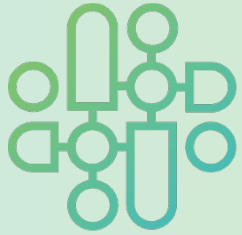


SC23
Denver, CO | i am hpc.

shmem4py: High-Performance One-Sided Communication for Python Applications

Marcin Rogowski *KAUST, NVIDIA*

Jeff R. Hammond *NVIDIA*

David E. Keyes *KAUST*

Lisandro Dalcin *KAUST*

OpenSHMEM

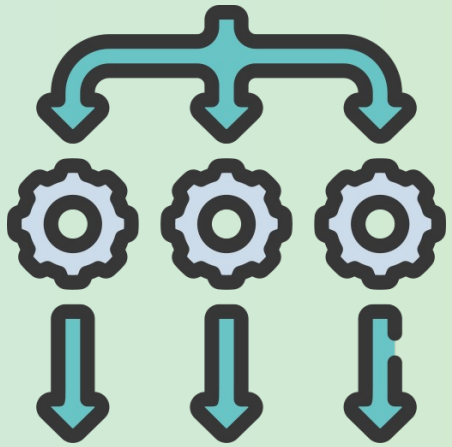
- shmem4py is a Python wrapper for the OpenSHMEM API
- OpenSHMEM:
  - PGAS library resembling MPI
  - Specializes in one-sided communication
  - Focuses on features that have native support in high-performance networks
  - Includes atomic operations and collective operations
- The latest specification is OpenSHMEM 1.5 (2020), work ongoing
- Implementations include:
  - OpenSHMEM Implementation on MPI 2.x (OSHMPI)
  - Sandia OpenSHMEM
  - Open Source Software Solutions (OSSS) OpenSHMEM
  - Implemented in MVAPICH2-X and Open MPI
  - Provided by Cray and SGI on their systems

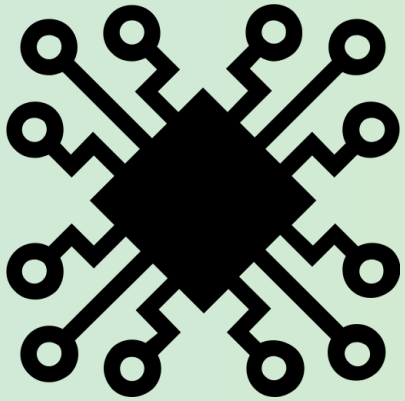
Python and performance

- One of the most widely used programming languages in computing today
- The language of data science and machine learning thanks to NumPy, SciPy, Pandas, Matplotlib, Project Jupyter, etc.
- Interpreted, hence not high-performant
- Python applications that utilize libraries written in lower-level languages such as C, C++, and Fortran can have excellent performance
- Just-in-time (JIT) compilation frameworks made it even more straightforward to achieve good performance

Python and parallelism

- Not the best with any kind of parallelism:
  - Global Interpreter Lock in CPython prevents multithreaded applications from taking full advantage of multiple cores
  - Message-passing and process-based approach favored
- Intra-node parallelism:
  - multiprocessing module
  - concurrent.futures module
- Distributed-memory parallelism:
  - socket module, Python wrapper to ZeroMQ
  - MPI (mpi4py)
  - Dask, Ray, mpi4py.futures





shmem4py

- shmem4py is a Python wrapper for the OpenSHMEM API, supports OpenSHMEM 1.5 API
- Tight interoperability with NumPy, symmetric variables are NumPy arrays (even single-element ones!)
- Similar to mpi4py in design and philosophy:
  - Users are required to invoke communication and synchronization calls explicitly

https://tech.blueyonder.com/python-calling-c++/#

- Goals
  - Exposing the high-performance communication capability of OpenSHMEM libraries to Python applications
  - Enabling OpenSHMEM users to do rapid prototyping in Python, before switching to lower-level languages (usually C)
  - Providing a foundation for building a Python interface to NVSHMEM and other GPU-centric communication libraries

shmem4py

- shmem4py supports all major implementations of the OpenSHMEM specification
  - Cray OpenSHMEMX
  - Open MPI OpenSHMEM (OSHMEM)
  - Open Source Software Solutions (OSSS) OpenSHMEM
  - OpenSHMEM Implementation on MPI 2.x (OSHMPI)
  - Sandia OpenSHMEM (SOS)

- Cray team is aware of issues with Cray OpenSHMEMX v11 support on Shasta

Usage example

```c
#include <stdio.h>
#include <shmem.h>

int main(void)
{
  shmem_init();

  int my_pe = shmem_my_pe();
  int n_pes = shmem_n_pes();

  int *A = shmem_calloc(1, sizeof(int));
  int *B = shmem_calloc(n_pes, sizeof(int));

  A[0] = my_pe;
  shmem_barrier_all();
  shmem_int_fcollect(SHMEM_TEAM_WORLD, B, A, 1);

  for (int i=0; i<n_pes; i++)
    printf("%d ", B[i]);
  printf("\n");

  shmem_free(A);
  shmem_free(B);

  shmem_finalize();
  return 0;
}
```

C

```python
from shmem4py import shmem

# shmem.init() is automatic

my_pe = shmem.my_pe()
n_pes = shmem.n_pes()

A = shmem.array(my_pe, dtype=np.int32)
B = shmem.empty(n_pes, dtype=np.int32)
shmem.fcollect(B, A)  # uses the world team by default

print(B)

shmem.free(A)
shmem.free(B)

# shmem.finalize() is automatic
```

Python

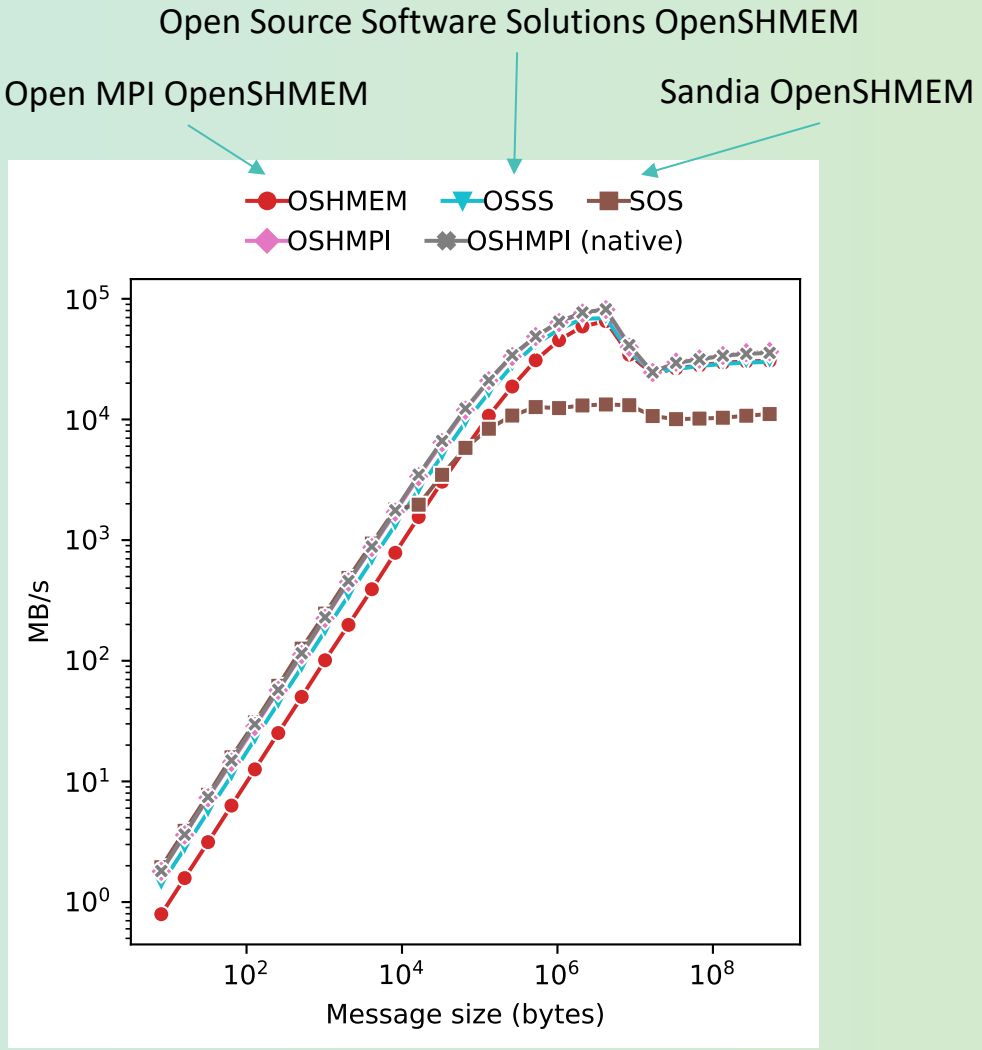
Performance evaluation

- Single-node
  - Performance differences with different OpenSHMEM backends
  - (Python vs. Numba vs. C) × (OpenSHMEM vs. MPI)
    - Memory bandwidth via a PingPong benchmark
    - Common communication patterns via Parallel Research Kernels
- Multi-node
  - shmem4py vs mpi4py

OpenSHMEM backends

- PingPong test with 2 processes on a single node
- The highest bandwidth observed during 100 repetitions
- `shmem_put`, `shmem_wait_until`

→ we use OSHMPI for single-node experiments



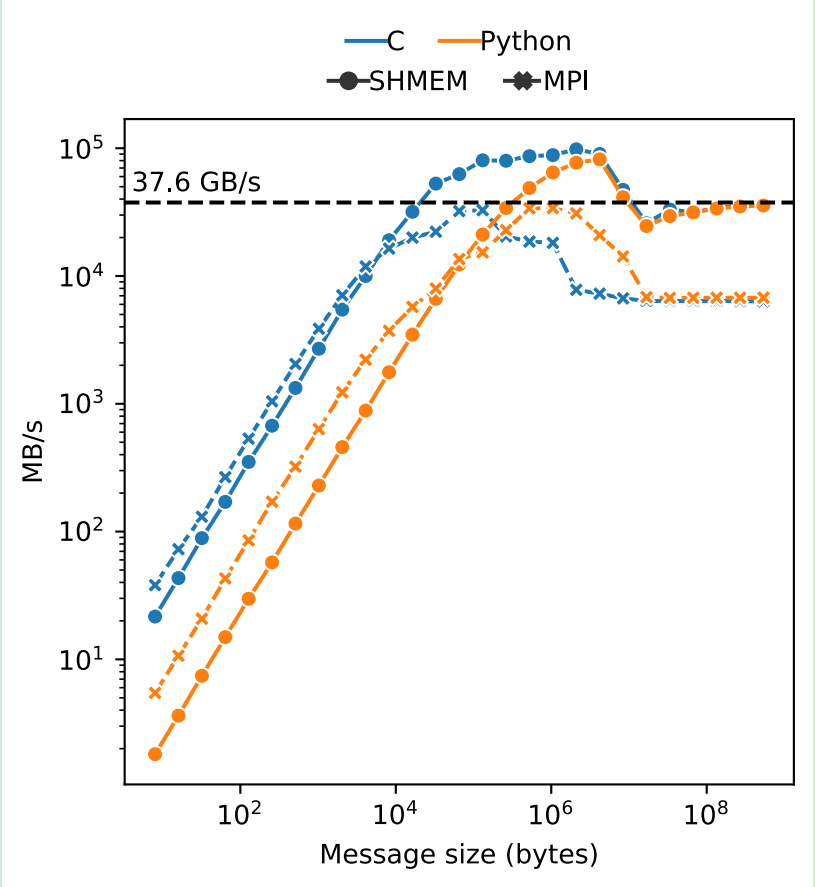
PingPong - C vs. Python and MPI vs. OpenSHMEM

- PingPong test with 2 processes on a single node
- The highest bandwidth observed during 100 repetitions

- `MPI_Send`, `MPI_Recv`
- `shmem_put`, `shmem_wait_until`

- No significant difference between C and Python implementations for large message sizes
- Increased latency for small messages
- mpi4py outperforms C+MPI for some message sizes – cache effects
- OpenSHMEM 2× faster than MPI? Single node, MPICH implementation of Send+Recv uses an intermediate buffer

- 37.6 GB/s is the `memcpy` bandwidth



Parallel Research Kernels (PRK)

- A collection of kernels supporting research on parallel computer systems
- Covers the most common patterns of communication, computation and synchronization encountered in parallel HPC applications

Selected publications:

- *R. F. Van der Wijngaart and T. G. Mattson. HPEC 2014. "The Parallel Research Kernels."*
- *R. F. Van der Wijngaart, A. Kayi, J. R. Hammond, G. Jost, T. St. John, S. Sridharan, T. G. Mattson, J. Abercrombie, and J. Nelson. ISC 2016. "Comparing runtime systems with exascale ambitions using the Parallel Research Kernels."*
- https://github.com/ParRes/Kernels

PRK	abstraction	parallel pattern	application pattern	stresses
transpose	dense matrix transpose	all-to-all comm.	global data remap, FFT, matrix transpose, sorting	memory, comm. bandwidth
synch_global	global reduction	global sync.	barrier	comm. latency
synch_p2p	2D data dependence	point-to-point sync. non-data parallelism	wave front Gauss-Seidel iteration	comm. latency
stream	streaming vector access	embarrassingly parallel	vector addition	memory bandwidth
refcount	atomic transaction	mutual exclusion	histograms	locks, atomics
reduce	vector collapse	vector reduction	reduction network	memory and comm. bandwidth
sparse	sparse matrix-vector product	unstructured grids graph analysis	data filter, sparse matrix-vector product	irregular memory access
random	random array updates	all-to-all comm.	graph processing pointer chasing	irregular memory access, comm. latency, bandwidth
stencil	stencil computation	data parallel array access, nearest-neighbor comm.	structured grids	memory bandwidth
dgemm	dense matrix-matrix product	partial broadcasts	numerical linear algebra	floating-point performance
branch	inner-loop branches	embarrassingly parallel	loop vectorization with conditionals, function calls	vectorization instruction cache

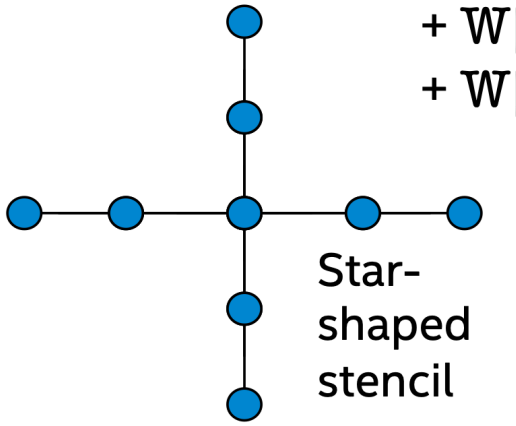
Table I
ATTRIBUTES OF ALL PARALLEL RESEARCH KERNELS

PRK Stencil kernel

- `MPI_Isend`
  `MPI_Irecv`
  `MPI_Wait`

- `shmem_put`
  `shmem_atomic_inc`
  `shmem_wait_until`

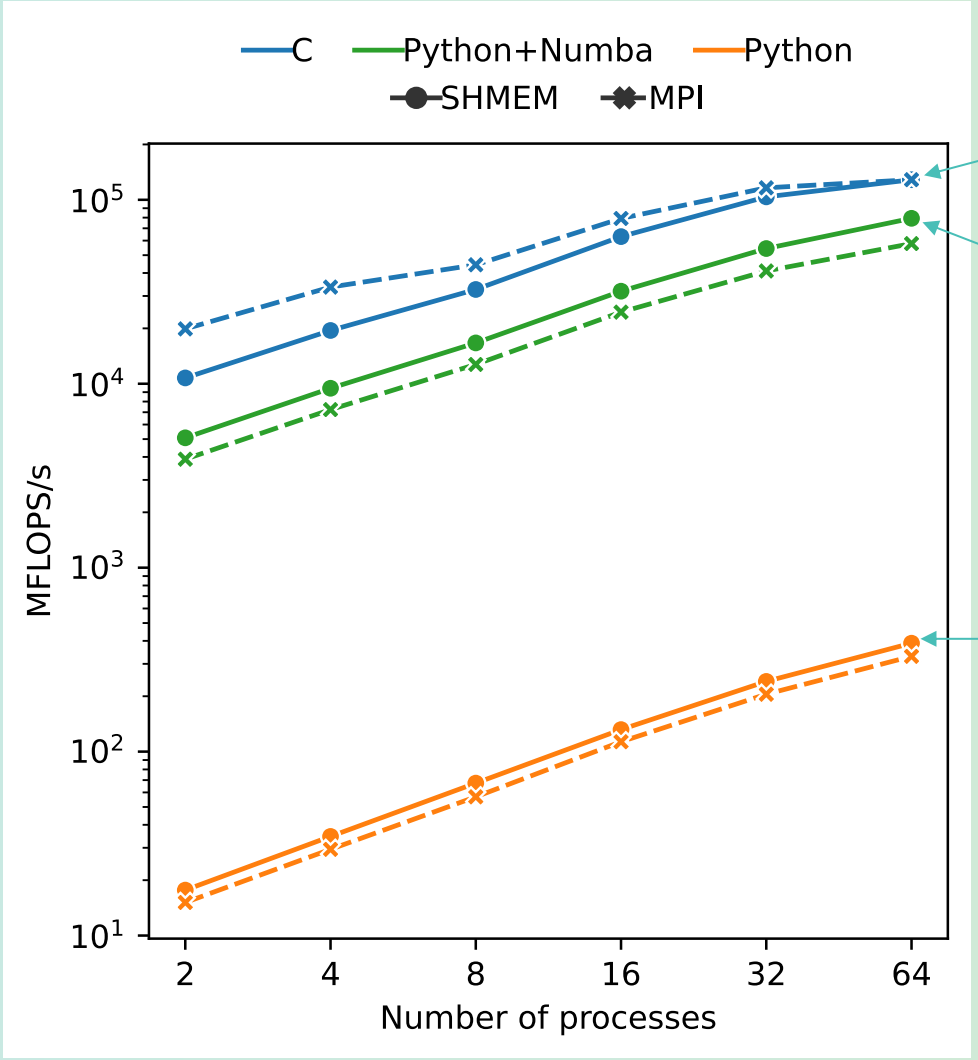
```
B[2:n-2,2:n-2] += W[2,2] * A[2:n-2,2:n-2]
                + W[2,0] * A[2:n-2,0:n-4]
                + W[2,1] * A[2:n-2,1:n-3]
                + W[2,3] * A[2:n-2,3:n-1]
                + W[2,4] * A[2:n-2,4:n-0]
                + W[0,2] * A[0:n-4,2:n-2]
                + W[1,2] * A[1:n-3,2:n-2]
                + W[3,2] * A[3:n-1,2:n-2]
                + W[4,2] * A[4:n-0,2:n-2]
```



Star-shaped stencil

Single-node Stencil performance

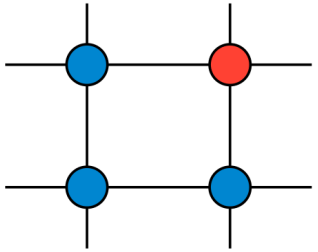
- Numba variant JIT compiles the triple nested loop containing the stencil update



PRK Synch_p2p kernel

- `MPI_Send`
  `MPI_Recv`
- `shmem_put`
  `shmem_wait_until`

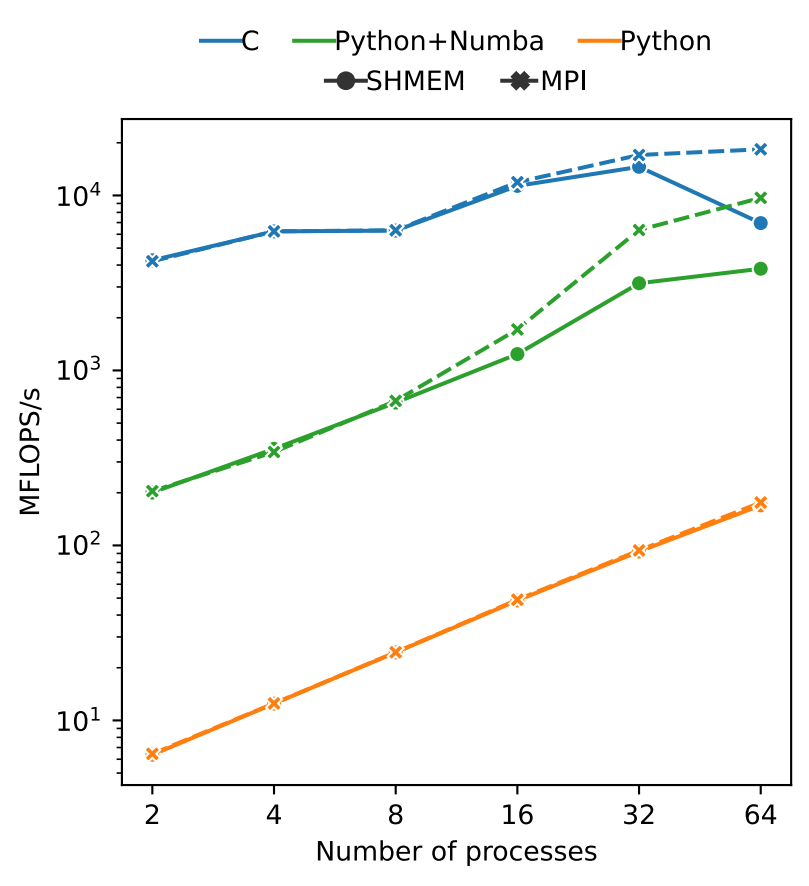
```
for i in range(1,m):
      for j in range(1,n):
            grid[i][j]     = grid[i-1][j]
                           + grid[i][j-1]
                           - grid[i-1][j-1]

grid[0][0] = -grid[m-1][n-1]
```



$A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}$

Single-node Synch_p2p performance

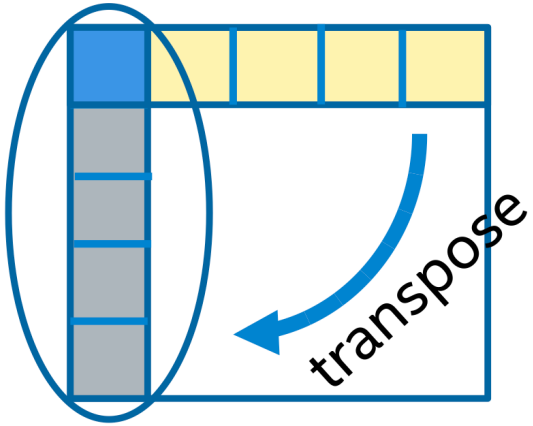
- Numba JIT compiles one loop, still called within a Python loop



PRK Transpose kernel

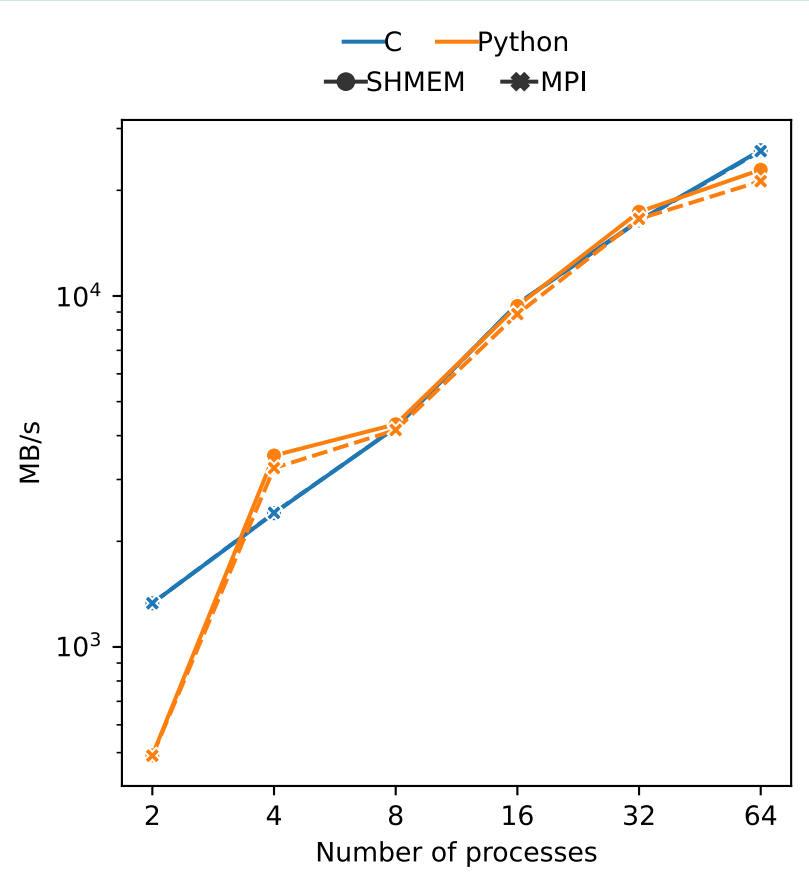
- `MPI_Alltoall`
- `shmem_alltoall`

```
for i in range(order):
    for j in range(order):
        B[i][j] += A[j][i]
        A[j][i] += 1.0
```



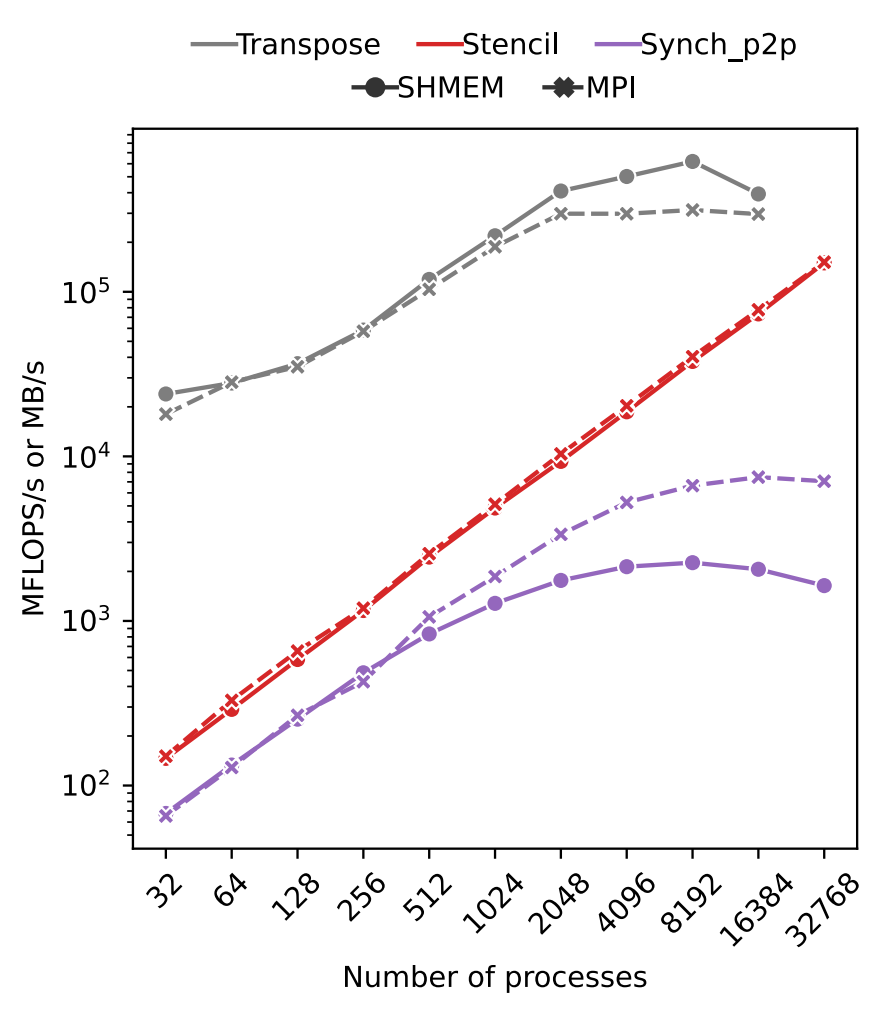
Single-node Transpose kernel performance

- Numba doesn't outperform NumPy transpose
- OSHMPI shmem_alltoall is MPI's MPI_Alltoall



Multi-node PRK

- Cray XC40 with Cray OpenSHMEMX, Cray MPICH
- Fully subscribed, 32 processes per node
- 1—1024 nodes



Work in progress and Future work

- Reductions and atomics APIs:
  - `shmem.reduce(target, source, MAX)`
  - `shmem.max_reduce(target, source)`
  - `shmem.atomic_op(target, value, ADD)`
  - `shmem.atomic_add(target, value)`
- Forced use of 1-sized slices:
  - `shmem.test(wait_vars[idx:idx+1], shmem.CMP.NE, 0)` – NumPy array slices are mutable
  - `shmem.test(wait_vars[idx], shmem.CMP.NE, 0)` – individual array elements are immutable
- Future work:
  - NVSHMEM
  - rocSHMEM

Disclaimers and Conclusions

- Our experiments **do not**:
  - Take advantage of the unique properties of OpenSHMEM
  - Compare OpenSHMEM to MPI RMA (one-sided)
- Our experiments **do**:
  - Validate the design
  - Ensure that the overheads from the Python wrappers are negligible
  - Show that common algorithm patterns are straightforward to implement
- Results are mixed:
  - OpenSHMEM up to 2.0× faster than MPI
  - MPI up to 4.3× faster than OpenSHMEM
- **Unremarkable differences → choose best programming model based on application needs**

Thank you