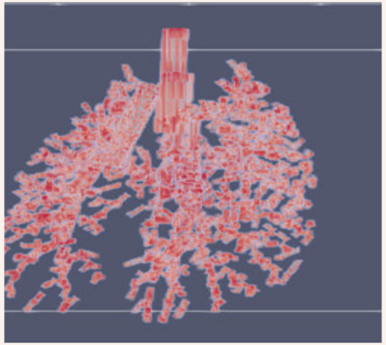
RMA and Global Pointers



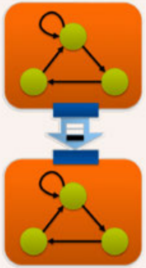
Remote Procedure Calls

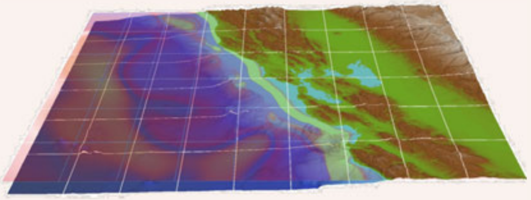


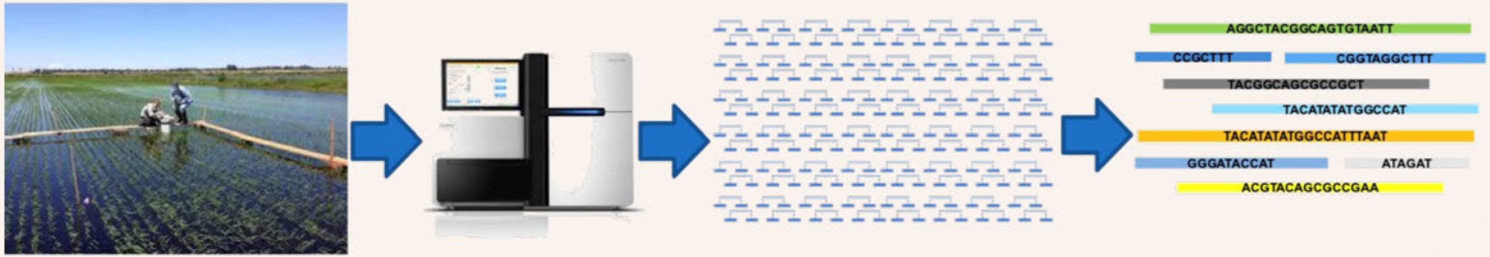
UPC++ Application Examples

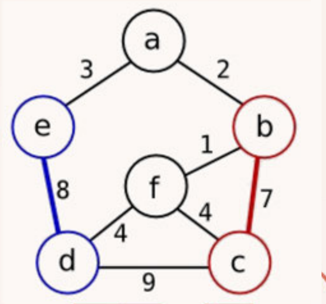


- Several applications have been written using UPC++, resulting in improved programmer productivity and runtime performance. Examples include:
- MetaHipMer, a genome assembler
- symPACK, a sparse symmetric matrix solver
- Pond, an actor-based shallow water simulation
- SIMCoV, an agent-based simulation of lungs with COVID-19
- Mel-UPX, a half-approximate graph matching solver









Applications and images courtesy of: S. Hofmeyr, A. Pöppl, jorisvr.nl, and the ECP ExaBiome team.

GPU Direct

- UPC++ RMA operations utilize Remote Direct Memory Access (RDMA) support when present in hardware
  - RDMA is an important performance feature in all modern HPC networks
  - GPUDirect RDMA (GDR) technology extends RDMA to GPU memory
- UPC++ "memory kinds" extend the PGAS model to encompass GPU memory
  - Distinction between host and GPU memory is part of the global pointer type
  - Permits static choice of appropriate communication code paths
- CUDA-aware MPI implementations lack equivalent static information

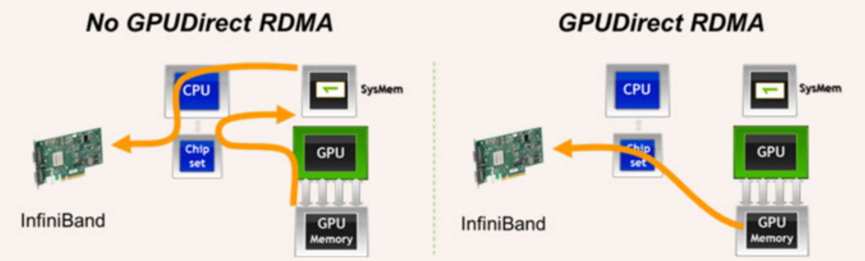


Image source: https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/

Purpose

- Demonstrate interoperability of UPC++ and Kokkos
  - Kokkos: A C++ parallelism abstraction framework designed for portable leveraging of computational resources within a heterogeneous node, such as a GPGPU
  - UPC++: A C++ template library implementing an asynchronous one-sided PGAS programming model using gets, puts, and remote procedure calls complete with serialization of nontrivial C++ objects
  - Complementary C++ libraries emphasizing performance and productivity. Kokkos for on-node parallelism, UPC++ for between nodes.
- Demonstrate that UPC++ can be used to perform inter-node communication with comparable performance to MPI
  - MPI interoperability example provided by Kokkos project
  - MPI message passing is a two-sided model
  - Not an apples-to-apples comparison
  - Goal is merely to show performance isn’t sacrificed

Heat Conduction Example

- 3D cube computation with halo exchange
- Ported to UPC++ without prior experience interoperating with Kokkos
- Both written naïvely, without excessive effort put into optimization

```
pack_T_halo();
compute_inner_dT();
exchange_T_halo();
compute_surface_dT();
Kokkos::fence();
double T_ave = compute_T();
```

Routines performed each timestep

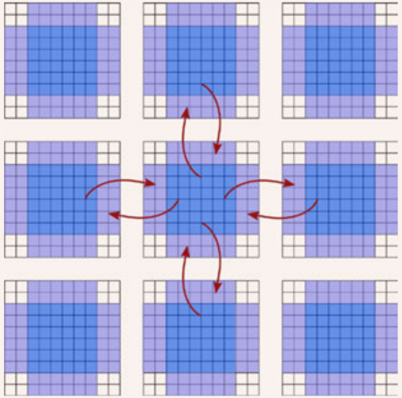


Image source: https://i.stack.imgur.com/R3Q01.png

Porting Communication from MPI to UPC++ : Pseudocode

```
for(n in neighbors) {
  n.outbuf.fence();
  MPI_Irecv(n.inbuf, n.inbuf.size(), MPI_DOUBLE, n.rank, ...);
  MPI_Isend(n.outbuf, n.outbuf.size(), MPI_DOUBLE, n.rank, ...);
}
MPI_Waitall(...);
```

```
for(n in neighbors) {
  n.outbuf.fence();
  upcxx::copy(n.outbuf, n.inbuf, n.outbuf.size(),
              remote_cx::as_rpc([](){ count++; }));
}
while (count < neighbors.size()) upcxx::progress();
```

GPU Memory Allocation

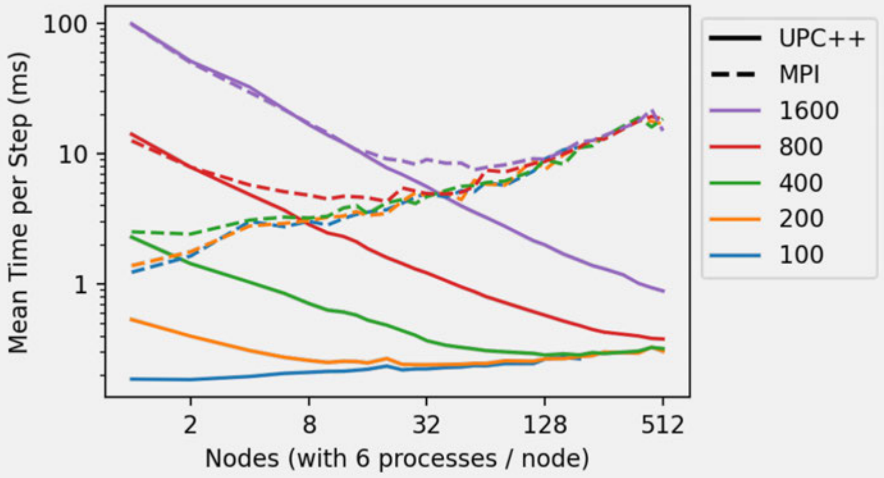
- Original two-sided MPI version interoperates with Kokkos `Views`
- UPC++ needs to interact with GPU memory with its own *memory kinds* interface
  - `upcxx::device_allocator` allocates large GPU memory segment and registers it with the network adapter
  - Partitions of segment assigned to each halo boundary region
  - Use non-owning construction of Kokkos `Views` from pointer allow UPC++ control of allocation
- Exchange of global pointers:
  - `upcxx::dist_object`'s were used to communicate pointers for puts

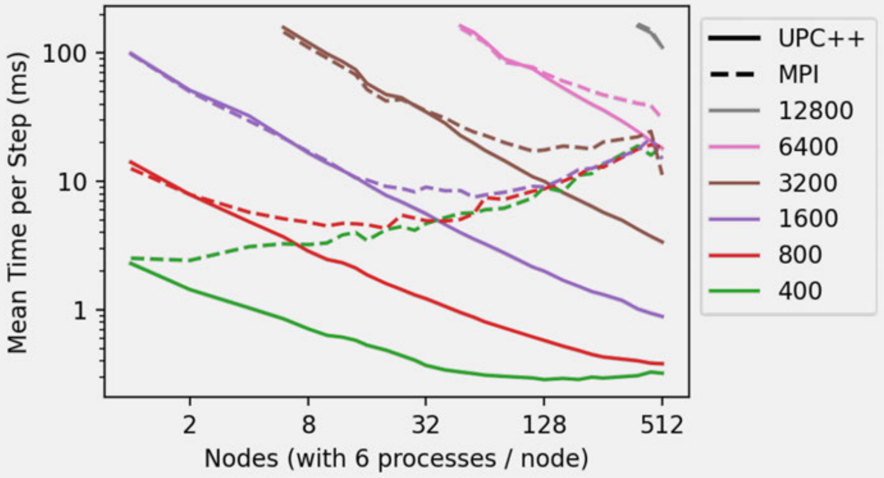
```
using namespace upcxx;

device_allocator<cuda_device> alloc(size, ...);
dist_object<global_ptr<double, memory_kind::cuda_device>> dptr(alloc->allocate<double>(size));
global_ptr<double, memory_kind::cuda_device> nbr = dptr.fetch(nbr_rank).wait();
```

Benchmarking

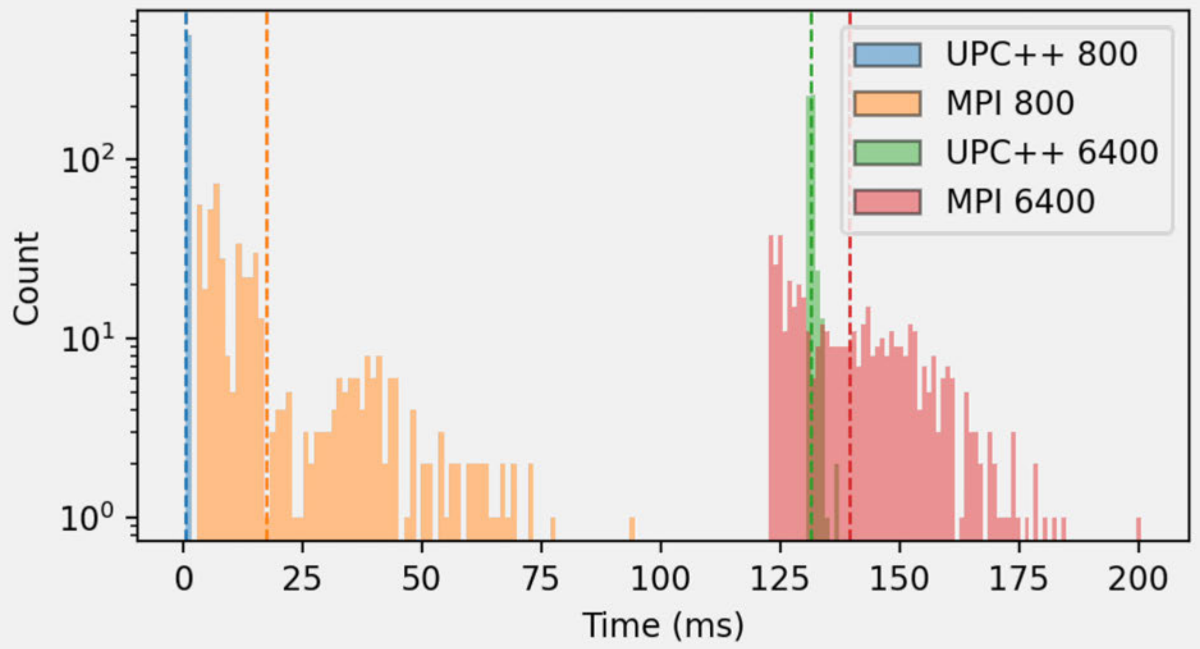
- Benchmarks performed on OLCF Summit
  - Each node has 6 NVIDIA V100 GPUs
  - One process per GPU
- CUDA-aware IBM Spectrum MPI 10.3.1.2-20200121 (for MPI version)
- GPUDirect RDMA enabled UPC++ 2021.3.0
- All benchmarks for a node count were executed within one job to minimize variability of network performance
- 500 timesteps, not including warmup iterations
- Profiled using sliding window of two timesteps used to ensure a communication happens within window, as load imbalance may result in early arrival of incoming data

Benchmark: Mean Time Per Step





Benchmark: Execution Time Variation



Vertical dotted lines indicate medians

128 Nodes

Conclusions

- UPC++ can interoperate with Kokkos without difficulty
- Porting from MPI to UPC++ was likewise straightforward
  - The intention is not to prove superiority to MPI performance
  - Well-tuned usage of both should get close to the hardware performance
- The new UPC++ memory kinds feature is performant
- UPC++ has many sophisticated high-level features, but don’t need to sacrifice performance

Acknowledgements

- This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Thank you!
Questions?

SC21
St. Louis, MO | science & beyond.