UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

Parallel Algorithm Design: UPC++

Ferdinand Vanmaele

Overview

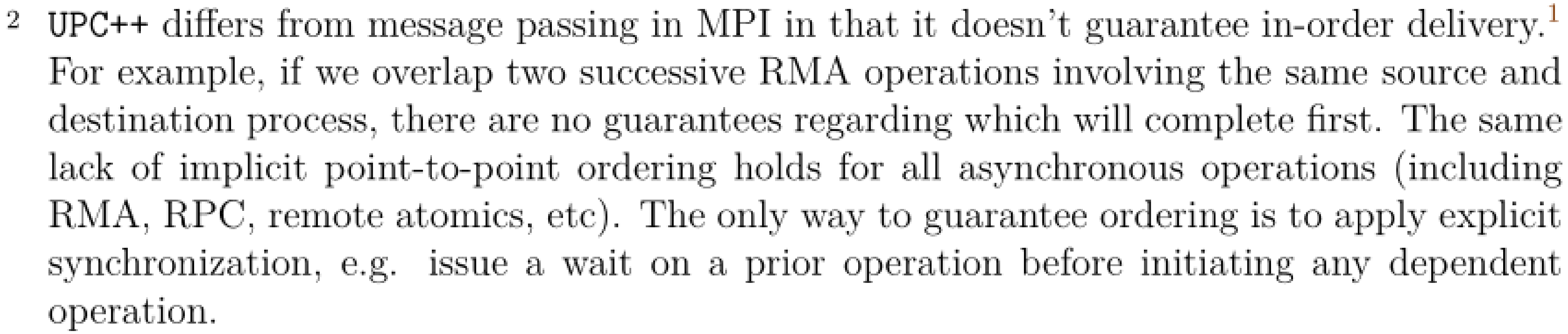
- Introduction to UPC++
  - Partitioned Global Address Space
- Reduction
  - Distributed objects
- Symmetrisation
  - Avoiding communication
- Stencil

Introduction to UPC++

- C++11 library that supports *Partitioned Global Address Space* (PGAS) programming
    - Similar to OpenMPI in usage (but no in-order delivery of messages)

Introduction to UPC++

- C++11 library that supports *Partitioned Global Address Space* (PGAS) programming
  - Similar to OpenMPI in usage (but no in-order delivery of messages)



[2] UPC++ differs from message passing in MPI in that it doesn't guarantee in-order delivery.[1] For example, if we overlap two successive RMA operations involving the same source and destination process, there are no guarantees regarding which will complete first. The same lack of implicit point-to-point ordering holds for all asynchronous operations (including RMA, RPC, remote atomics, etc). The only way to guarantee ordering is to apply explicit synchronization, e.g. issue a wait on a prior operation before initiating any dependent operation.

https://bitbucket.org/berkeleylab/upcxx/downloads/upcxx-spec-2020.10.0.pdf (1.3 Memory model)

Introduction to UPC++

- C++11 library that supports *Partitioned Global Address Space* (PGAS) programming
  - Usage similar to OpenMPI (but no in-order delivery of messages)
- *Single program, multiple data* (SPMD) model via *processes*
  - Global address space
  - No implicit communication
  - Fixed number of processes
- Multiple *conduits*: SMP (sockets), UDP (ethernet), Infiniband, Cray
  - Here limited to SMP and UDP

Partitioned Global Address Space

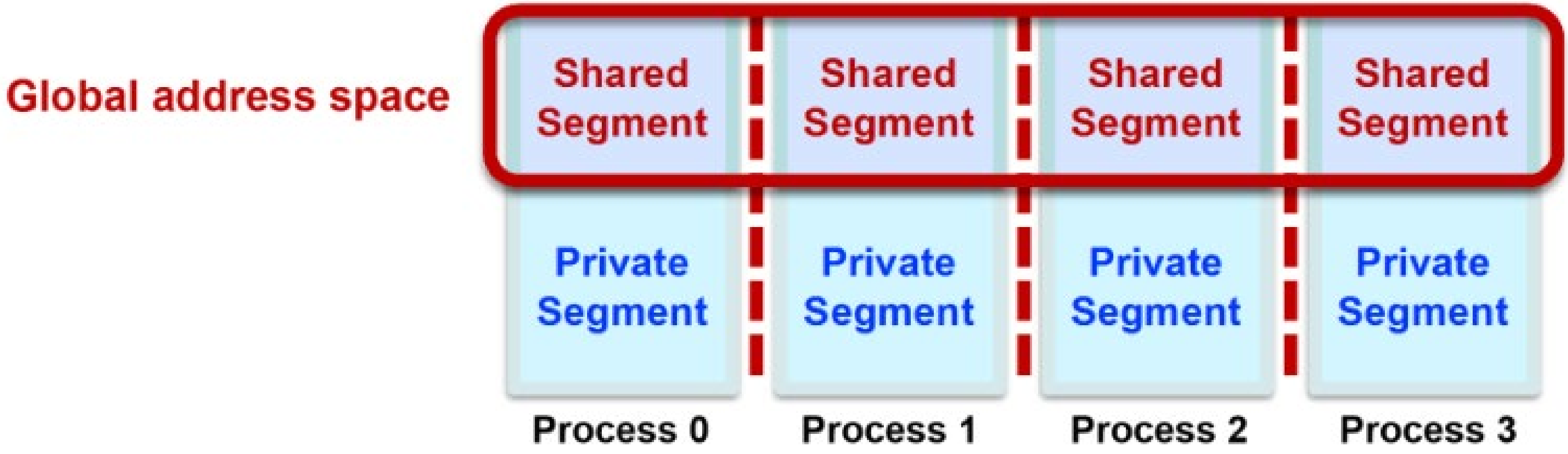
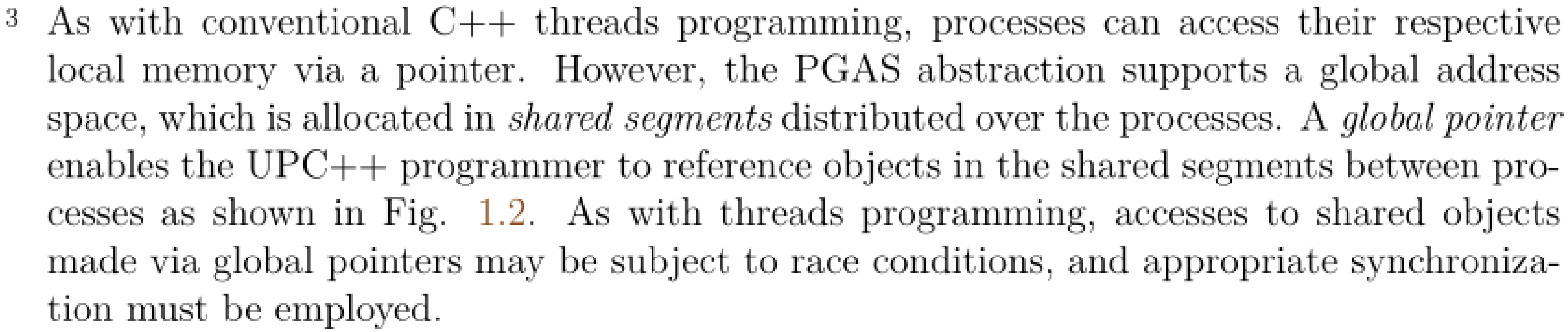


Figure 1.1: Abstract Machine Model of a PGAS program memory

Partitioned Global Address Space

- Shared segment accessible through **upcxx::global_ptr<T>**
  - Logically consists of raw pointer and an *affinity* (binding of a location in a *shared segment* to a process)
  - Requires synchronization (subject to race conditions)



[3] As with conventional C++ threads programming, processes can access their respective local memory via a pointer. However, the PGAS abstraction supports a global address space, which is allocated in *shared segments* distributed over the processes. A *global pointer* enables the UPC++ programmer to reference objects in the shared segments between processes as shown in Fig. 1.2. As with threads programming, accesses to shared objects made via global pointers may be subject to race conditions, and appropriate synchronization must be employed.

https://bitbucket.org/berkeleylab/upcxx/downloads/upcxx-spec-2020.10.0.pdf (1.1 Preliminaries)

Partitioned Global Address Space

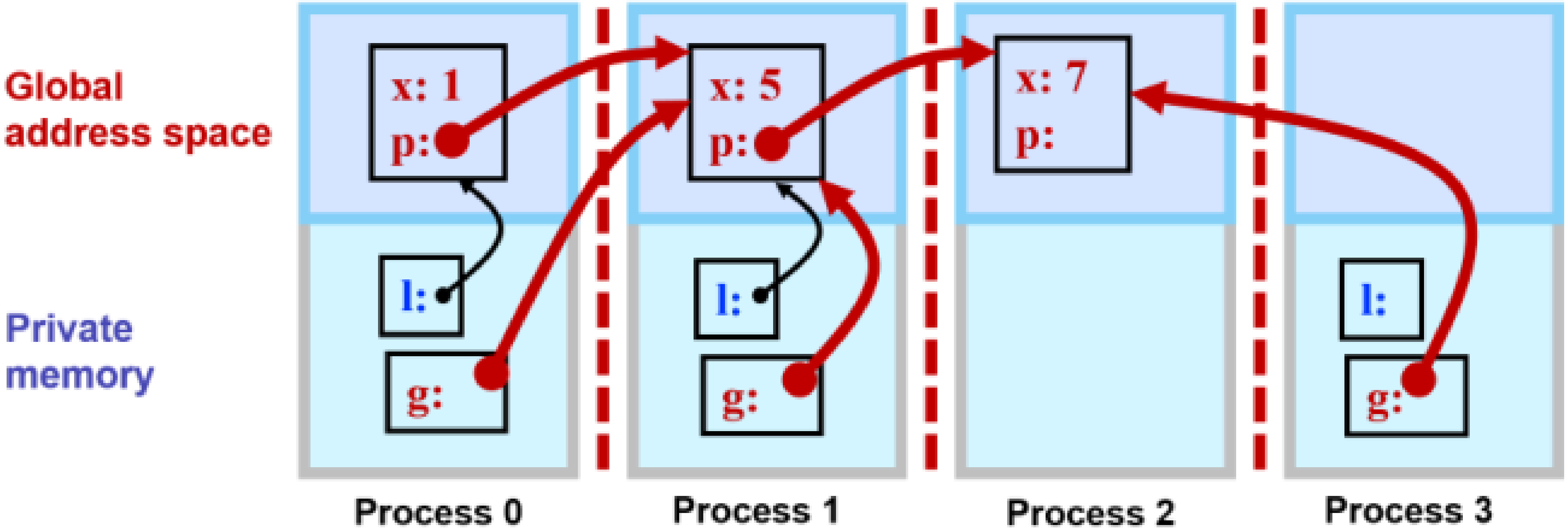
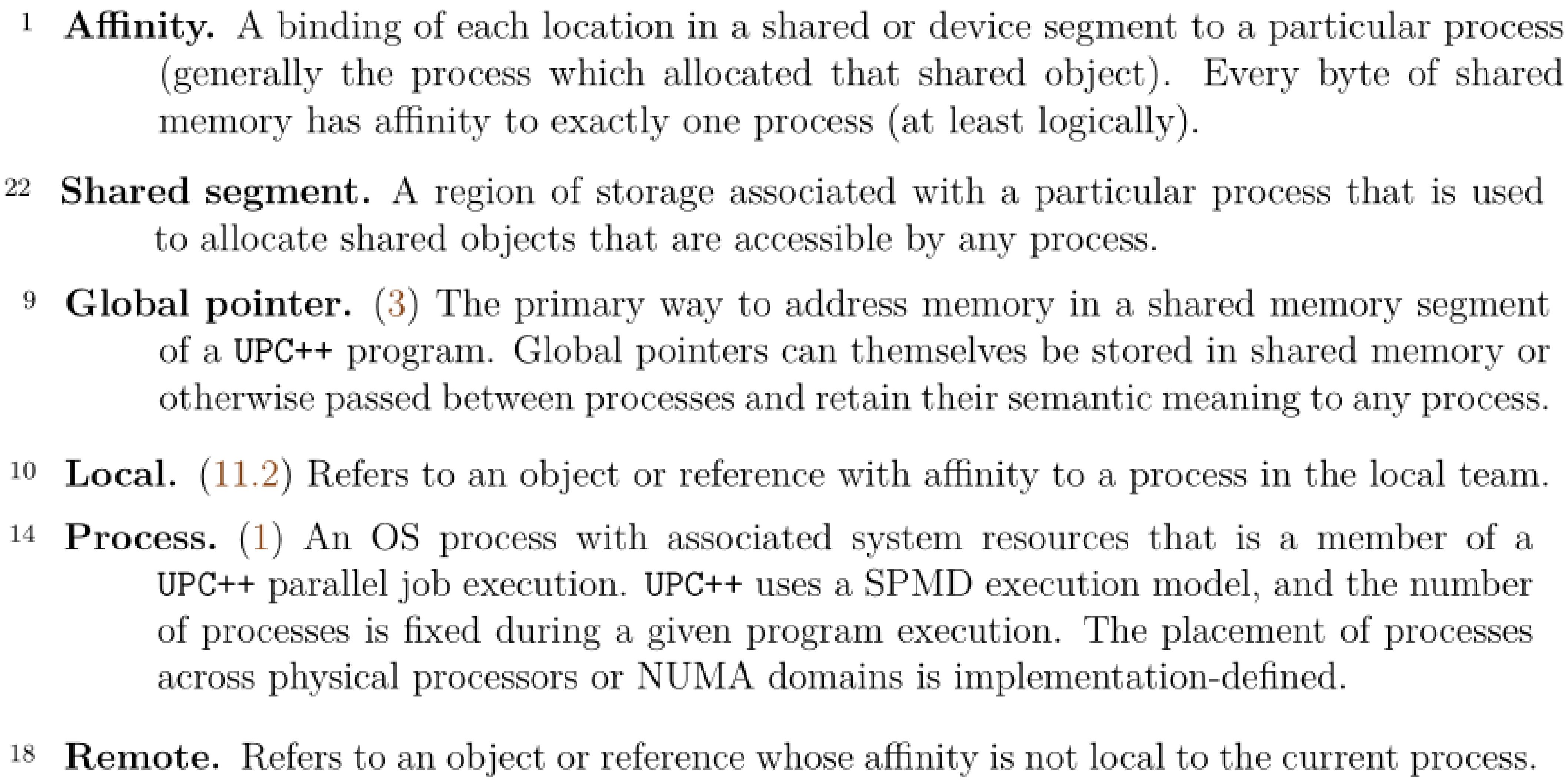


Figure 1.2: Global pointers and shared memory objects in a PGAS model

Overview of important terminology (1)



Affinity. A binding of each location in a shared or device segment to a particular process (generally the process which allocated that shared object). Every byte of shared memory has affinity to exactly one process (at least logically).

Shared segment. A region of storage associated with a particular process that is used to allocate shared objects that are accessible by any process.

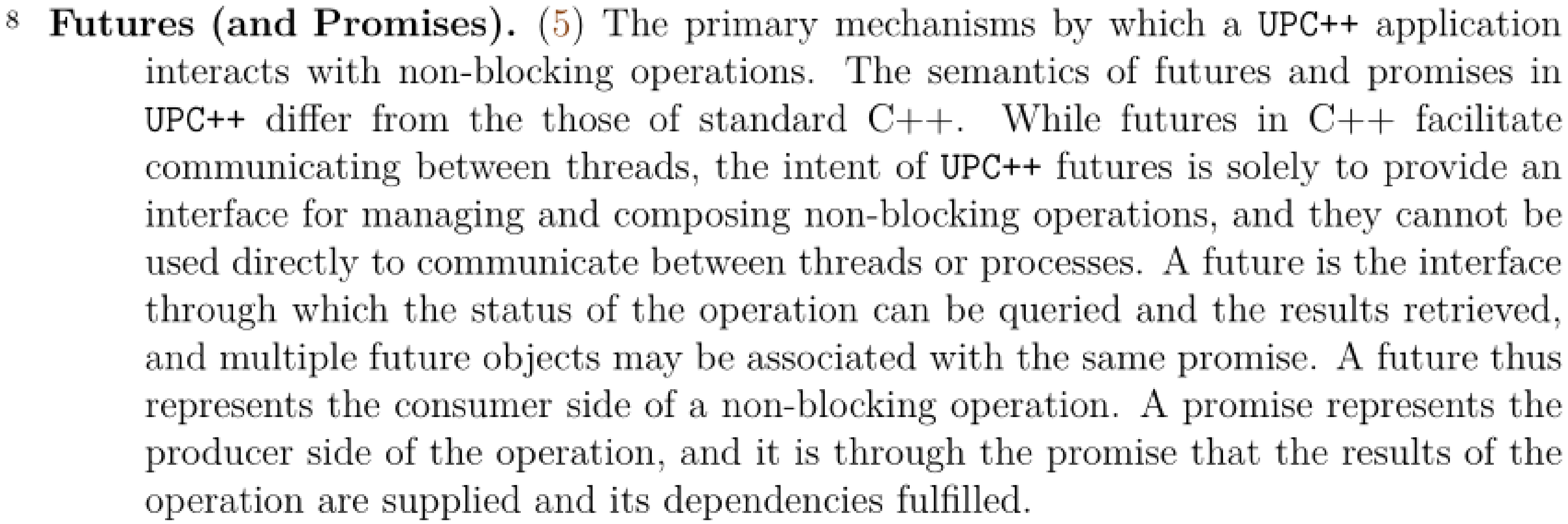
Global pointer. (3) The primary way to address memory in a shared memory segment of a `UPC++` program. Global pointers can themselves be stored in shared memory or otherwise passed between processes and retain their semantic meaning to any process.

Local. (11.2) Refers to an object or reference with affinity to a process in the local team.

Process. (1) An OS process with associated system resources that is a member of a `UPC++` parallel job execution. `UPC++` uses a SPMD execution model, and the number of processes is fixed during a given program execution. The placement of processes across physical processors or NUMA domains is implementation-defined.

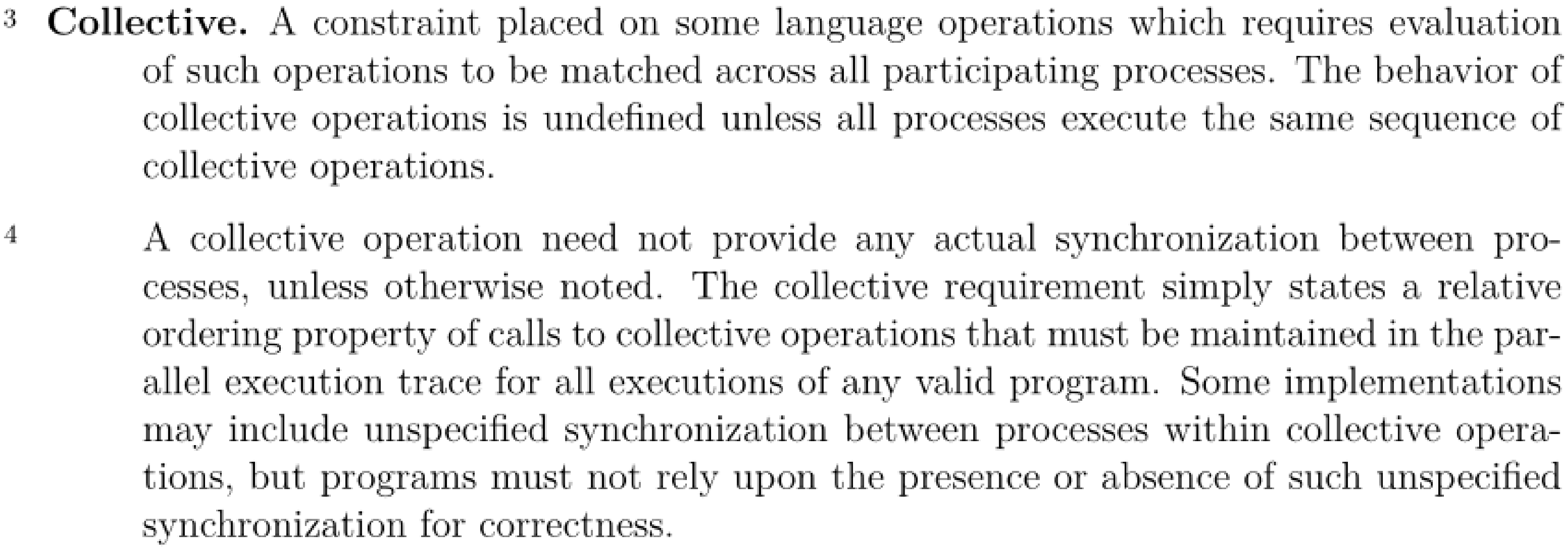
Remote. Refers to an object or reference whose affinity is not local to the current process.

Overview of important terminology (2)



Futures (and Promises). (5) The primary mechanisms by which a `UPC++` application interacts with non-blocking operations. The semantics of futures and promises in `UPC++` differ from the those of standard C++. While futures in C++ facilitate communicating between threads, the intent of `UPC++` futures is solely to provide an interface for managing and composing non-blocking operations, and they cannot be used directly to communicate between threads or processes. A future is the interface through which the status of the operation can be queried and the results retrieved, and multiple future objects may be associated with the same promise. A future thus represents the consumer side of a non-blocking operation. A promise represents the producer side of the operation, and it is through the promise that the results of the operation are supplied and its dependencies fulfilled.

Overview of important terminology (3)

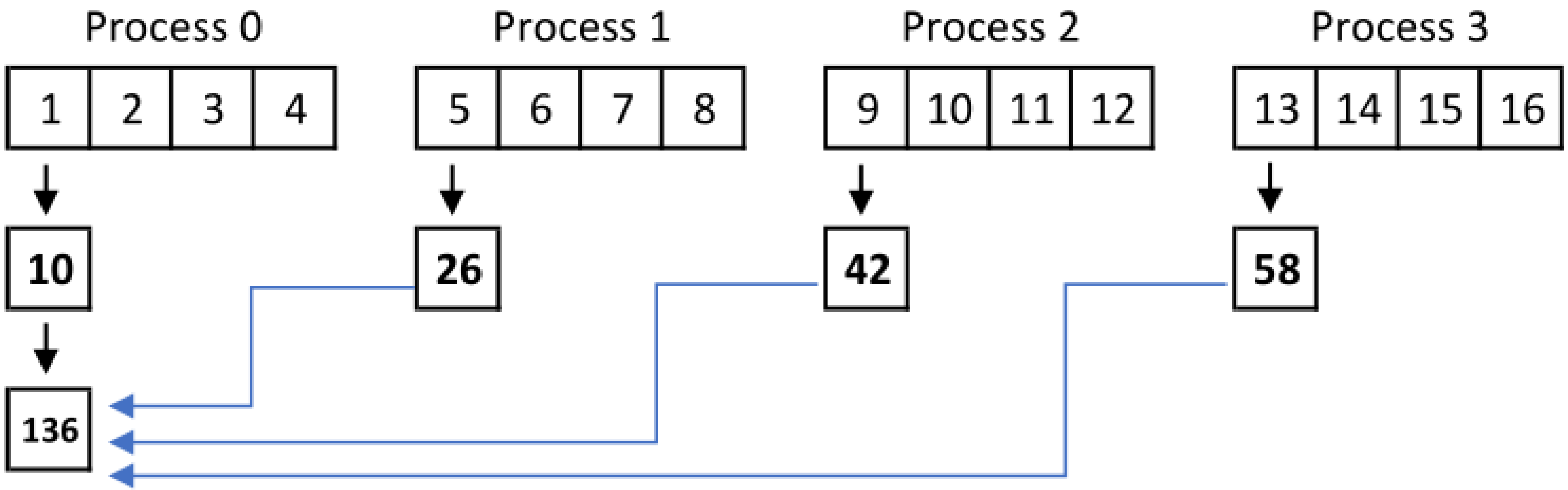


3 **Collective.** A constraint placed on some language operations which requires evaluation of such operations to be matched across all participating processes. The behavior of collective operations is undefined unless all processes execute the same sequence of collective operations.

4 A collective operation need not provide any actual synchronization between processes, unless otherwise noted. The collective requirement simply states a relative ordering property of calls to collective operations that must be maintained in the parallel execution trace for all executions of any valid program. Some implementations may include unspecified synchronization between processes within collective operations, but programs must not rely upon the presence or absence of such unspecified synchronization for correctness.

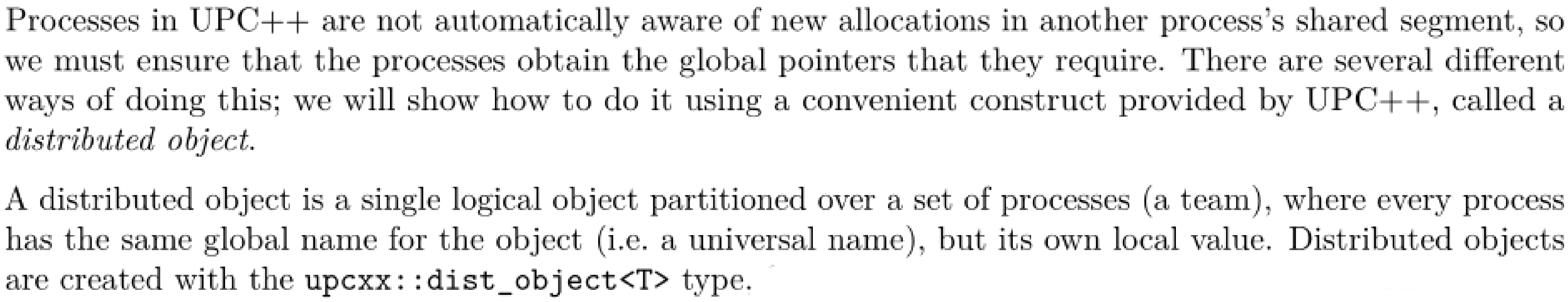
Reduction

- Basic hierarchical reduction



Reduction

- Basic hierarchical reduction
- Reduce partial sums with **upcxx::dist_object<T>**
  - Order is determined by the user (e.g. equivalent to serial order)



Processes in UPC++ are not automatically aware of new allocations in another process's shared segment, so we must ensure that the processes obtain the global pointers that they require. There are several different ways of doing this; we will show how to do it using a convenient construct provided by UPC++, called a *distributed object*.

A distributed object is a single logical object partitioned over a set of processes (a team), where every process has the same global name for the object (i.e. a universal name), but its own local value. Distributed objects are created with the `upcxx::dist_object<T>` type.

https://bitbucket.org/berkeleylab/upcxx/downloads/upcxx-guide-2020.10.0.pdf (Section 5)

```cpp
// Compute partial sums
double psum(0);
for (int64_t i = 0; i < block_size; ++i) {
    psum += u[i];
}

// Assign partial sum to distributed object (universal name, local value)
upcxx::dist_object<double> psum_d(psum);

// Reduce partial sums with dist_object::fetch (communication) on first process.
if (proc_id == 0) {
    double res(*psum_d);

    // Partial sums for remaining processes (in ascending order).
    // Can be done asynchronously, e.g. with .then() and upcxx::when_all().
    for (int k = 1; k < upcxx::rank_n(); ++k) {
        double psum = psum_d.fetch(k).wait();
        res += psum;
    }
}
```

Reduction

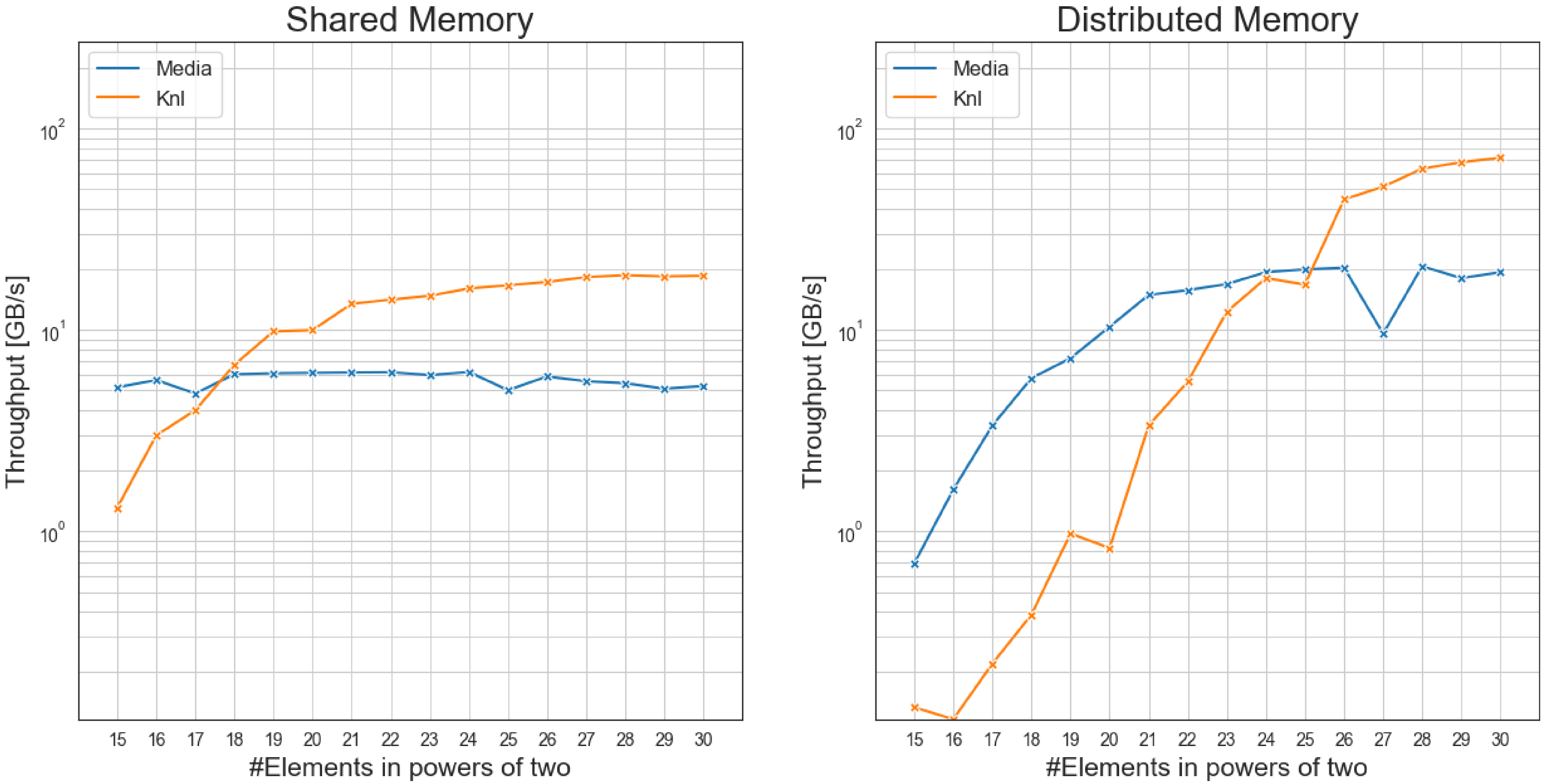
- Basic hierarchical reduction
- Reduce partial sums with **upcxx::dist_object<T>**
  - Order is determined by the user (e.g. equivalent to serial order)
- Reduce partial sums with **upcxx::reduce_one**
  - Non-deterministic order
  - Simple usage

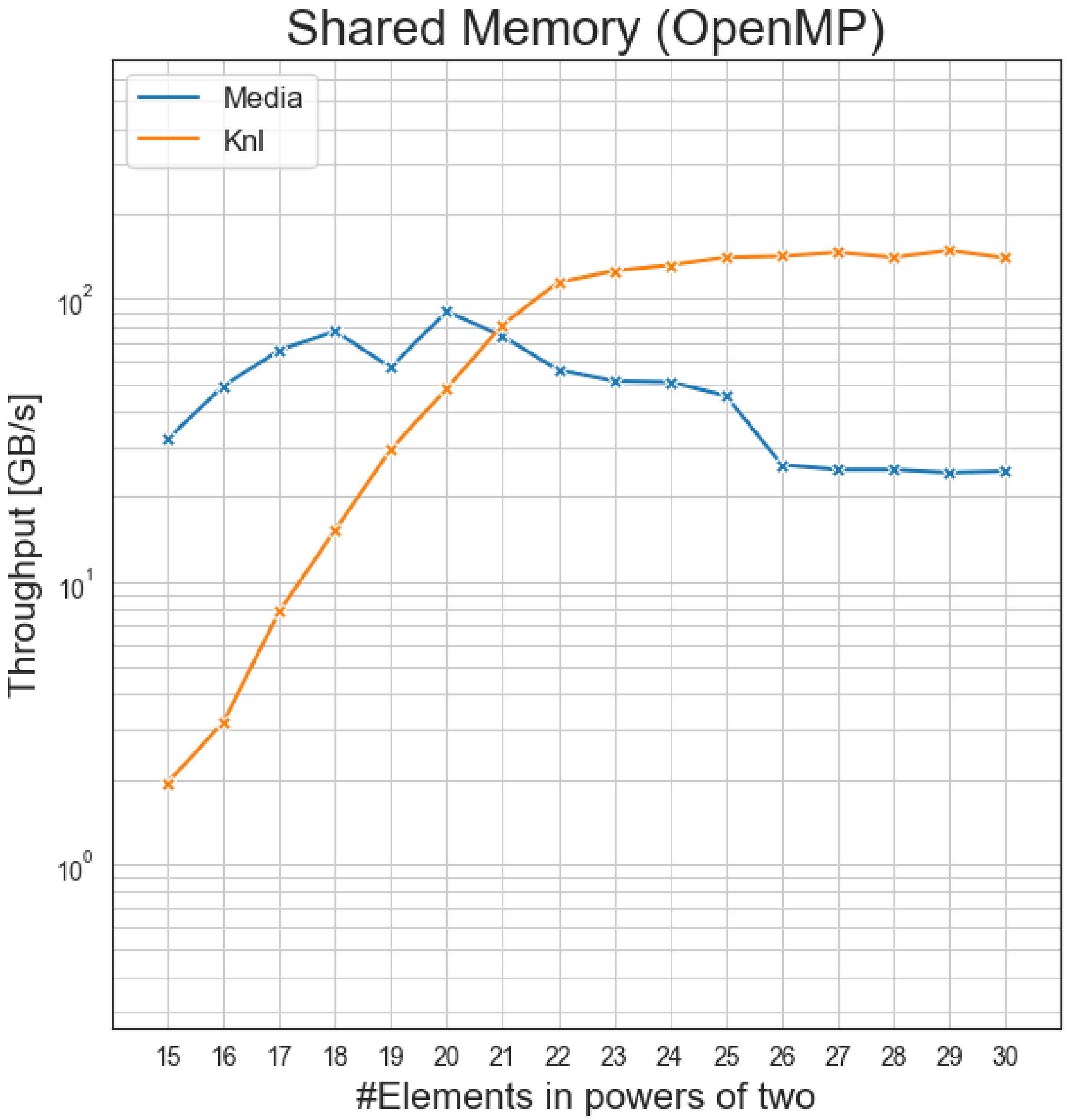
Reduction

```
// Create a reduction value for each process
double psum(0);

// Compute partial sums
for (int64_t i = 0; i < block_size; ++i) {
    psum += u[i];
}
double sum = upcxx::reduce_one(psum, upcxx::op_fast_add, 0).wait();
```

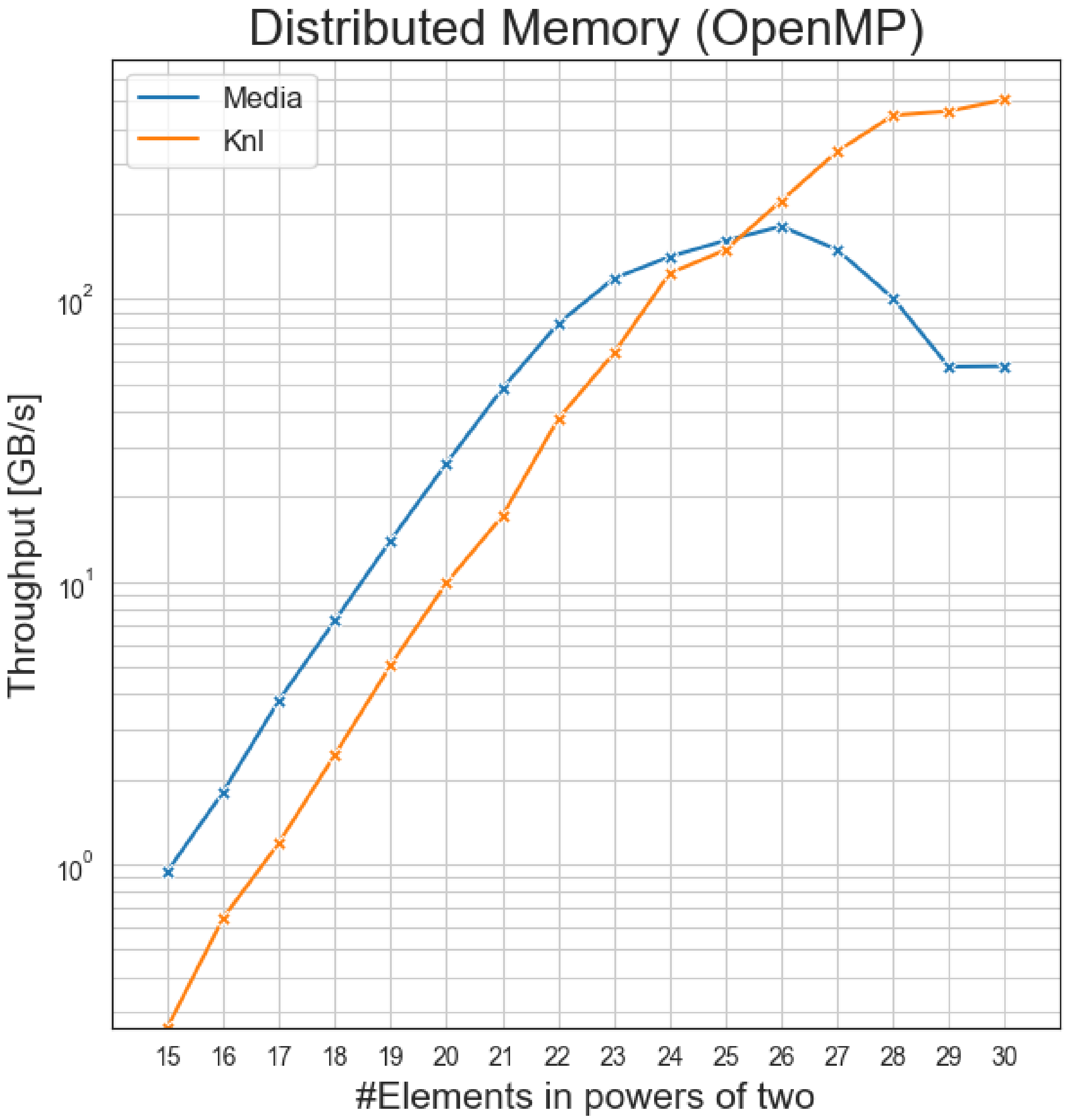
Reduction

- Throughput
  - **size * sizeof(float) * 1e-9 / time**
- Benchmarks
  - **upcxx::reduce_one**
  - 4 processes per SKL node, 64 processes per KNL node
  - Averaged over 100 iterations
- OpenMP
  - 1 process per node, 4 (SKL) or 64 (KNL) OpenMP threads
  - *OMP_PROC_BIND=true*, *OMP_PLACES=cores* (NUMA-aware)
  - Higher throughput from lower amount of communication


Shared Memory
Distributed Memory
Media
Knl
Throughput [GB/s]
10^2
10^1
10^0
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
#Elements in powers of two



Shared Memory (OpenMP)



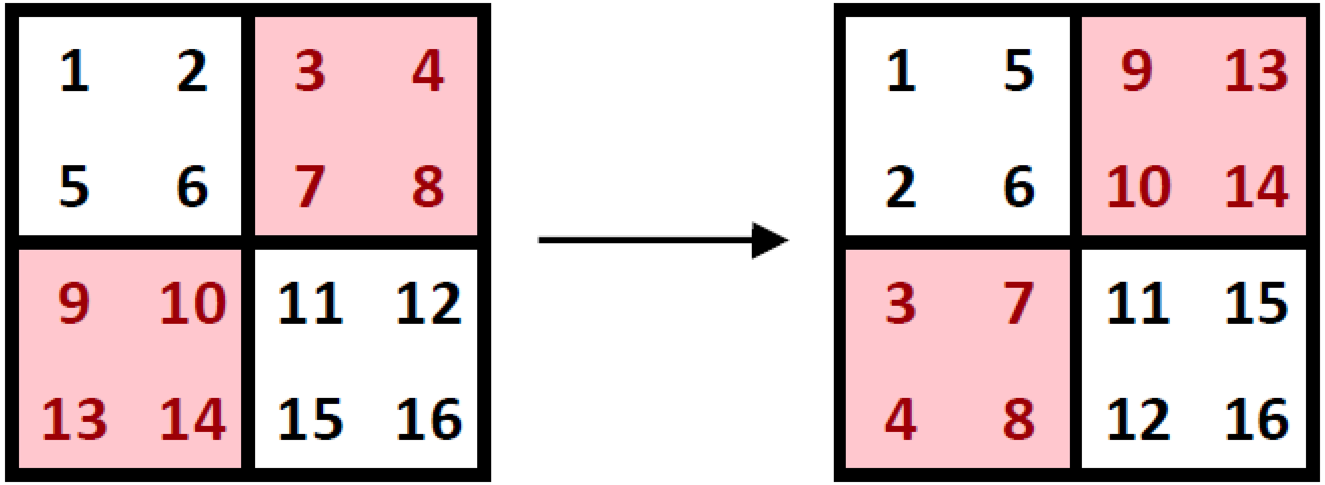
Distributed Memory (OpenMP)

Symmetrisation

- Data structures for dense matrix
  - Row-major (or col-major)
  - Block matrix for partitioning

Symmetrisation

- Data structures for dense matrix
  - Row-major (or col-major)
  - Block matrix for partitioning
    - Communication between blocks for transposition



$$\left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right) \longrightarrow \left(\begin{array}{cc|cc} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ \hline 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{array}\right)$$

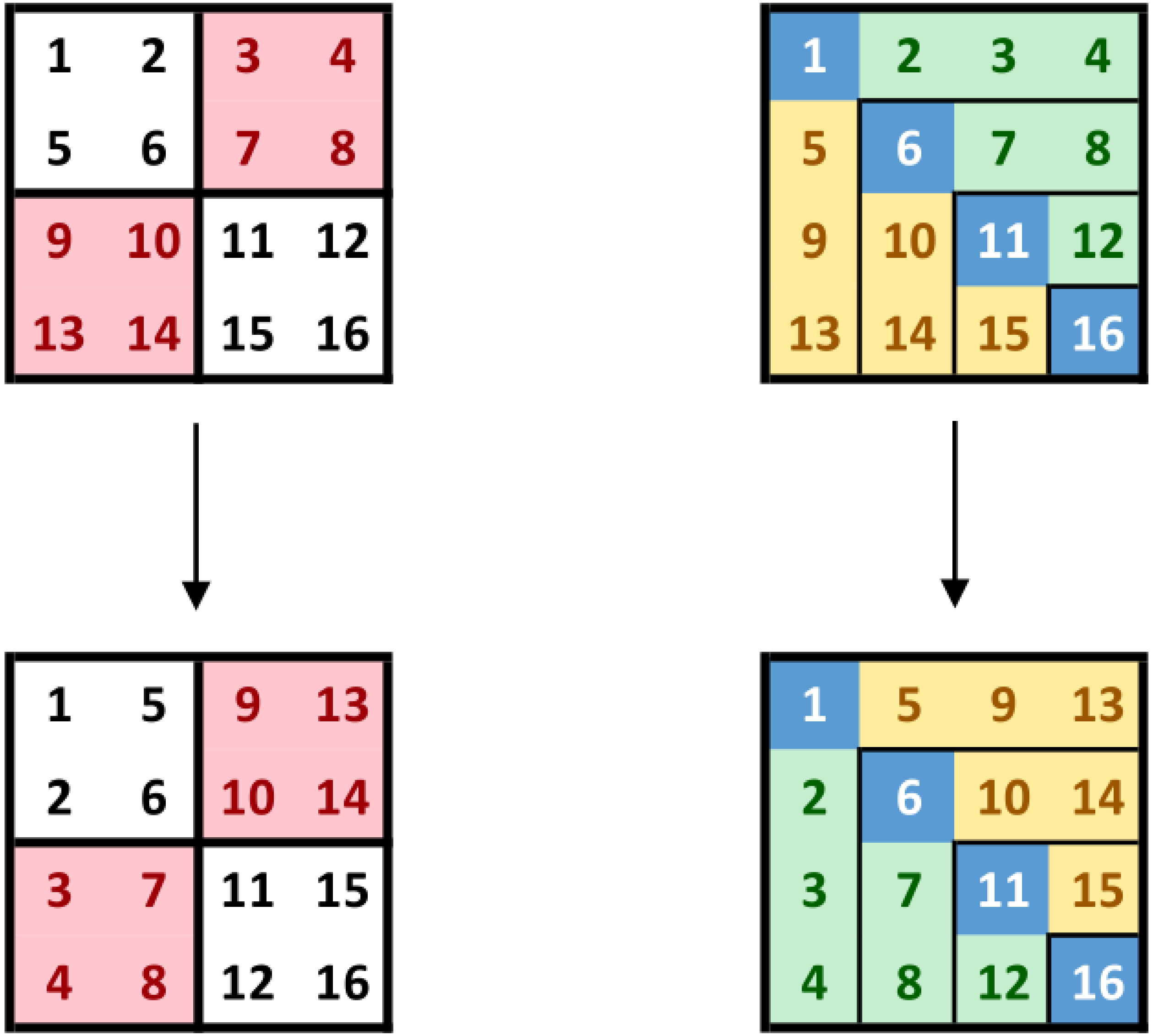
Symmetrisation

- Data structures for dense matrix
  - Store diagonal, lower and upper matrix separately
  - Store elements symmetrically
    - Lower triangle in *col-major* order
    - Upper triangle in *row-major* order
- **Symmetrisation as SAXPY operation**

Symmetrisation

- Data structures for dense matrix
  - Store diagonal, lower and upper matrix separately
  - Store elements symmetrically
    - Lower triangle in *col-major* order
    - Upper triangle in *row-major* order
- **Symmetrisation as SAXPY operation**
  - Arrays chunked per process:
    - *Diagonal*: dim / nproc
    - *Lower triangle*: (dim) * (dim-1) / 2 / nproc
    - *Upper triangle*: (dim) * (dim-1) / 2 / nproc

Symmetrisation



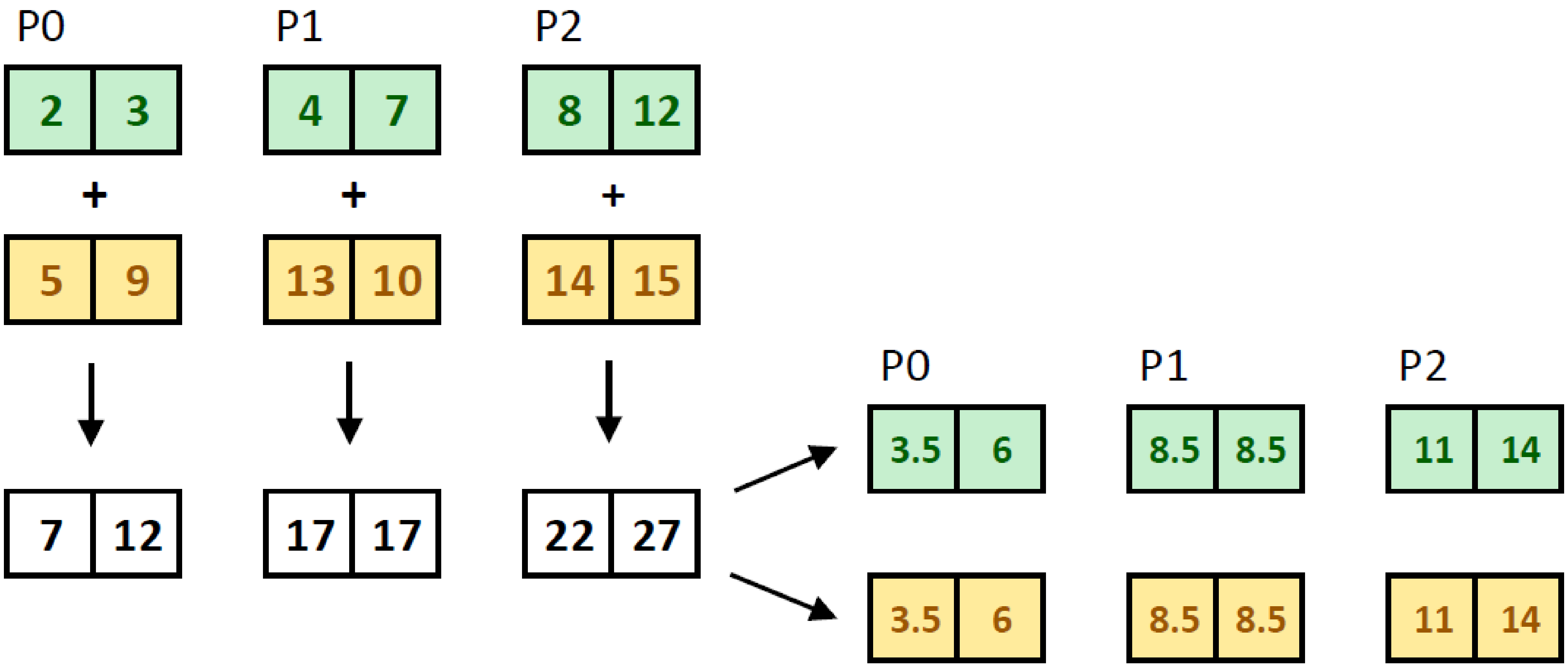
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Symmetrisation



P0 P1 P2

2 3 4 7 8 12

+ + +

5 9 13 10 14 15

7 12 17 17 22 27

P0 P1 P2

3.5 6 8.5 8.5 11 14

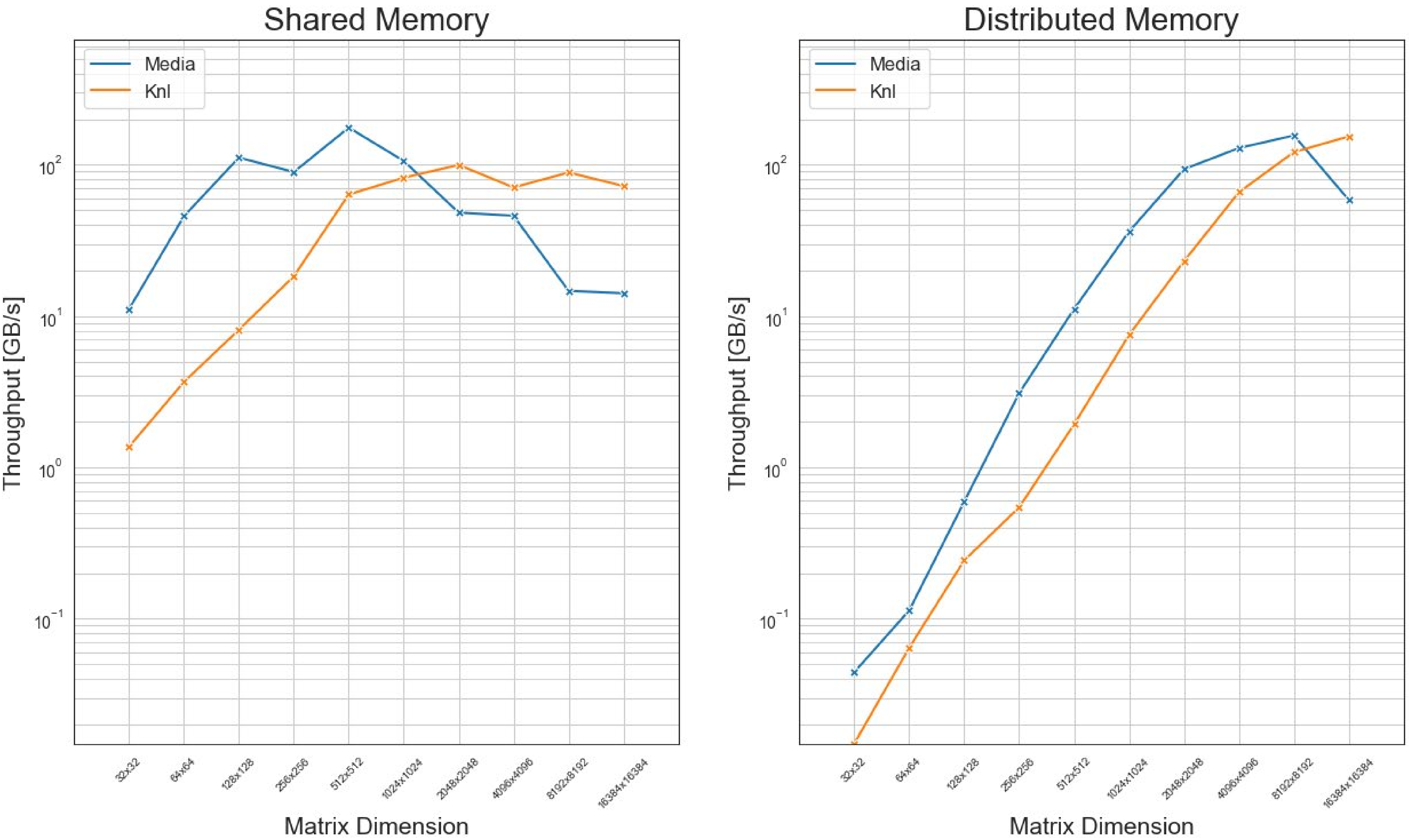
3.5 6 8.5 8.5 11 14

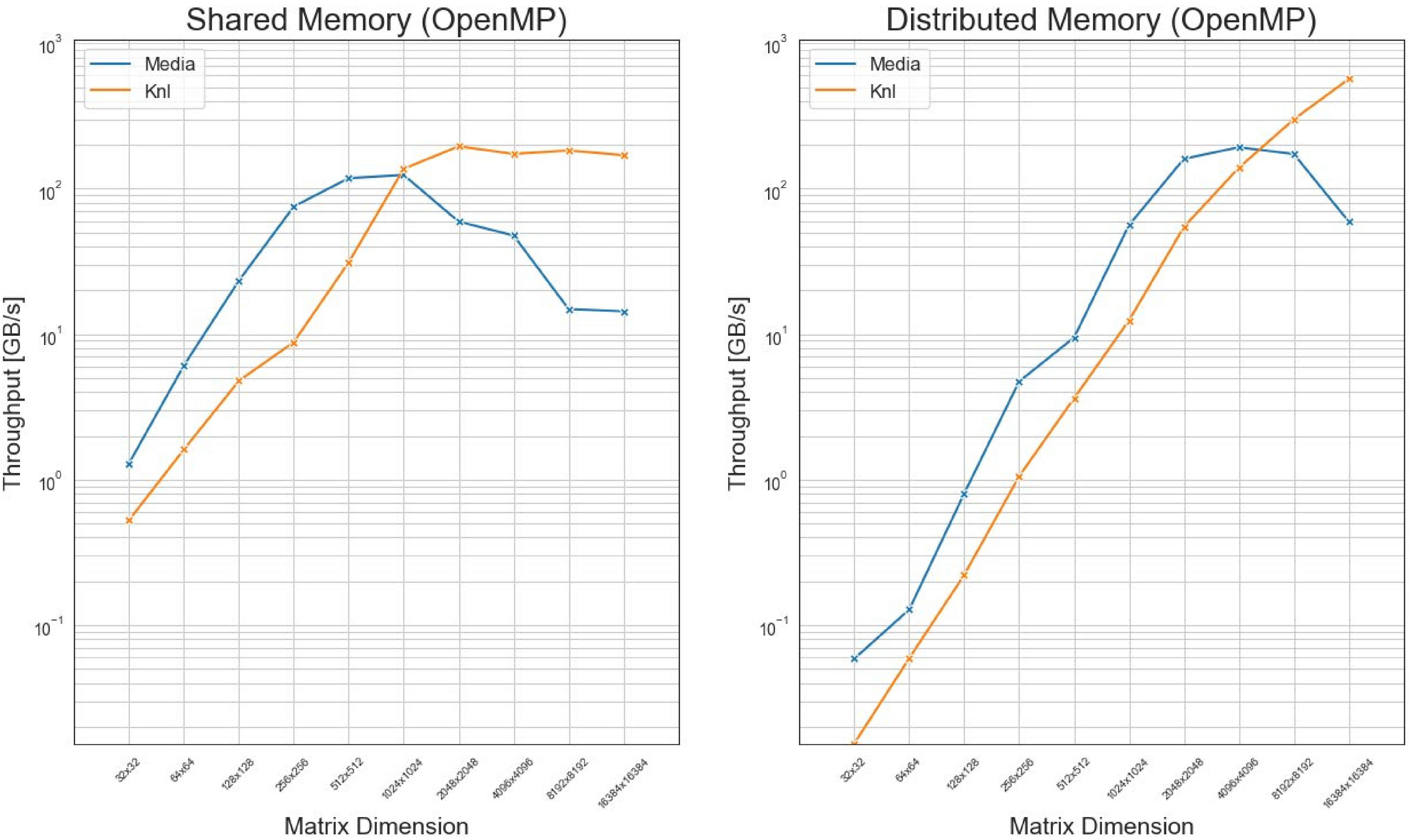
Symmetrisation

```
// Symmetrize matrix (SAXPY over lower and upper triangle). We only require
// a single for loop because lower and upper triangle are stored symmetrically
// (in col-major and row-major, respectively)
for (index_t i = 0; i < triangle_n; ++i) {
    float s = (lower[i] + upper[i]) / 2.;
    lower[i] = s;
    upper[i] = s;
}
upcxx::barrier(); // ensure symmetrisation is complete
```

Symmetrisation

- Throughput
    - **dim * (dim-1) * sizeof(float) * 1e-9 / time**
- Benchmarks
    - Process amount chosen as size allowed
        - e.g. for the KNL cluster, 256 processes for sizes (1 << 10) up to (1 << 14).
    - No communication
- OpenMP
    - 1 process per node with up to 64 (KNL) OpenMP threads per node
    - Still measurable improvement from lower amount of processes

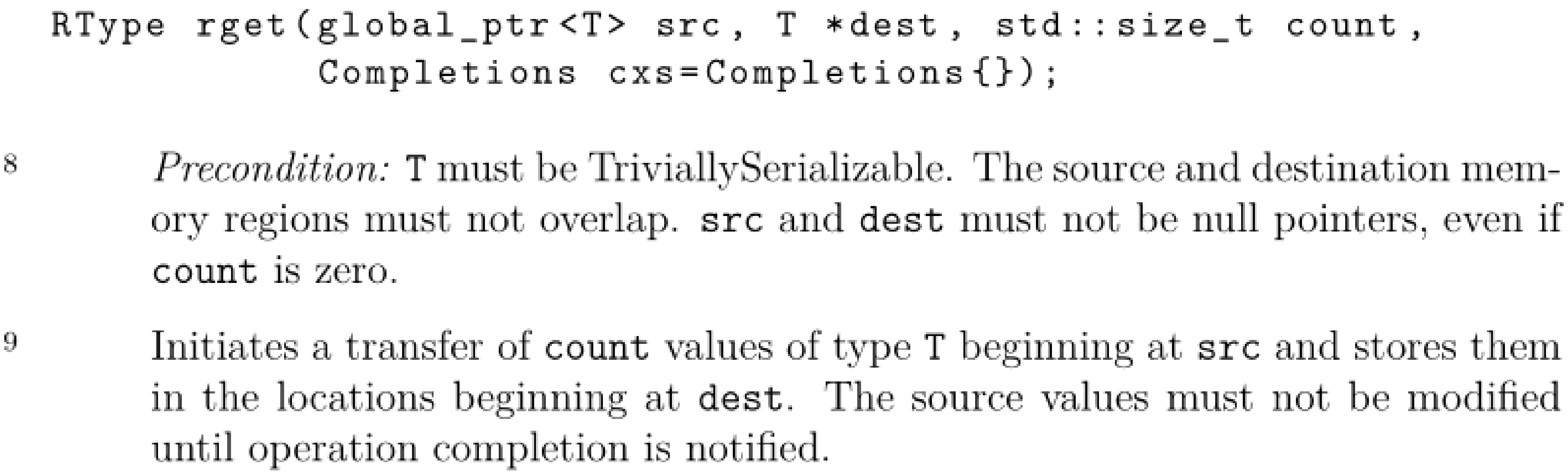

Shared Memory
Media
Knl
Throughput [GB/s]
10^{-1}
10^{0}
10^{1}
10^{2}
32x32
64x64
128x128
256x256
512x512
1024x1024
2048x2048
4096x4096
8192x8192
16384x16384
Matrix Dimension
Distributed Memory
Media
Knl
Throughput [GB/s]
10^{-1}
10^{0}
10^{1}
10^{2}
32x32
64x64
128x128
256x256
512x512
1024x1024
2048x2048
4096x4096
8192x8192
16384x16384
Matrix Dimension



Shared Memory (OpenMP)
Media
Knl
Throughput [GB/s]
10^3
10^2
10^1
10^0
10^{-1}
32x32
64x64
128x128
256x256
512x512
1024x1024
2048x2048
4096x4096
8192x8192
16384x16384
Matrix Dimension
Distributed Memory (OpenMP)
Media
Knl
Throughput [GB/s]
10^3
10^2
10^1
10^0
10^{-1}
32x32
64x64
128x128
256x256
512x512
1024x1024
2048x2048
4096x4096
8192x8192
16384x16384
Matrix Dimension

Stencil

- Implementation using array padding and ghost cells
  - Fetch left and right neighbours separately
  - Synchronize twice in each step



```
RType rget(global_ptr<T> src, T *dest, std::size_t count,
         Completions cxs=Completions{});
```

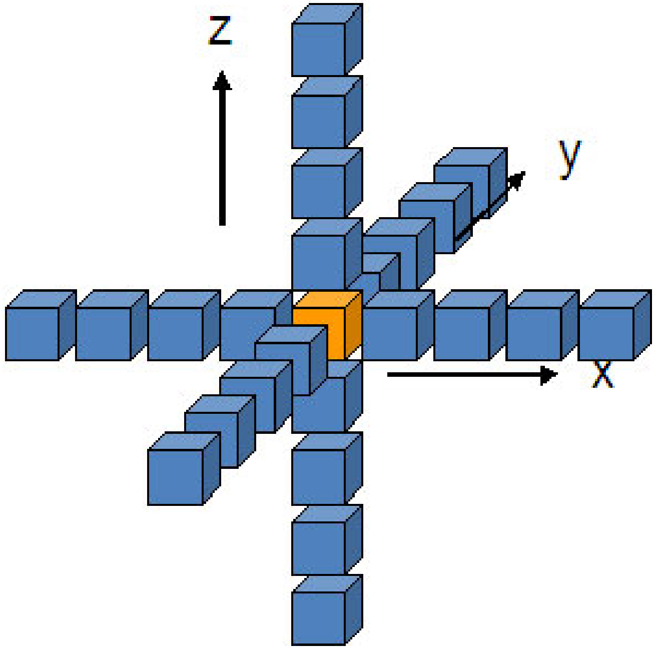
8 *Precondition:* `T` must be TriviallySerializable. The source and destination memory regions must not overlap. `src` and `dest` must not be null pointers, even if `count` is zero.

9 Initiates a transfer of `count` values of type `T` beginning at `src` and stores them in the locations beginning at `dest`. The source values must not be modified until operation completion is notified.

https://bitbucket.org/berkeleylab/upcxx/downloads/upcxx-spec-2020.10.0.pdf (8.2.2 Remote Gets)

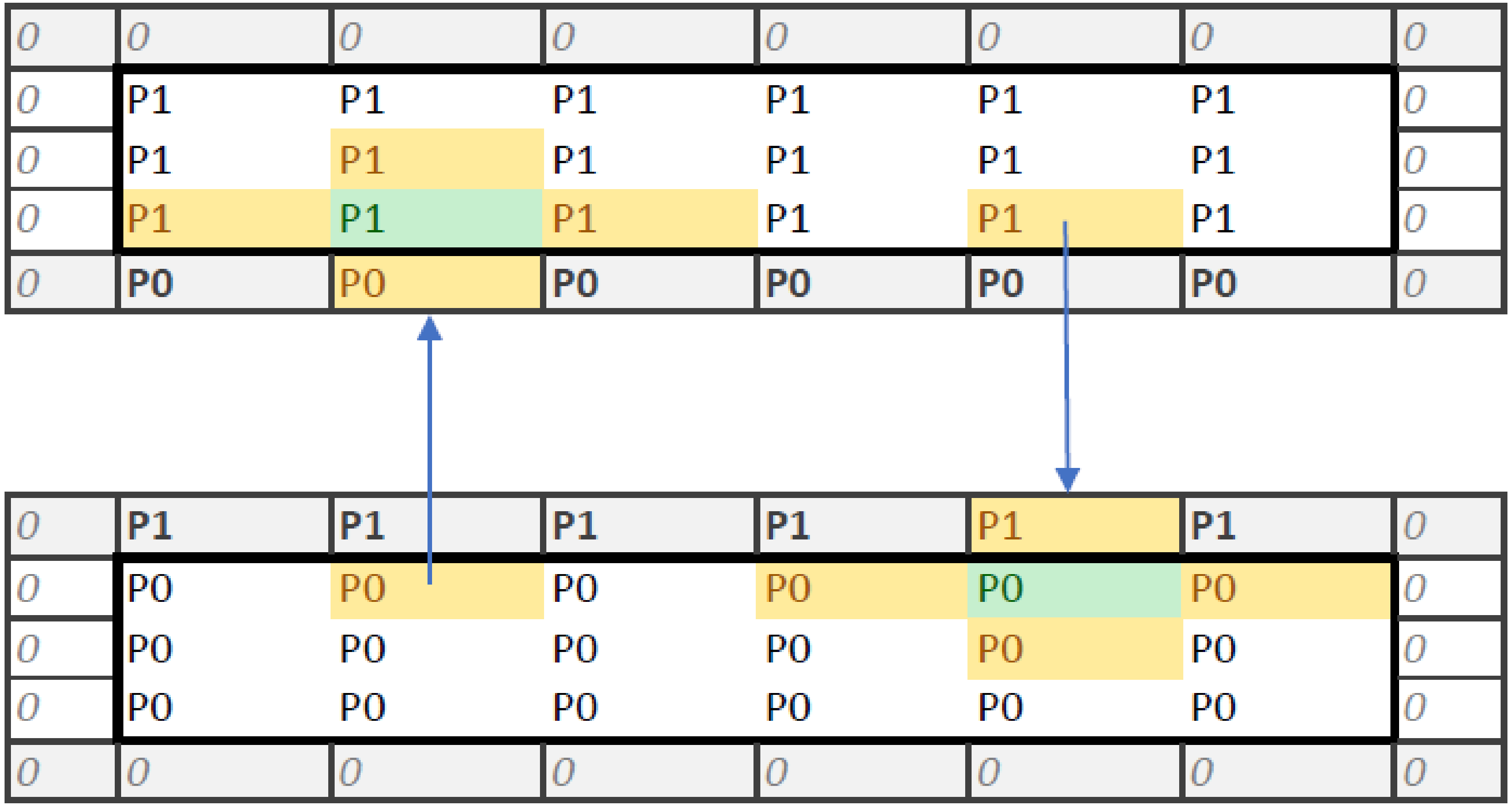
Stencil

- Implementation using array padding and ghost cells
  - Fetch left and right neighbours separately
  - Synchronize twice in each step
- Split blocks in z-direction



https://software.intel.com/content/www/us/en/develop/articles/3d-finite-differences-on-multi-core-processors.html

Stencil



0	*0*	*0*	*0*	*0*	*0*	*0*	*0*
0	P1	P1	P1	P1	P1	P1	*0*
0	P1	P1	P1	P1	P1	P1	*0*
0	P1	P1	P1	P1	P1	P1	*0*
0	**P0**	**P0**	**P0**	**P0**	**P0**	**P0**	*0*

0	**P1**	**P1**	**P1**	**P1**	**P1**	**P1**	*0*
0	P0	P0	P0	P0	P0	P0	*0*
0	P0	P0	P0	P0	P0	P0	*0*
0	P0	P0	P0	P0	P0	P0	*0*
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Stencil

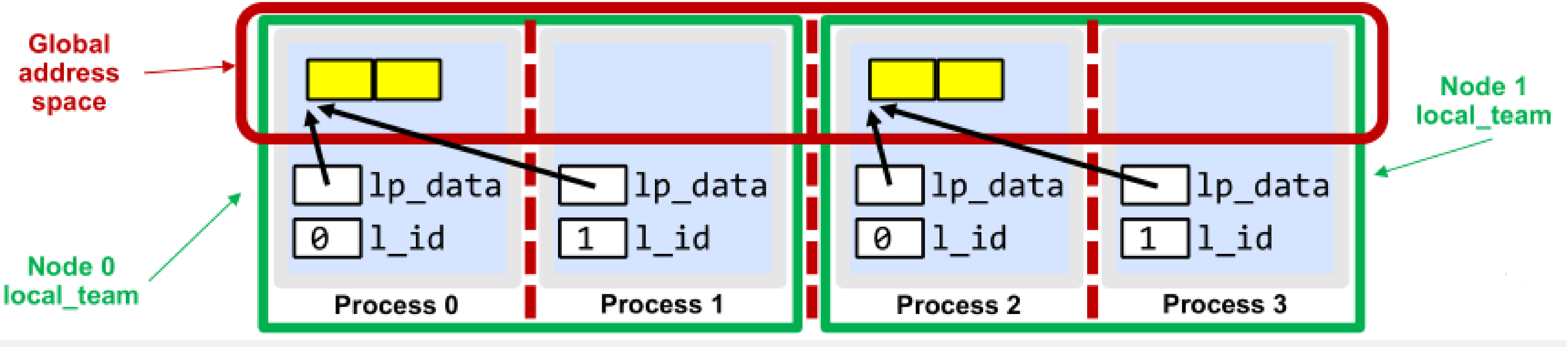
```cpp
// As rget does not allow source values to be modified until operation completion
// is notified, first retrieve all right neighbors, then all left neighbors.
if (proc_id != proc_n - 1) {
    upcxx::global_ptr<float> input_r = input_g.fetch(proc_id + 1).wait();
    upcxx::rget(input_r + n_ghost_offset,
                input + n_local - n_ghost_offset,
                n_ghost_offset).wait();
}
upcxx::barrier();

if (proc_id != 0) {
    upcxx::global_ptr<float> input_l = input_g.fetch(proc_id - 1).wait();
    upcxx::rget(input_l + n_local - 2*n_ghost_offset,
                input,
                n_ghost_offset).wait();
}
upcxx::barrier();
```

Stencil

- Possible improvements
    - Latency hiding
        - Compute inner block asynchronously
        - Use **upcxx::dist_object** for asynchronous point-to-point
    - Tiling (x- and y-dimension)
        - Hybrid implementation: **OpenMP** for x- and y-tiling, UPC++ for z-tiling
        - **upcxx::local_team** (processes corresponding to a physical node, avoid communication)

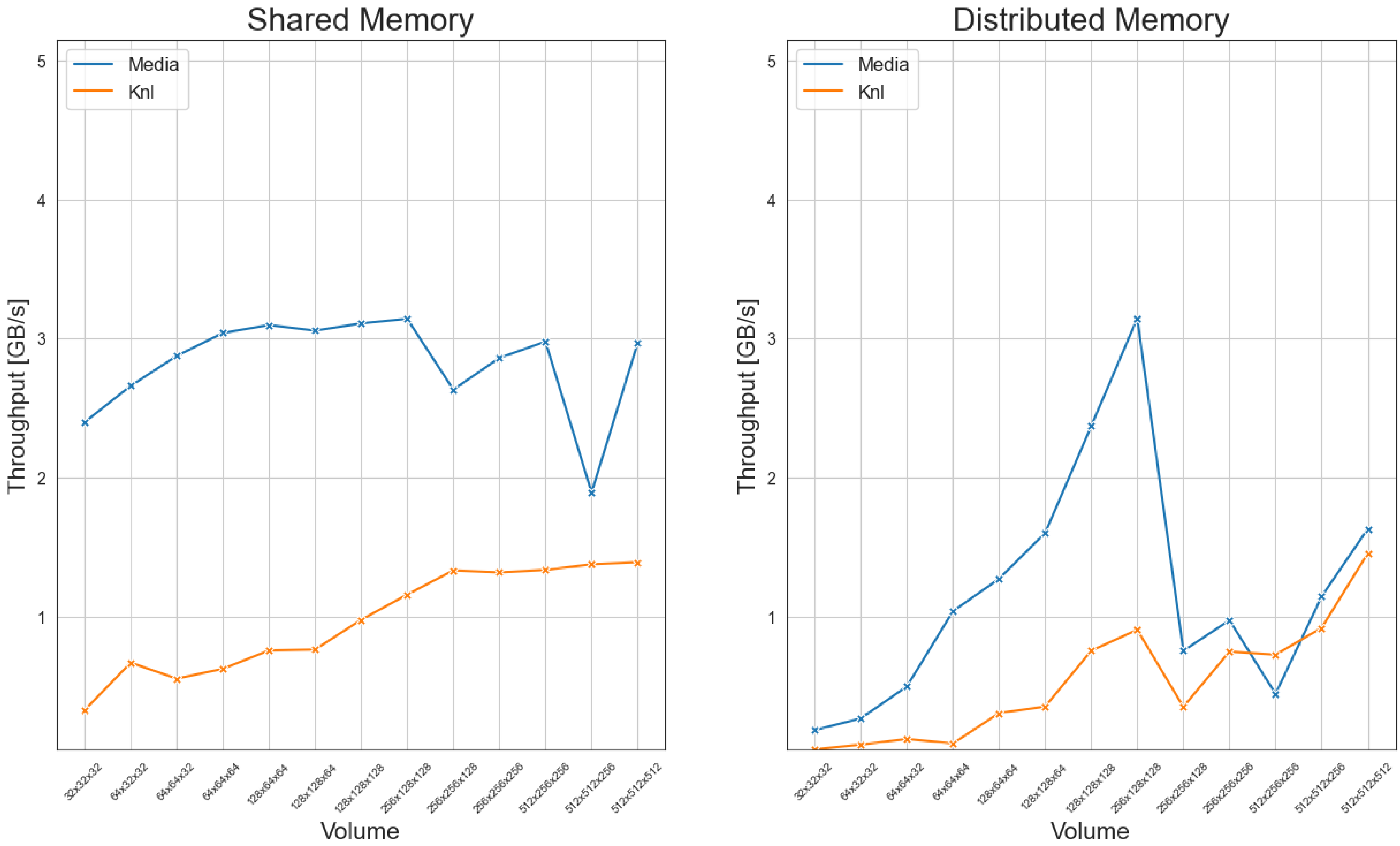
Stencil

- Possible improvements
  - **upcxx::local_team**



Stencil

- Throughput:
  - Count only elements in input and output array:
    steps * sizeof(float) * dim_x * dim_y * dim_z * 1e-9 / time
- Benchmarks
  - 4 processes per SKL node, 16 processes per KNL node
    - ⚠ Lower dimensions need a smaller amount of processes for correct results.
  - Parameters: 5 steps, radius 2
  - Averaged over 100 iterations


Shared Memory
Media
Knl
Throughput [GB/s]
1
2
3
4
5
32x32x32
64x32x32
64x64x32
64x64x64
128x64x64
128x128x64
128x128x128
256x128x128
256x256x128
256x256x256
512x256x256
512x512x256
512x512x512
Volume
Distributed Memory
Media
Knl
Throughput [GB/s]
1
2
3
4
5
32x32x32
64x32x32
64x64x32
64x64x64
128x64x64
128x128x64
128x128x128
256x128x128
256x256x128
256x256x256
512x256x256
512x512x256
512x512x512
Volume

Thank you for your attention!

Questions?