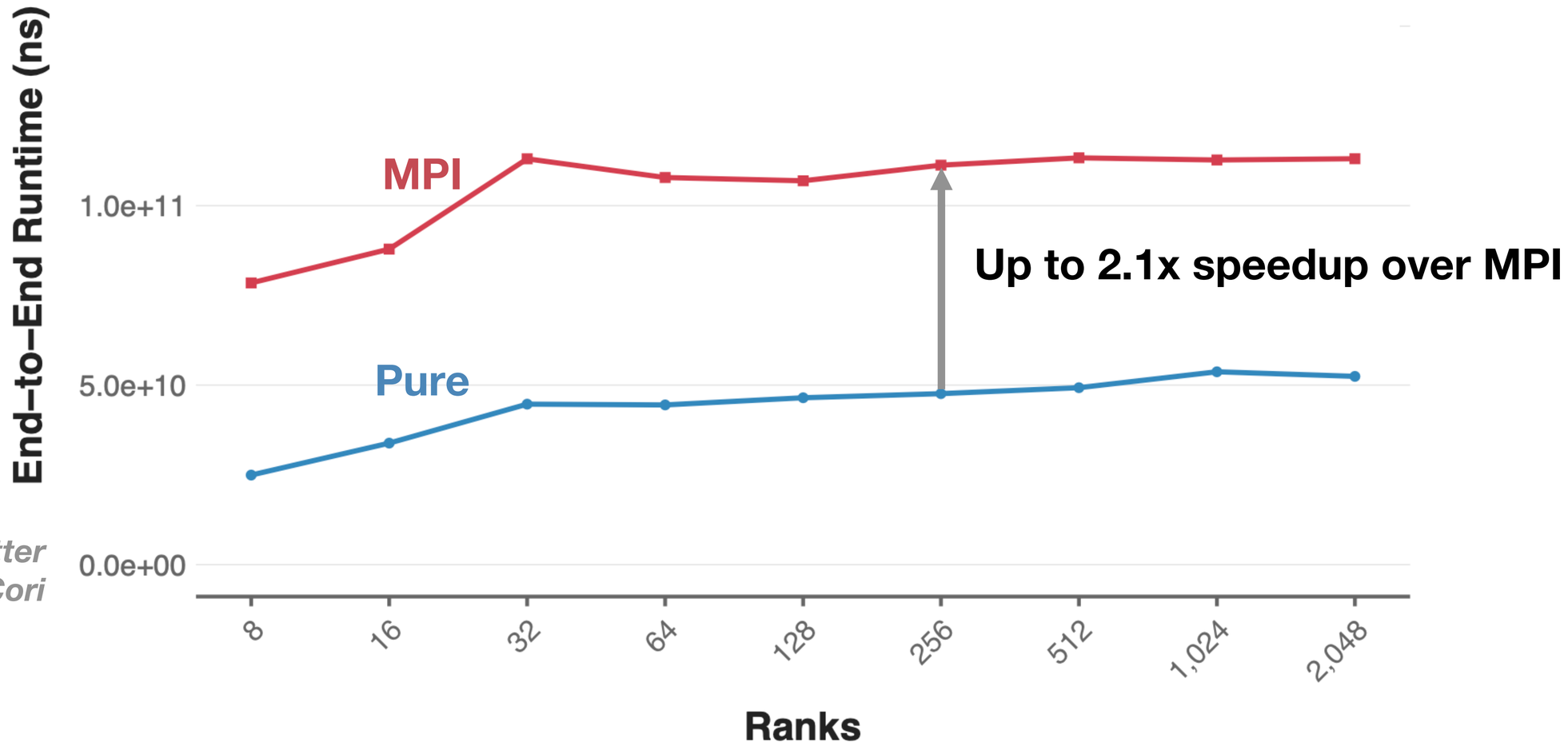
Pure: Evolving Message Passing To Better Leverage Shared Memory Within Nodes

James Psota
Armando Solar-Lezama

Goal of the Pure Programming System

Outperform MPI with less programmer effort (than MPI+X or alternatives)

CoMD with Imbalance: 1.5x—2.1x Speedup



lower is better
On NERSC's Cori

Pure System Overview

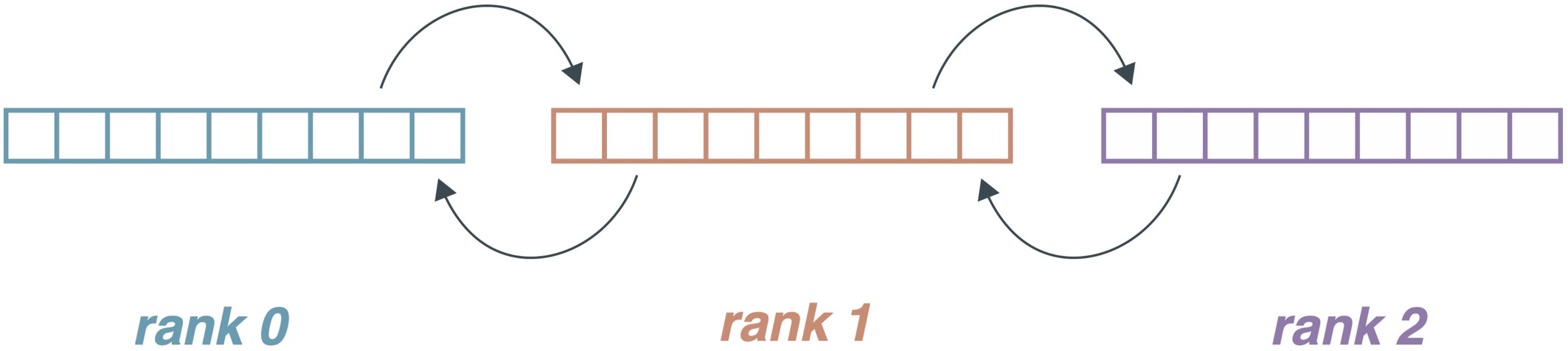
Pure is a **programming model** and **runtime system** that improves **performance over MPI** on **clusters of multicores** with **minimal** additional programming **effort**

Example Code

(MPI and Pure)

Example Code Diagram

- 1-D stencil with **random work** applied to each element (load imbalance)
- After work done, halo exchange



Example Code Diagram

- 1-D stencil with **random work** applied to each element (load imbalance)
- After work done, halo exchange

rank 0

rank 1

Done work. Blocking on receive.

rank 2

Done work. Blocking on receive.

```
1  void rand_stencil_mpi(double* const a, size_t SIZE, size_t iters, int my_rank,
2                        int n_ranks) {
3      double temp[SIZE];
4
5
6
7
8
9
10
11     for (all iters) {
12         for (all elements) {
13             temp[i] = random_work(a[i]);
14         }
15         // average with adjacent elements
16         for (auto i = 1; i < SIZE - 1; ++i) {
17             a[i] = (temp[i - 1] + temp[i] + temp[i + 1]) / 3.0;
18         }
19         if (my_rank > 0) { // exchange elements with low neighbor
20             MPI_Send(...);
21             MPI_Recv(...);
22             a[0] = (neighbor_hi_val + temp[0] + temp[1]) / 3.0;
23         }                              // ends if not first rank
24         if (my_rank < n_ranks - 1) { // exchange elements with low neighbor
25             MPI_Send(...);
26             MPI_Recv(...);
27             a[SIZE - 1] =
28                     (temp[SIZE - 2] + temp[SIZE - 1] + neighbor_lo_val) / 3.0;
29         } // ends if not last rank
30     }     // ends for all iterations
31 }
```

1. call random_work on all array elements

2. local updates and halo exchange

MPI Code

```mpi
void rand_stencil_mpi(double* const a, size_t SIZE, size_t iters, int my_rank,
                      int n_ranks) {
    double temp[SIZE];

    for (all iters) {
        for (all elements) {
            // average with adjacent elements
            for (auto i = 1; i < SIZE - 1; ++i) {
                a[i] = (temp[i - 1] + temp[i] + temp[i + 1]) / 3.0;
            }
            if (my_rank > 0) { // exchange elements with low neighbor
                MPI_Send(...);
                MPI_Recv(...);
                a[0] = (neighbor_hi_val + temp[0] + temp[1]) / 3.0;
            }                              // ends if not first rank
            if (my_rank < n_ranks - 1) { // exchange elements with low neighbor
                MPI_Send(...);
                MPI_Recv(...);
                a[SIZE - 1] =
                        (temp[SIZE - 2] + temp[SIZE - 1] + neighbor_lo_val) / 3.0;
            } // ends if not last rank
        }   // ends for all iterations
}
```

MPI Code

```pure
void rand_stencil_pure(double* const a, size_t SIZE, size_t iters, int my_rank,
                       int n_ranks) {
    double   temp[SIZE];

    for (all iters) {
        // execute all chunks of rand_work_task
        // average with adjacent elements
        for (auto i = 1; i < SIZE - 1; ++i) {
            a[i] = (temp[i - 1] + temp[i] + temp[i + 1]) / 3.0;
        }
        if (my_rank > 0) { // exchange elements with low neighbor
            pure_send_msg(...);
            pure_recv_msg(...);
            a[0] = (neighbor_hi_val + temp[0] + temp[1]) / 3.0;
        }                              // ends if not first rank
        if (my_rank < n_ranks - 1) { // exchange elements with low neighbor
            pure_send_msg(...);
            pure_recv_msg(...);
            a[SIZE - 1] =
                    (temp[SIZE - 2] + temp[SIZE - 1] + neighbor_lo_val) / 3.0;
        } // ends if not last rank
    }   // ends for all iterations
}
```

Pure Code

Almost same code — same arguments, different function names

```mpi
void rand_stencil_mpi(double* const a, size_t SIZE, size_t iters, int my_rank,
                      int n_ranks) {
    double temp[SIZE];

    for (all iters) {
        for (all elements) {
            temp[i] = random_work(a[i]);
        }
        // average with adjacent elements
        for (auto i = 1; i < SIZE - 1; ++i) {
            a[i] = (temp[i - 1] + temp[i] + temp[i + 1]) / 3.0;
        }
        if (my_rank > 0) { // exchange elements with low neighbor
            MPI_Send(...);
            MPI_Recv(...);
            a[0] = (neighbor_hi_val + temp[0] + temp[1]) / 3.0;
        }                              // ends if not first rank
        if (my_rank < n_ranks - 1) { // exchange elements with low neighbor
            MPI_Send(...);
            MPI_Recv(...);
            a[SIZE - 1] =
                    (temp[SIZE - 2] + temp[SIZE - 1] + neighbor_lo_val) / 3.0;
        } // ends if not last rank
    }     // ends for all iterations
}
```

MPI Code

Define Pure Task Once
C++ Lambda (thread-safe)

```pure
void rand_stencil_pure(double* const a, size_t SIZE, size_t iters, int my_rank,
                       int n_ranks) {
    double   temp[SIZE];
    PureTask rand_work_task = [a, temp, my_rank](unsigned chunk) {
        auto [min_idx, max_idx] = pure_aligned_idx_range<double>(SIZE, chunk);
        for (auto i = min_idx; i <= max_idx; ++i) {
            temp[i] = random_work(a[i]);
        }
    }; // ends defining the Pure Task rand_work_task

    for (all iters) {
        // execute all chunks of rand_work_task
        rand_work_task.execute();

        // average with adjacent elements
        for (auto i = 1; i < SIZE - 1; ++i) {
            a[i] = (temp[i - 1] + temp[i] + temp[i + 1]) / 3.0;
        }
        if (my_rank > 0) { // exchange elements with low neighbor
            pure_send_msg(...);
            pure_recv_msg(...);
            a[0] = (neighbor_hi_val + temp[0] + temp[1]) / 3.0;
        }                              // ends if not first rank
        if (my_rank < n_ranks - 1) { // exchange elements with low neighbor
            pure_send_msg(...);
            pure_recv_msg(...);
            a[SIZE - 1] =
                    (temp[SIZE - 2] + temp[SIZE - 1] + neighbor_lo_val) / 3.0;
        } // ends if not last rank
    }     // ends for all iterations
}
```

Pure Code

CSAIL

```mpi
void rand_stencil_mpi(double* const a, size_t SIZE, size_t iters, int my_rank,
                      int n_ranks) {
    double temp[SIZE];

    for (all iters) {
        for (all elements) {
            temp[i] = random_work(a[i]);
        }
        // average with adjacent elements
        for (auto i = 1; i < SIZE - 1; ++i) {
            a[i] = (temp[i - 1] + temp[i] + temp[i + 1]) / 3.0;
        }
        if (my_rank > 0) { // exchange elements with low neighbor
            MPI_Send(...);
            MPI_Recv(...);
            a[0] = (neighbor_hi_val + temp[0] + temp[1]) / 3.0;
        }                              // ends if not first rank
        if (my_rank < n_ranks - 1) { // exchange elements with low neighbor
            MPI_Send(...);
            MPI_Recv(...);
            a[SIZE - 1] =
                    (temp[SIZE - 2] + temp[SIZE - 1] + neighbor_lo_val) / 3.0;
        } // ends if not last rank
    }     // ends for all iterations
}
```

Application calls function directly

MPI Code

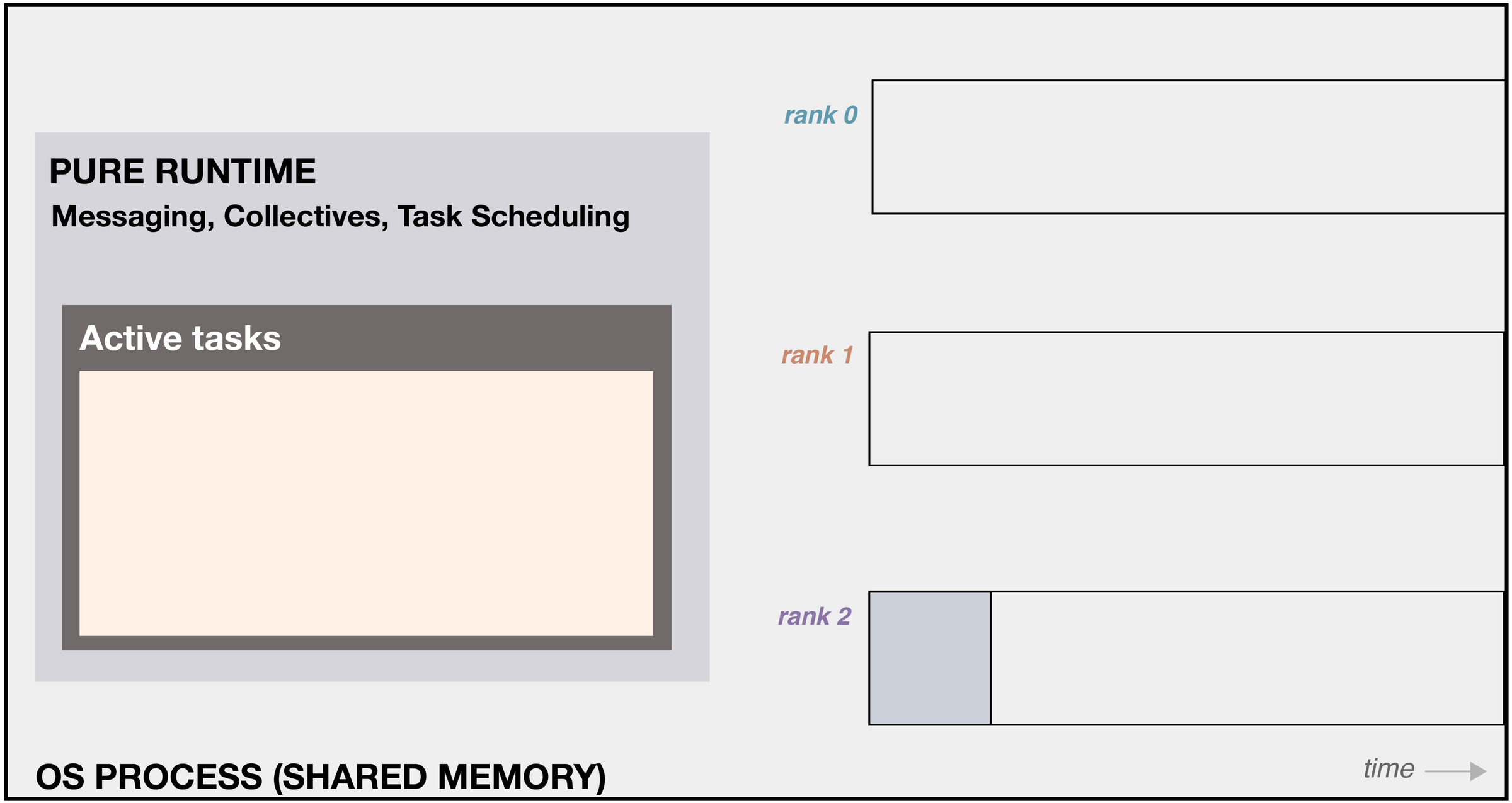
```pure
void rand_stencil_pure(double* const a, size_t SIZE, size_t iters, int my_rank,
                       int n_ranks) {
    double   temp[SIZE];
    PureTask rand_work_task = [a, temp, my_rank](unsigned chunk) {
        auto [min_idx, max_idx] = pure_aligned_idx_range<double>(SIZE, chunk);
        for (auto i = min_idx; i <= max_idx; ++i) {
            temp[i] = random_work(a[i]);
        }
    };

    for (all iters) {
        // execute all chunks of rand_work_task
        rand_work_task.execute();

        // average with adjacent elements
        for (auto i = 1; i < SIZE - 1; ++i) {
            a[i] = (temp[i - 1] + temp[i] + temp[i + 1]) / 3.0;
        }
        if (my_rank > 0) { // exchange elements with low neighbor
            pure_send_msg(...);
            pure_recv_msg(...);
            a[0] = (neighbor_hi_val + temp[0] + temp[1]) / 3.0;
        }                              // ends if not first rank
        if (my_rank < n_ranks - 1) { // exchange elements with low neighbor
            pure_send_msg(...);
            pure_recv_msg(...);
            a[SIZE - 1] =
                    (temp[SIZE - 2] + temp[SIZE - 1] + neighbor_lo_val) / 3.0;
        } // ends if not last rank
    }     // ends for all iterations
}
```

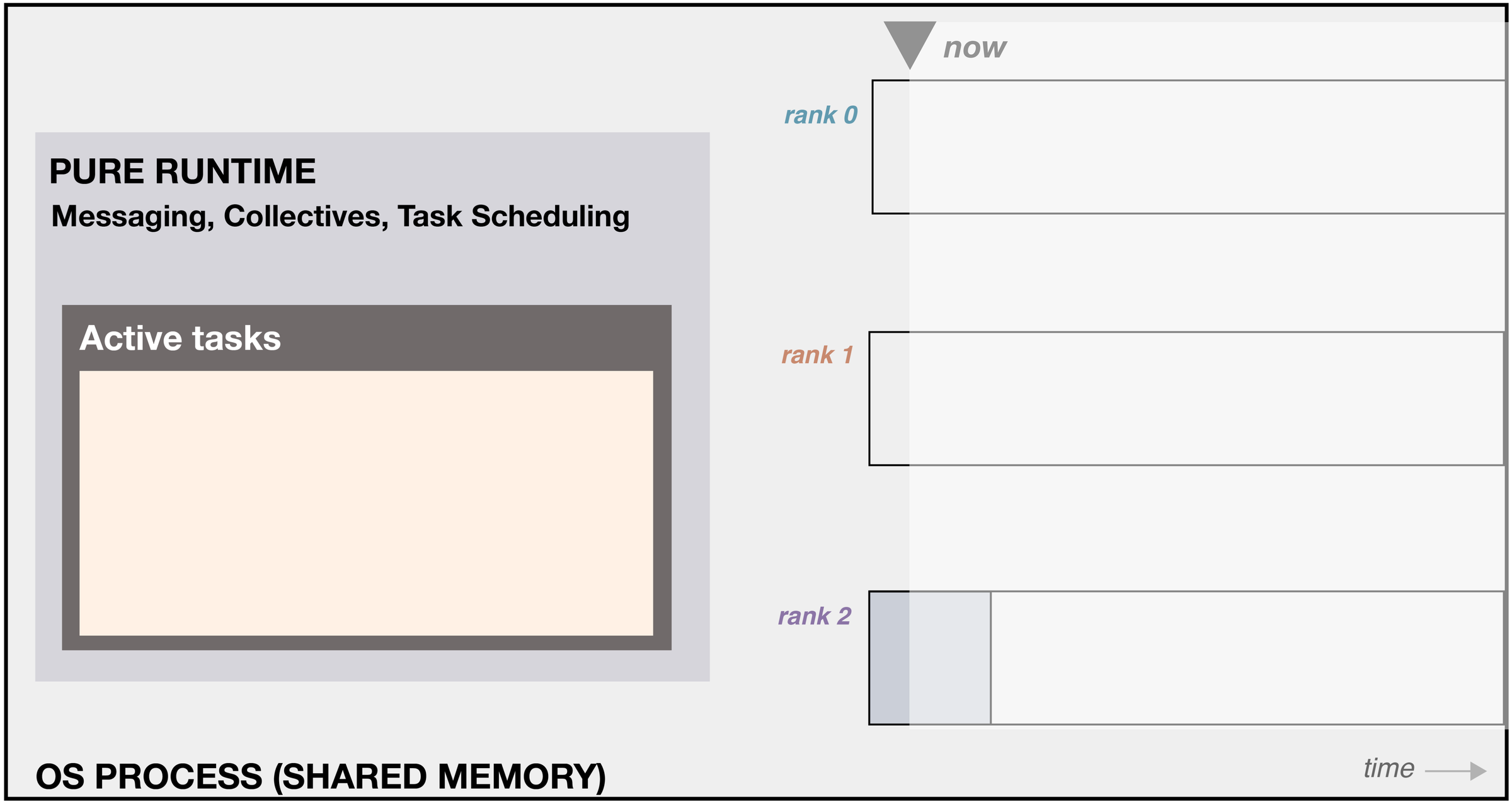
Pure Runtime executes task

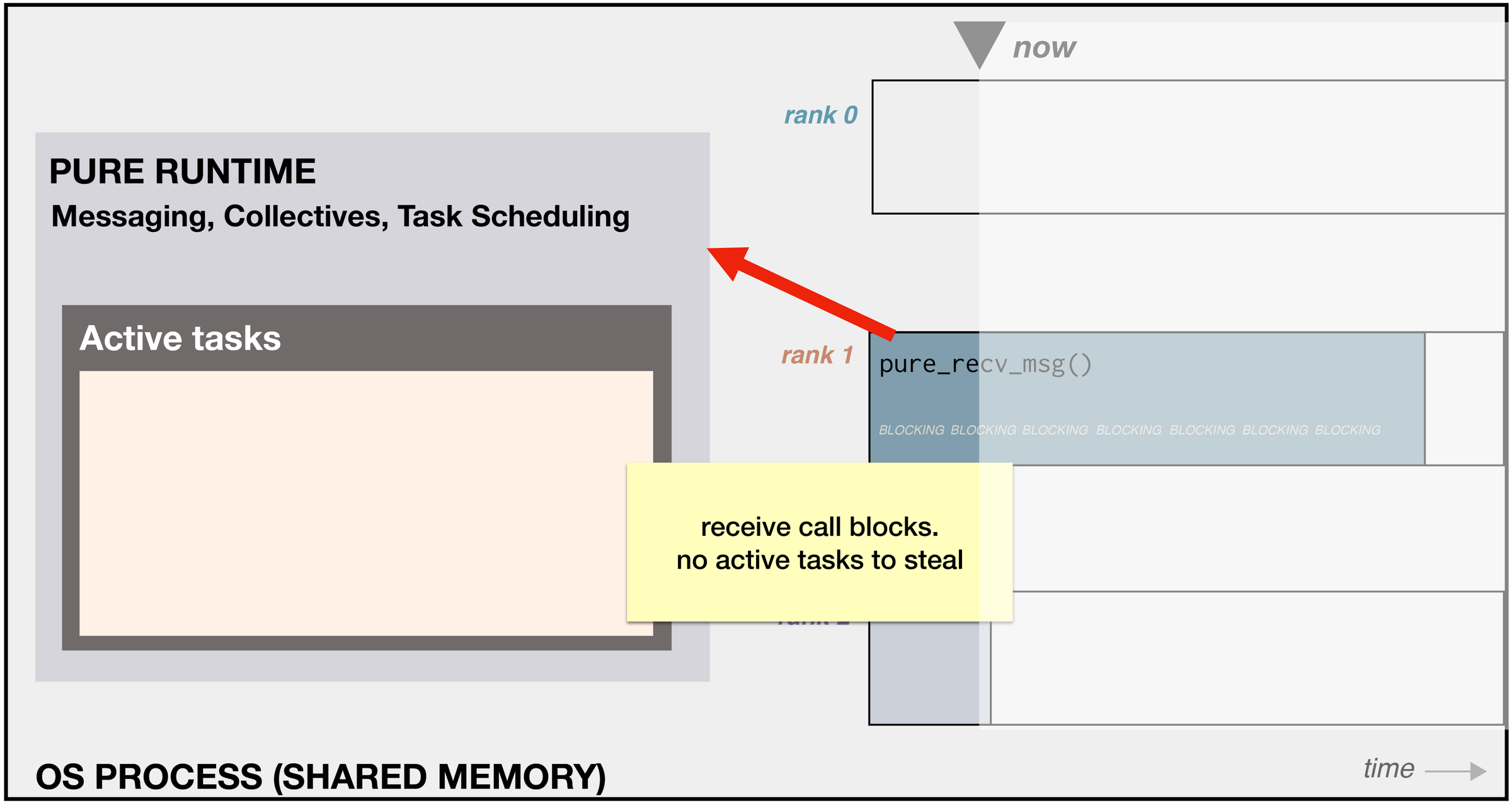
Pure Code

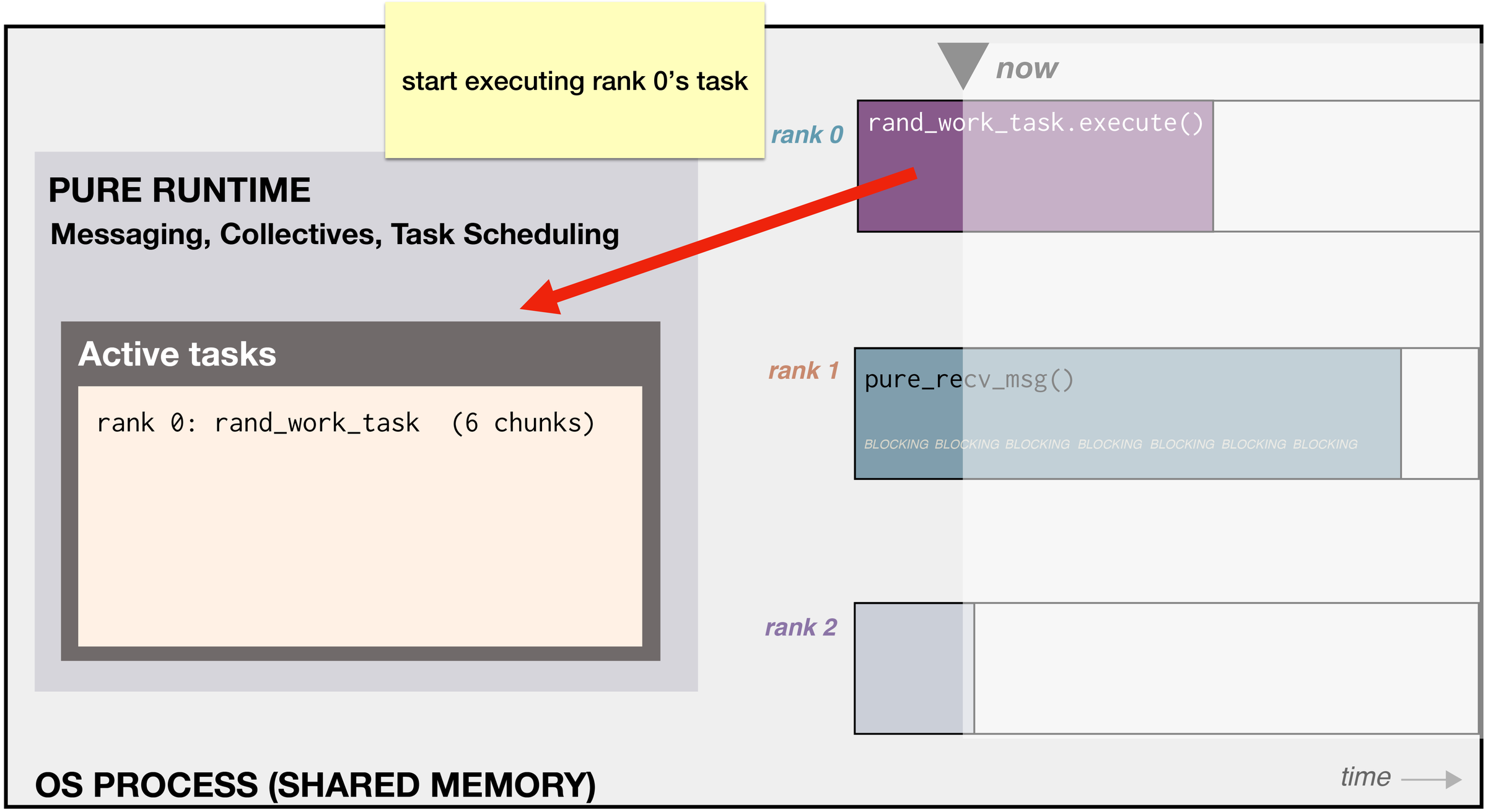
CSAIL

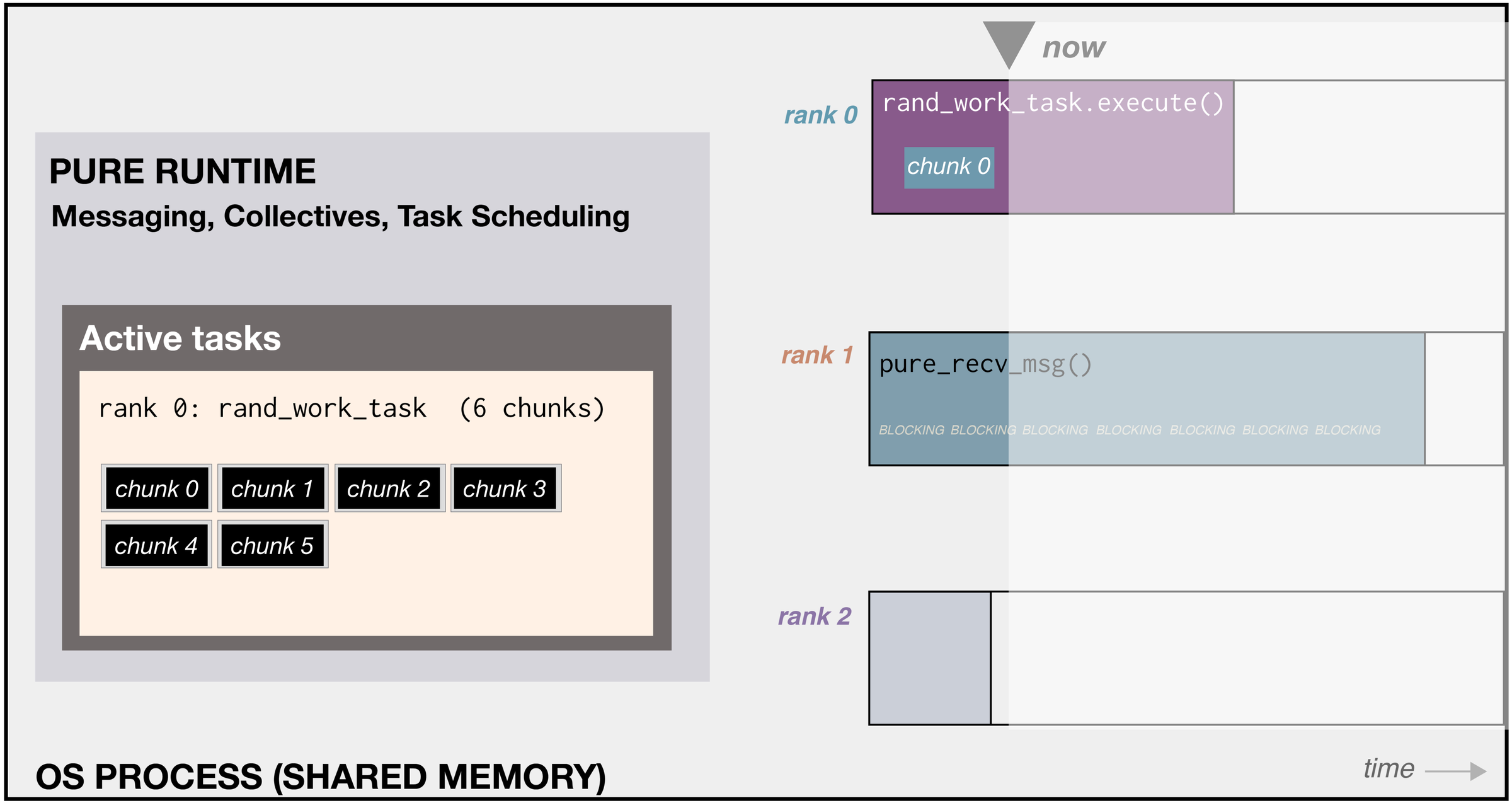
Example Walk-Through

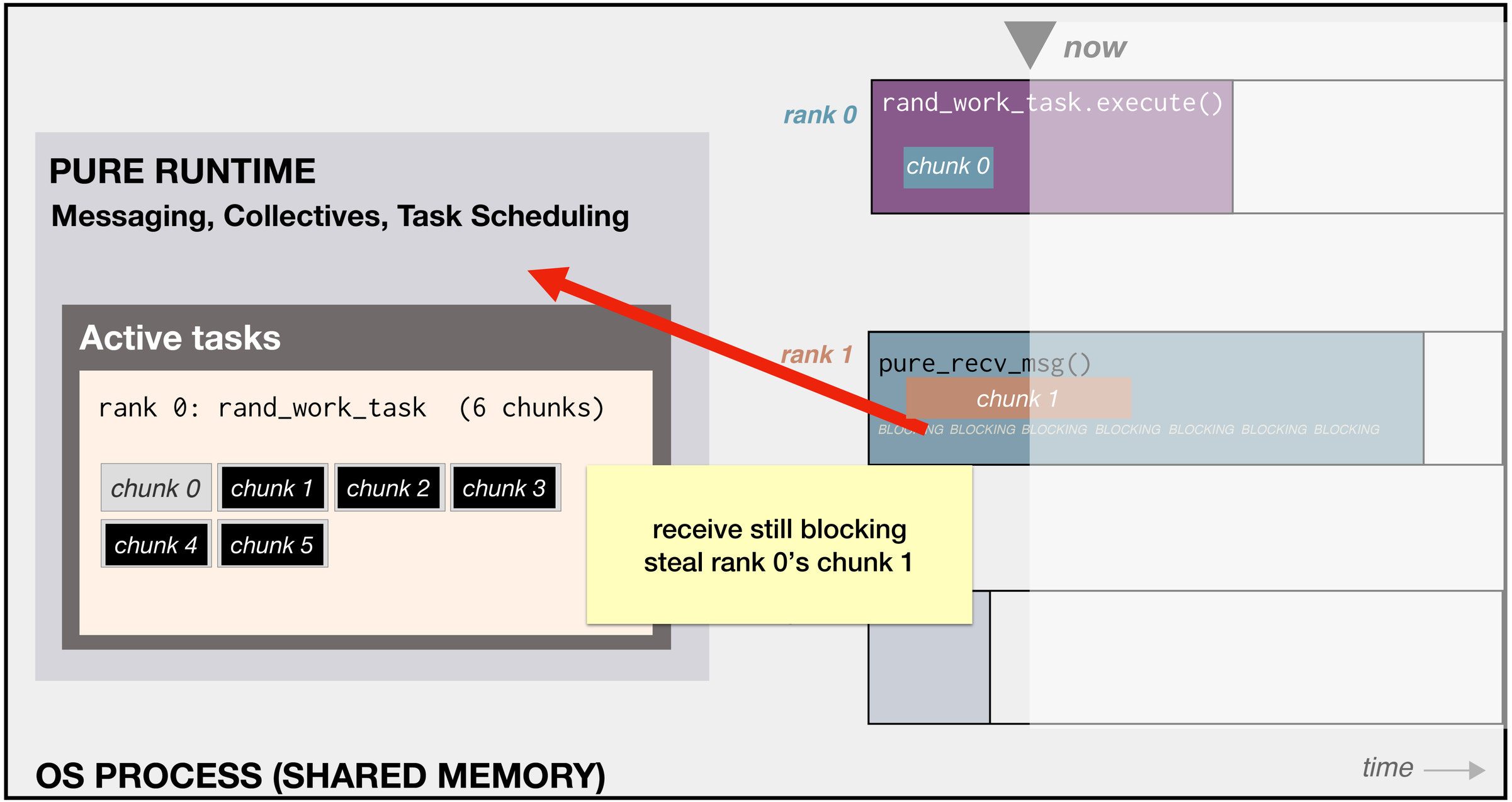

PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
rank 0
rank 1
rank 2
time
OS PROCESS (SHARED MEMORY)

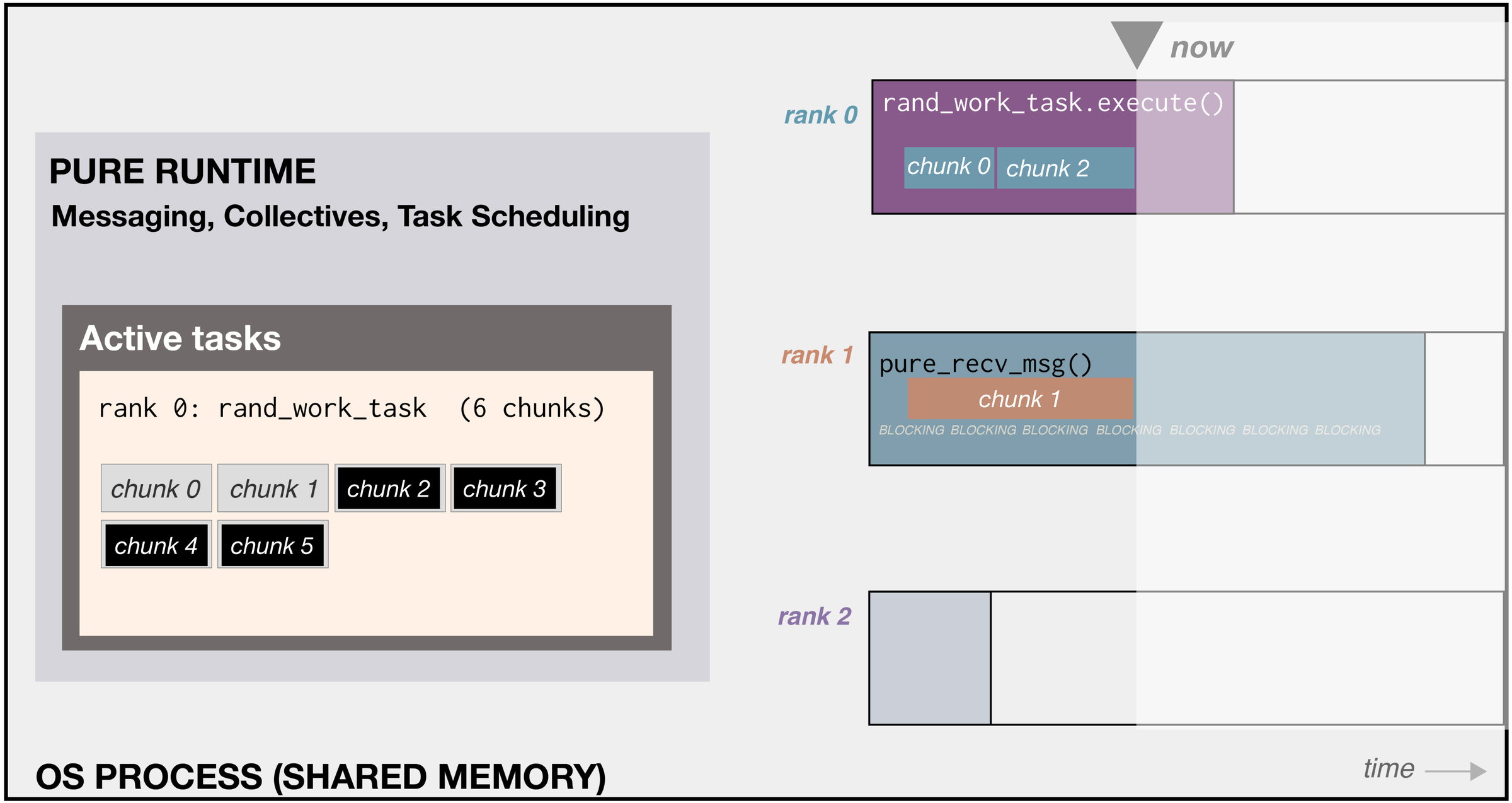

now
rank 0
rank 1
rank 2
PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
OS PROCESS (SHARED MEMORY)
time

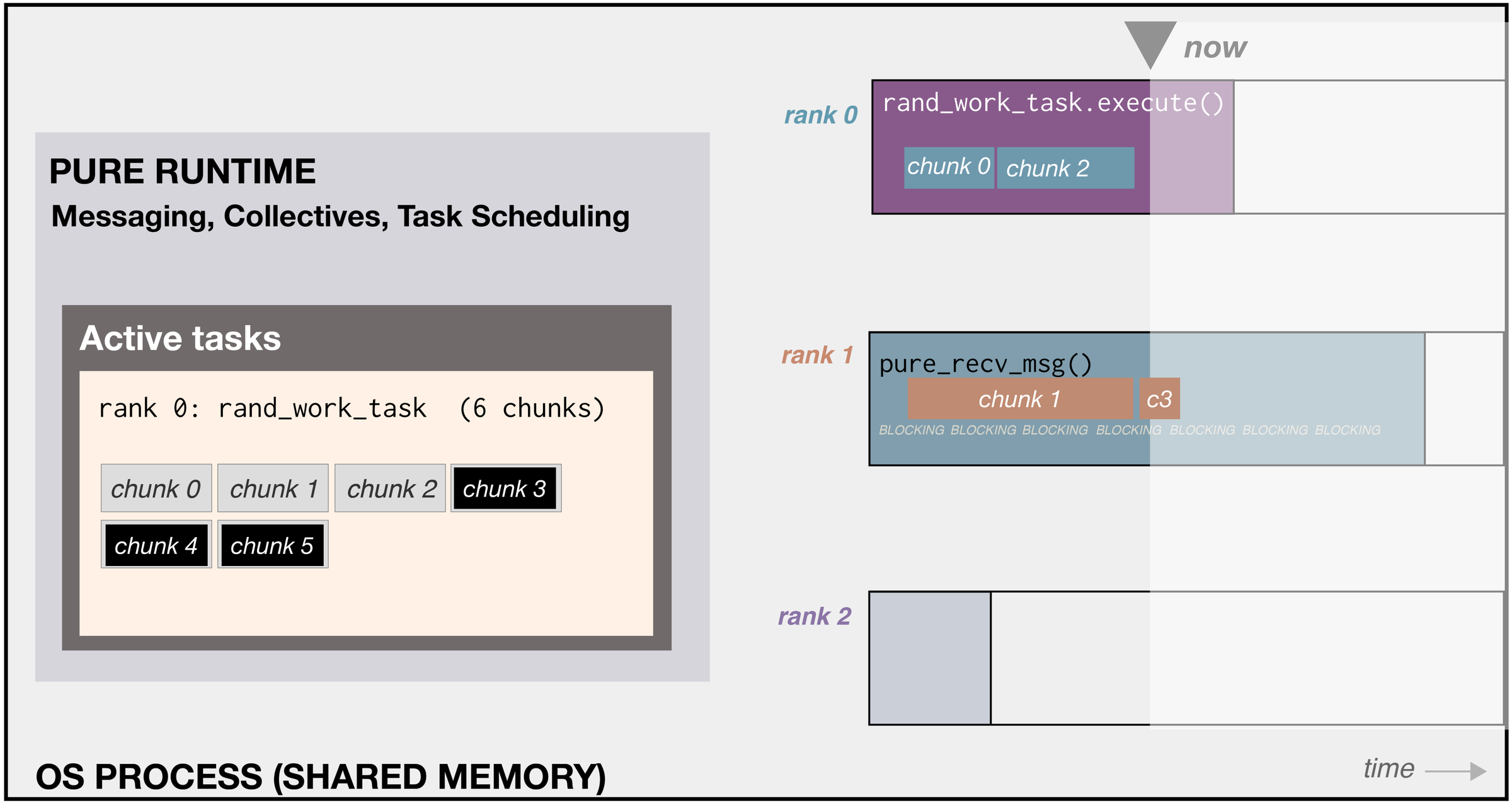

now
rank 0
PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
rank 1
pure_recv_msg()
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
receive call blocks.
no active tasks to steal
OS PROCESS (SHARED MEMORY)
time

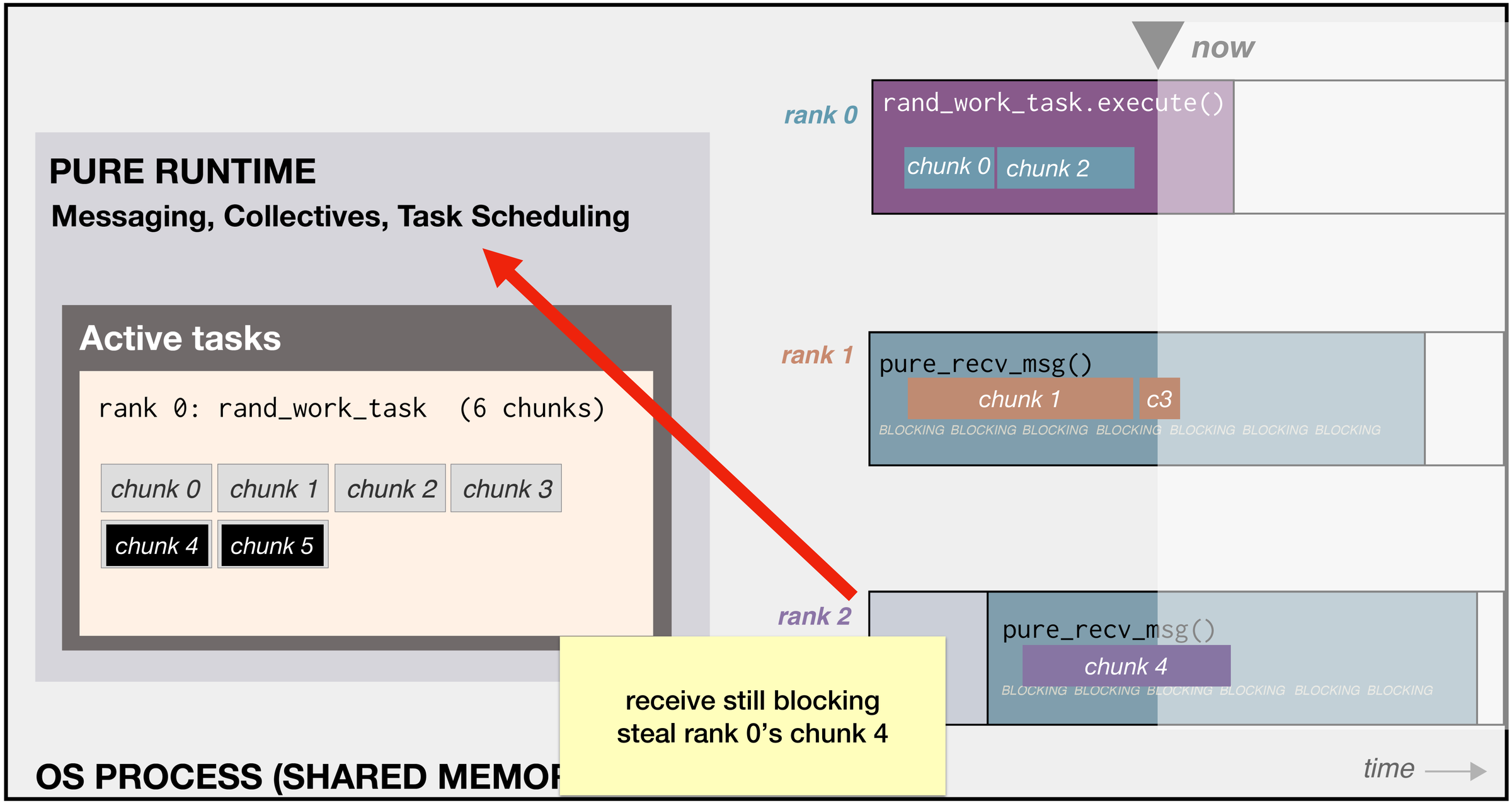

start executing rank 0's task
PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
rank 0: rand_work_task (6 chunks)
now
rank 0
rand_work_task.execute()
rank 1
pure_recv_msg()
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
rank 2
time
OS PROCESS (SHARED MEMORY)

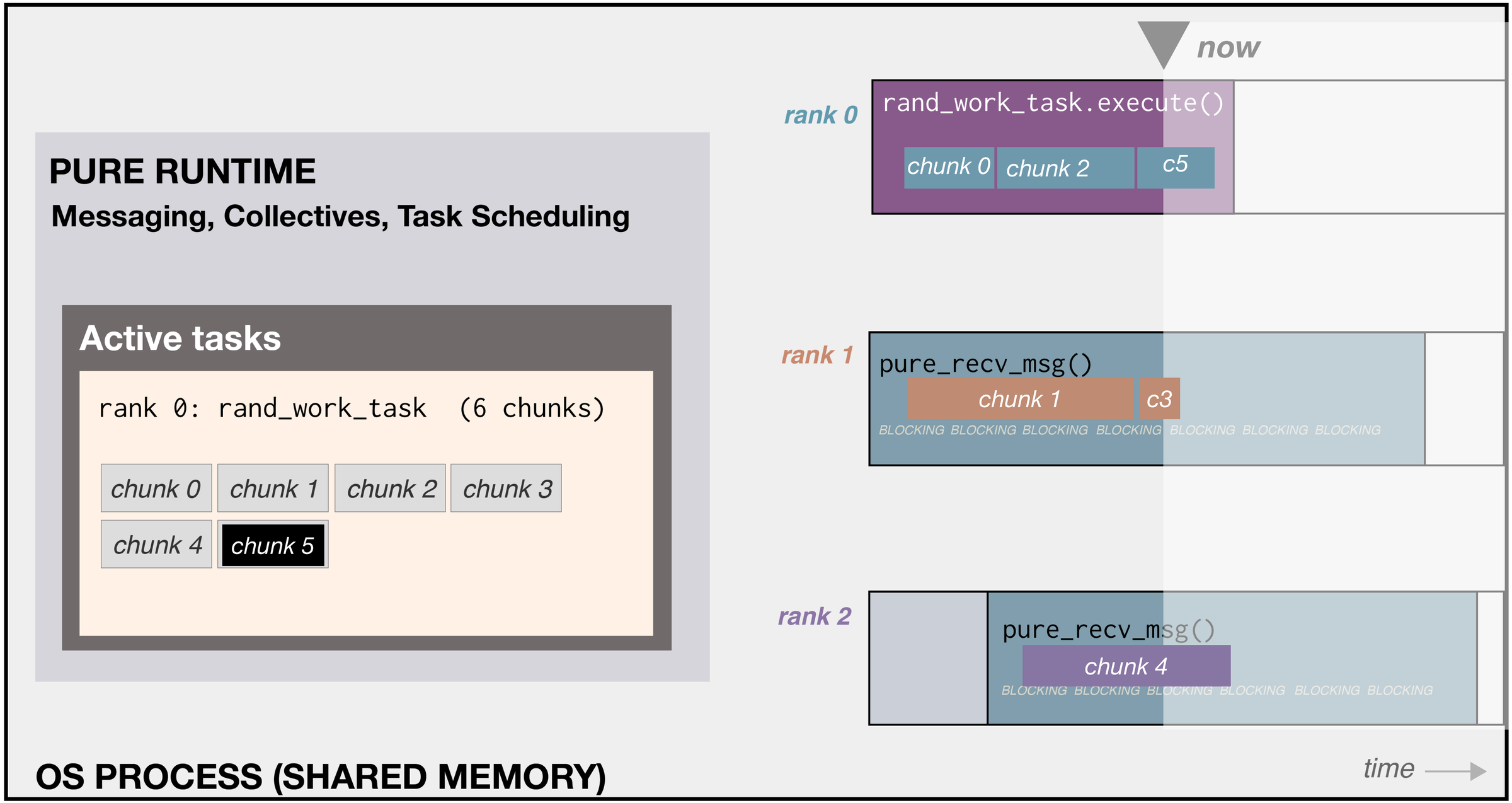

PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
rank 0: rand_work_task (6 chunks)
chunk 0
chunk 1
chunk 2
chunk 3
chunk 4
chunk 5
now
rank 0
rand_work_task.execute()
chunk 0
rank 1
pure_recv_msg()
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
rank 2
time
OS PROCESS (SHARED MEMORY)

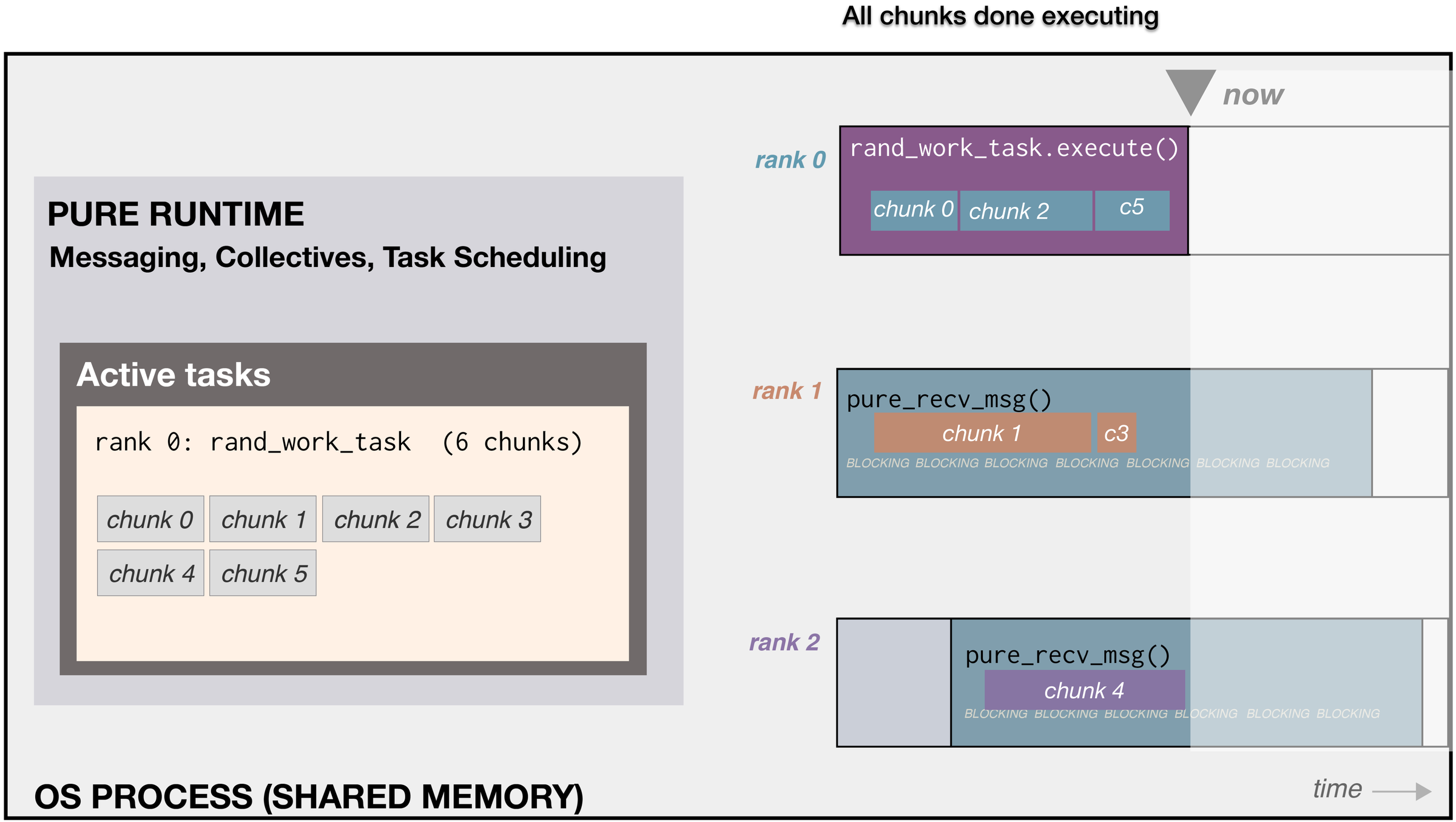

now
rank 0
rand_work_task.execute()
chunk 0
PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
rank 0: rand_work_task (6 chunks)
chunk 0
chunk 1
chunk 2
chunk 3
chunk 4
chunk 5
rank 1
pure_recv_msg()
chunk 1
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
receive still blocking
steal rank 0's chunk 1
OS PROCESS (SHARED MEMORY)
time

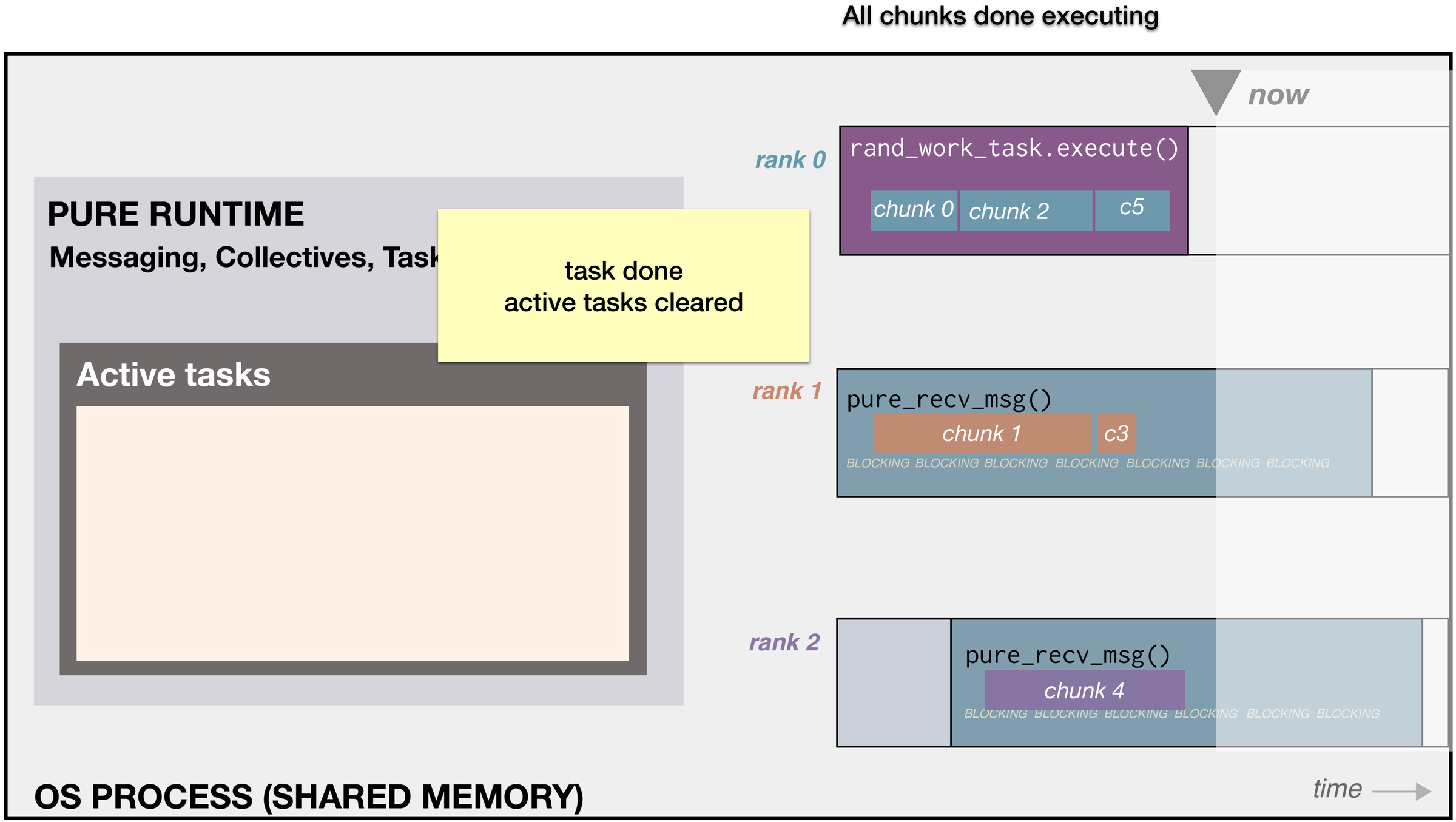

PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
rank 0: rand_work_task (6 chunks)
chunk 0
chunk 1
chunk 2
chunk 3
chunk 4
chunk 5
now
rank 0
rand_work_task.execute()
chunk 0
chunk 2
rank 1
pure_recv_msg()
chunk 1
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
rank 2
OS PROCESS (SHARED MEMORY)
time


PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
rank 0: rand_work_task (6 chunks)
chunk 0
chunk 1
chunk 2
chunk 3
chunk 4
chunk 5
now
rank 0
rand_work_task.execute()
chunk 0
chunk 2
rank 1
pure_recv_msg()
chunk 1
c3
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
rank 2
OS PROCESS (SHARED MEMORY)
time


now
rank 0
rand_work_task.execute()
chunk 0
chunk 2
rank 1
pure_recv_msg()
chunk 1
c3
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
rank 2
pure_recv_msg()
chunk 4
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
time
PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
rank 0: rand_work_task (6 chunks)
chunk 0
chunk 1
chunk 2
chunk 3
chunk 4
chunk 5
receive still blocking
steal rank 0's chunk 4
OS PROCESS (SHARED MEMOR

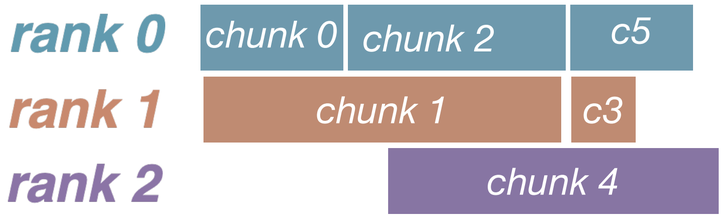

PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
rank 0: rand_work_task (6 chunks)
chunk 0
chunk 1
chunk 2
chunk 3
chunk 4
chunk 5
now
rank 0
rand_work_task.execute()
chunk 0
chunk 2
c5
rank 1
pure_recv_msg()
chunk 1
c3
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
rank 2
pure_recv_msg()
chunk 4
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
time
OS PROCESS (SHARED MEMORY)


All chunks done executing
now
rank 0
rand_work_task.execute()
chunk 0
chunk 2
c5
rank 1
pure_recv_msg()
chunk 1
c3
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
rank 2
pure_recv_msg()
chunk 4
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
time
PURE RUNTIME
Messaging, Collectives, Task Scheduling
Active tasks
rank 0: rand_work_task (6 chunks)
chunk 0
chunk 1
chunk 2
chunk 3
chunk 4
chunk 5
OS PROCESS (SHARED MEMORY)


All chunks done executing
now
rank 0
rand_work_task.execute()
chunk 0
chunk 2
c5
PURE RUNTIME
Messaging, Collectives, Task
task done
active tasks cleared
Active tasks
rank 1
pure_recv_msg()
chunk 1
c3
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
rank 2
pure_recv_msg()
chunk 4
BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING BLOCKING
OS PROCESS (SHARED MEMORY)
time

Without Task Stealing

With Task Stealing



time ⟶

Example Recap

- Unified Pure runtime responsible both communication and task scheduling
- Owning rank (and other ranks that are idle) carry out task execution
- Runtime automatically tries to steal while blocking on communication
- Stealing happens only between ranks sharing memory
- Number of chunks determined by a programmer setting
- Runtime’s task scheduler responsible for concurrently allocating chunks / ensuring task completion

Related Work

1. **MPI**
2. **Hybrid / Hierarchical models**
3. **Higher-level parallel frameworks**

How does Pure work?

Key Ideas Driving Pure's Performance

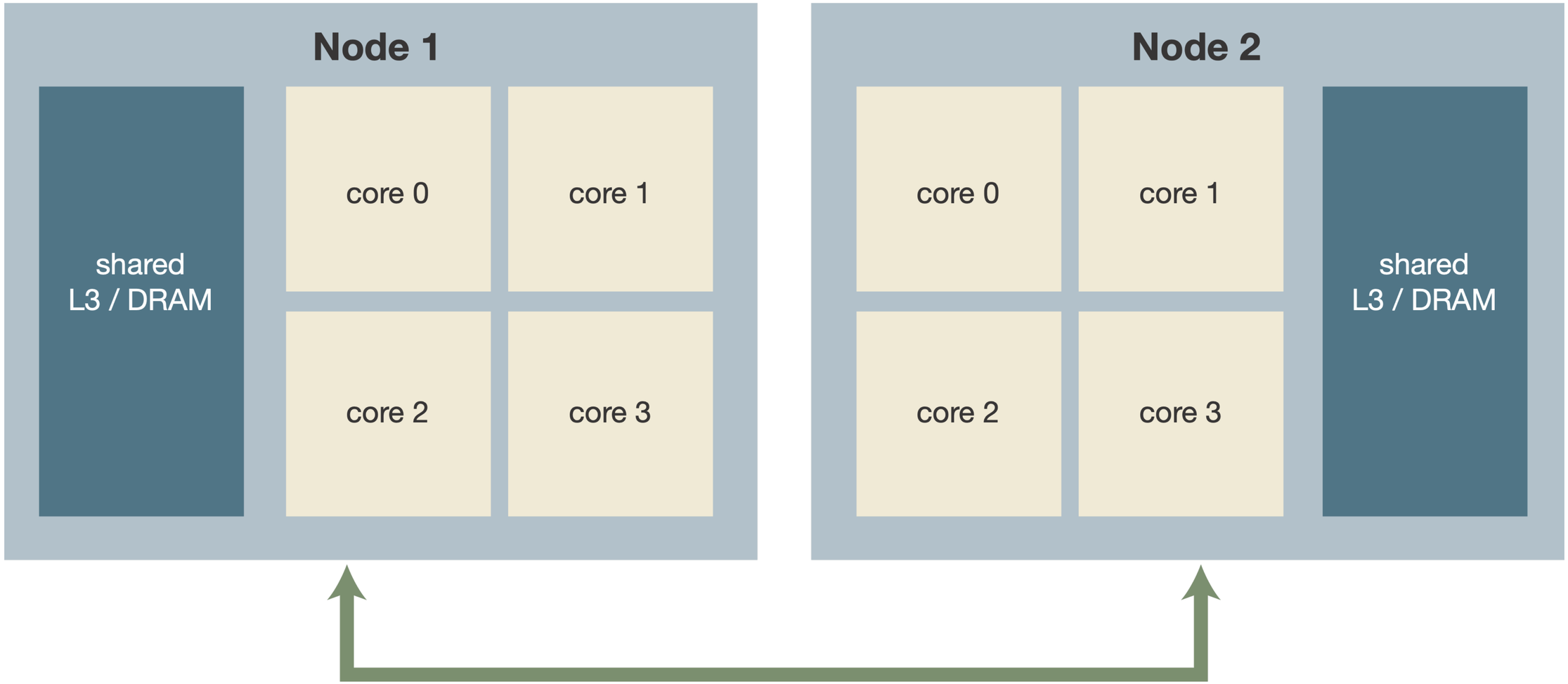
Design Driver #1: Leverage shared memory via threads within nodes

- Pure's use of threads and shared memory enables highly efficient fine-grain synchronization
- Enables fine-grain, low overhead work stealing between threads

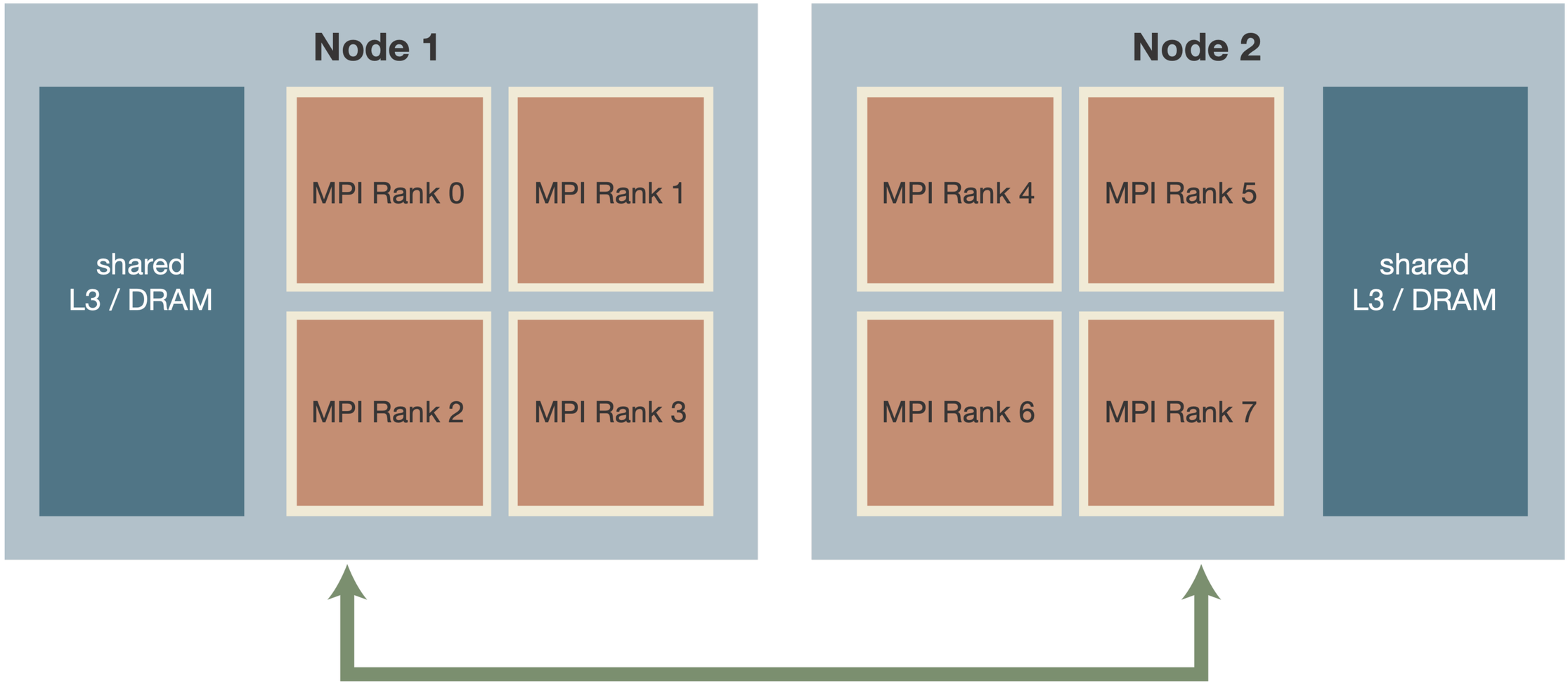
Design Driver #2: Expand responsibility of runtime system to handle communication and task scheduling

- Runtime optionally executes programmer-defined Tasks; enables automatic work stealing when communication is blocked
- Enables dynamic load balancing (within nodes)

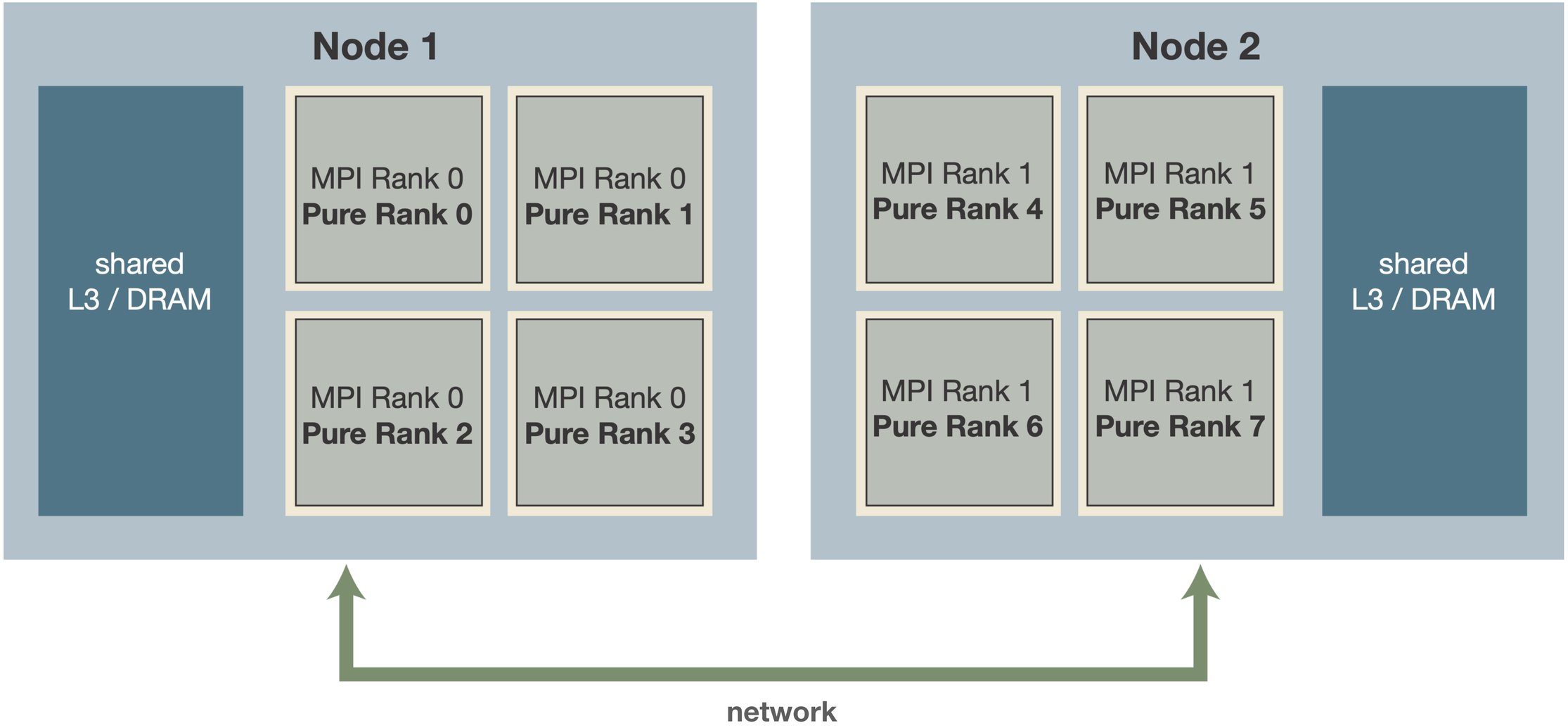
Rank to Core Mapping: Hardware



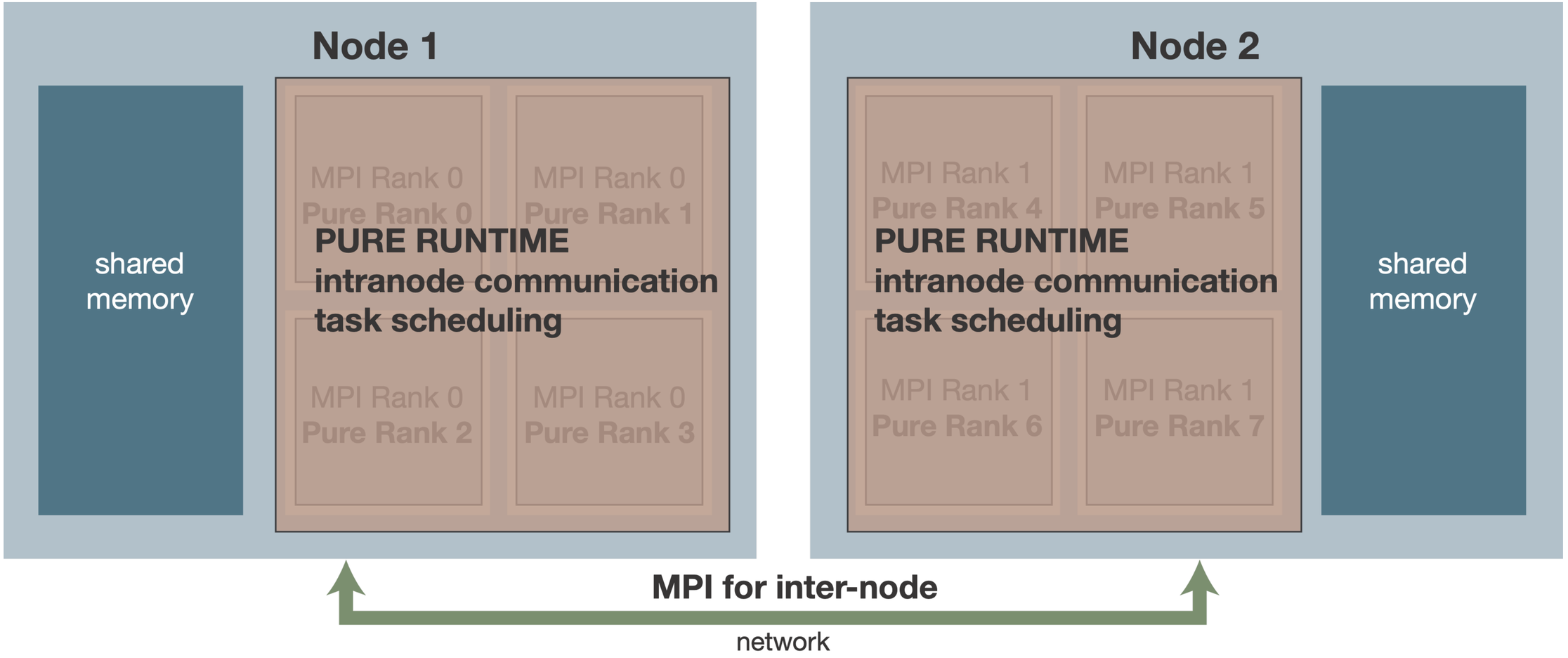
Rank to Core Mapping: MPI



Rank to Core Mapping: Pure



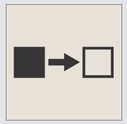
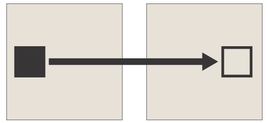
Rank to Core Mapping: Pure



4 Optimizations Driving Performance

	Optimization	Application Performance Impact	Application Change Needed?	Applicability
#1	**Faster within-node messaging**	medium	no	any point-to-point messaging
#2	**Faster collectives**	medium	no	any collectives use
#3	**Automatic work stealing of tasks**	large	yes	when load imbalance and partitionable work
#4	**Helper threads**	small	no	when Pure Tasks are used and unused cores exists

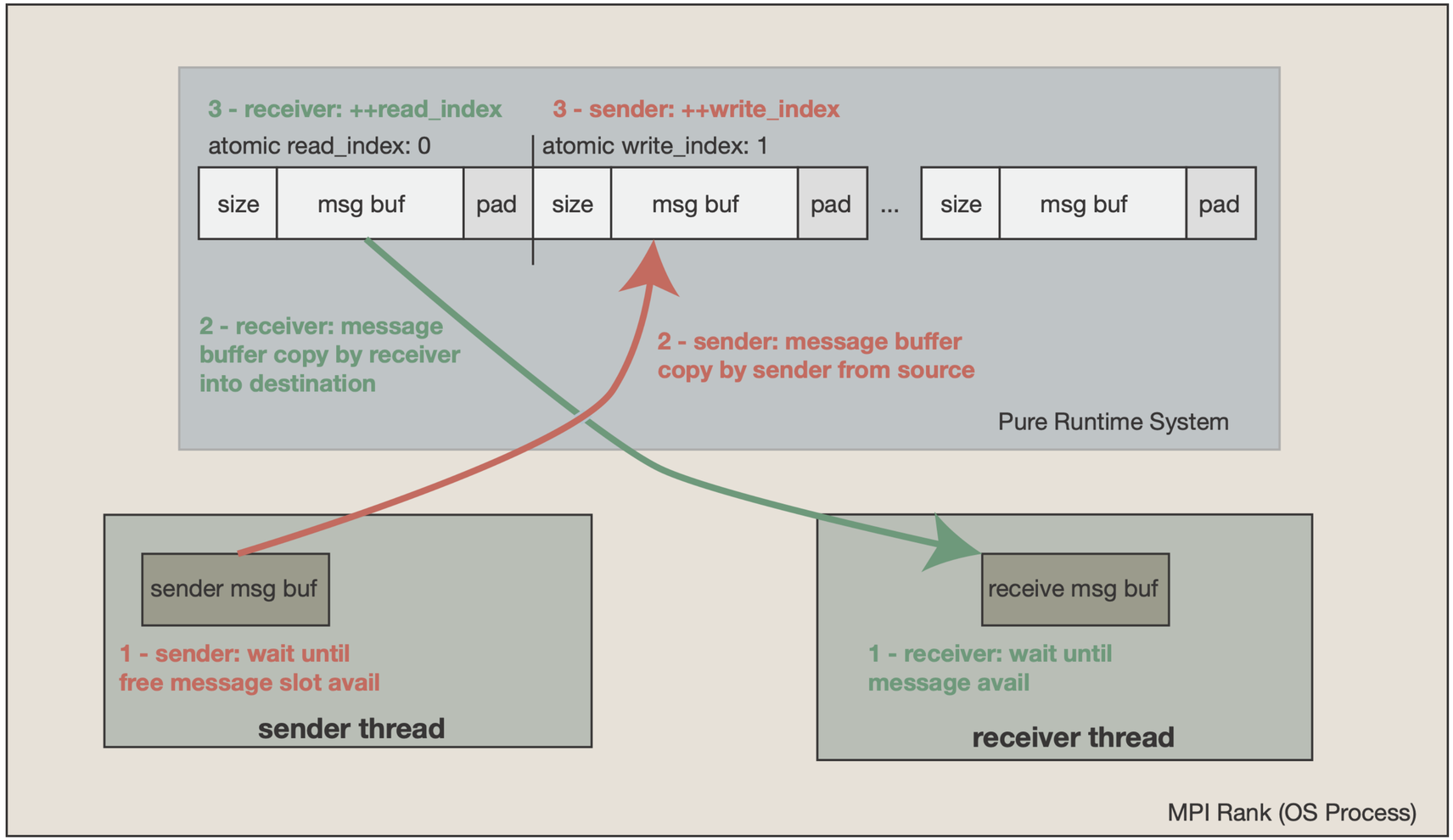
Pure Uses 3 Different Messaging Strategies

		Small Messages	**Large Messages**
		msg	msg
	Sender-Receiver on **Same Node**	Buffered protocol, `PureBufferQueue`	Rendezvous protocol, lock-free queue sync
	Sender-Receiver on **Different Nodes**	MPI with thread-pair encoded in tag high bits	

…but user interface is always the same

Small Messages on Same Node

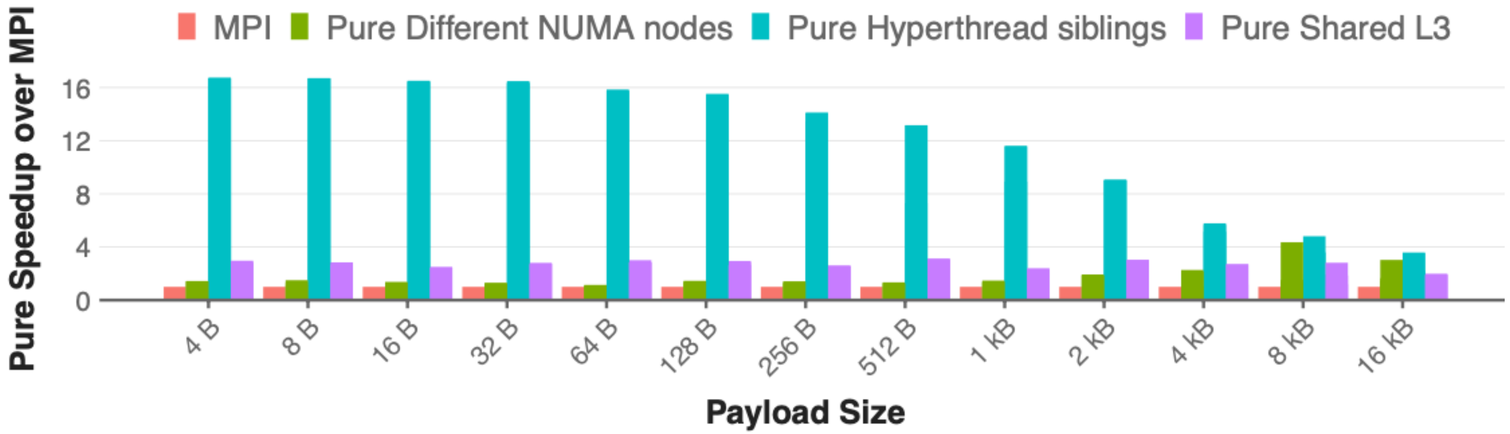
- Lock-free `PureBufferQueue` in Pure Runtime with reused inline buffers
- Two atomic index circular buffer
- Acquire-release memory ordering, x86 `pause` in loops, aligned and padded buffers, configurable sizes
- `PureBufferQueue` outperformed other designs using std::mutex, buffer pool design, etc.



Same-Node Flood Benchmark

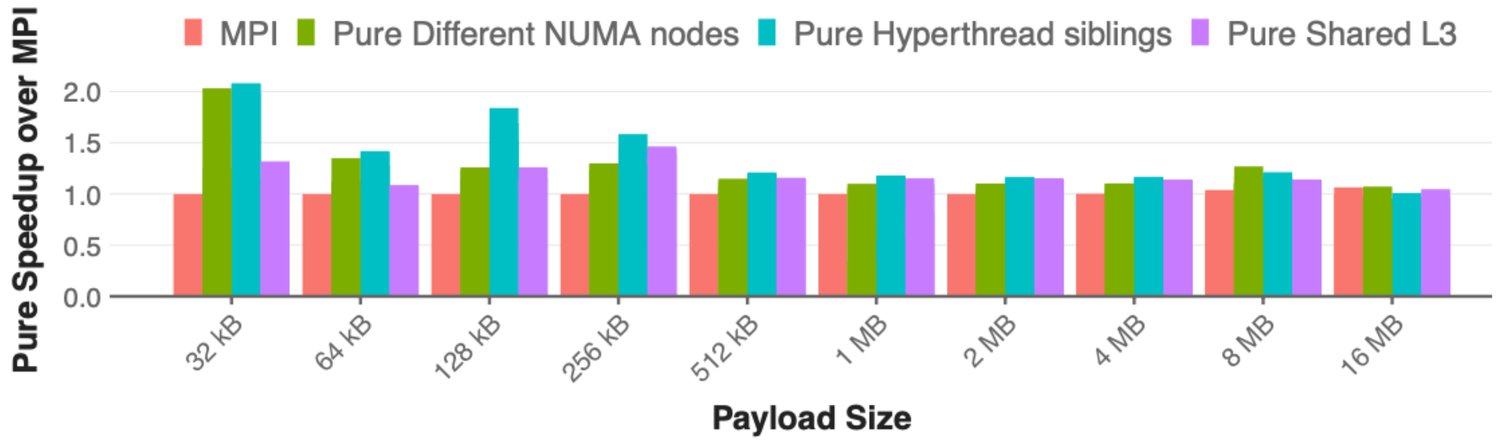
4B — 16kB messages

up to 17x speedup

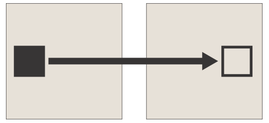


32B — 16MB messages

up to 2x speedup

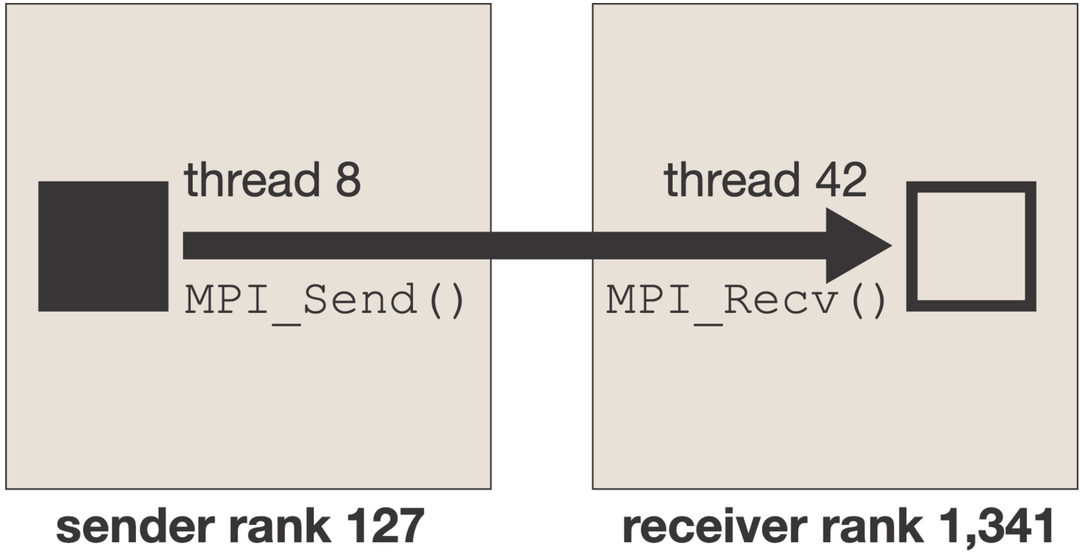


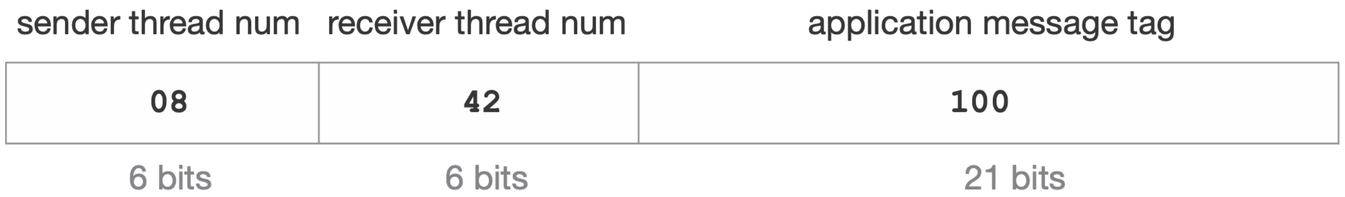
Pure outperforms MPI for all payload sizes and rank placements



Messaging Across Nodes

- Inter-node messaging: uses MPI_Send / MPI_Recv
- Encode thread number in tag high bits
- `MPI_THREAD_MULTIPLE`
- Also have an experimental GASnet version
- No speedup across nodes





Encoded Message Tag

walk through animation?

Optimizations Driving Performance

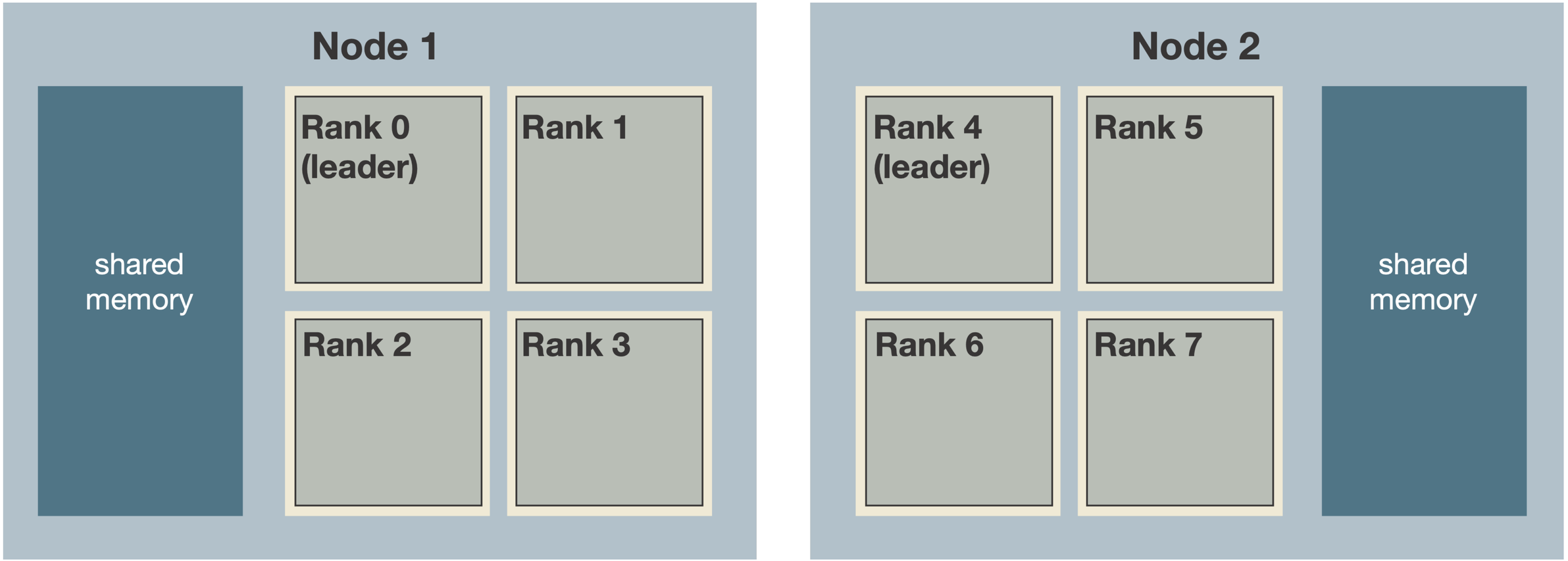
	Optimization	Application Performance Impact	Application Change Needed?	Applicability
#1	**Faster within-node messaging**	medium	no	any point-to-point messaging
#2	**Faster collectives**	medium	no	any collectives use
#3	**Automatic work stealing of tasks**	large	yes	when load imbalance and partitionable work
#4	**Helper threads**	small	no	when Pure Tasks are used and unused cores exists

Pure Collectives Overview

- Implements broadcast, barrier, reduce, all-reduce, with the same semantics as MPI analogs
- Performance optimizations are within nodes across threads in shared memory
- Uses custom lock-free collective algorithms / data structures that outperform both MPI DMAPP and OpenMP analogs
- Transparently uses MPI between nodes
- Automatically try to steal task chunks when blocking

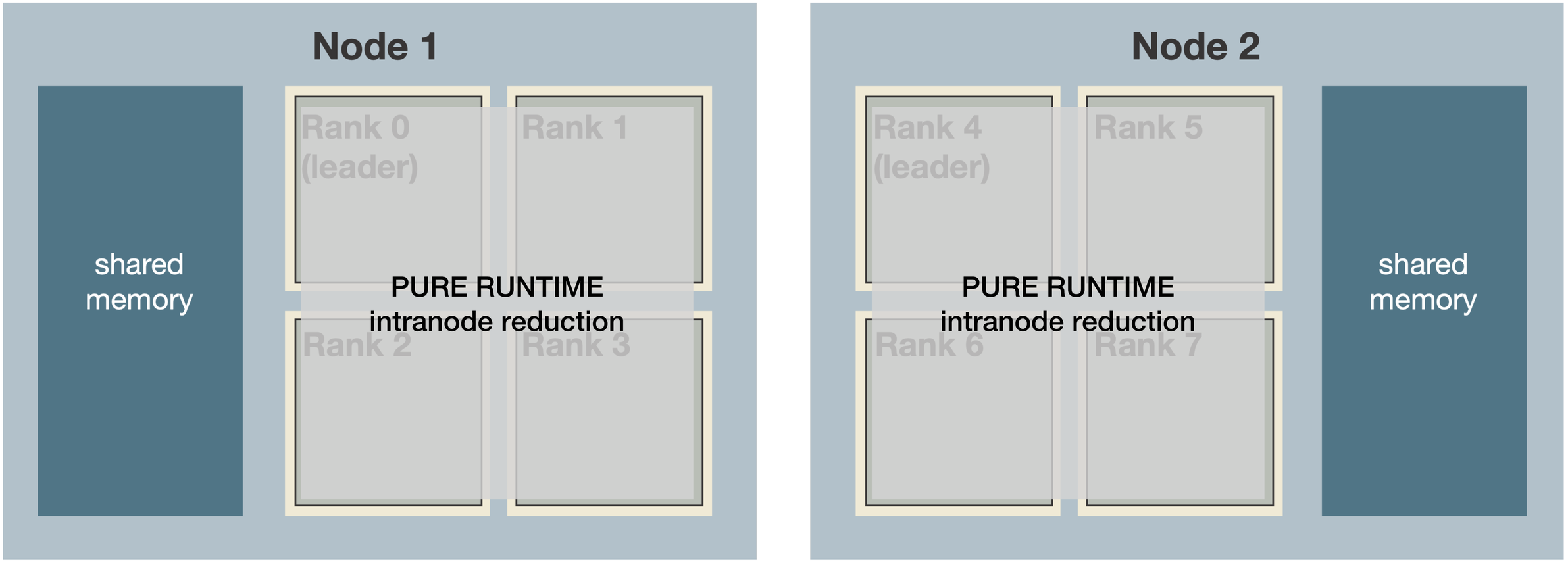
Hierarchical All-Reduce Approach

2 node, 8 rank example



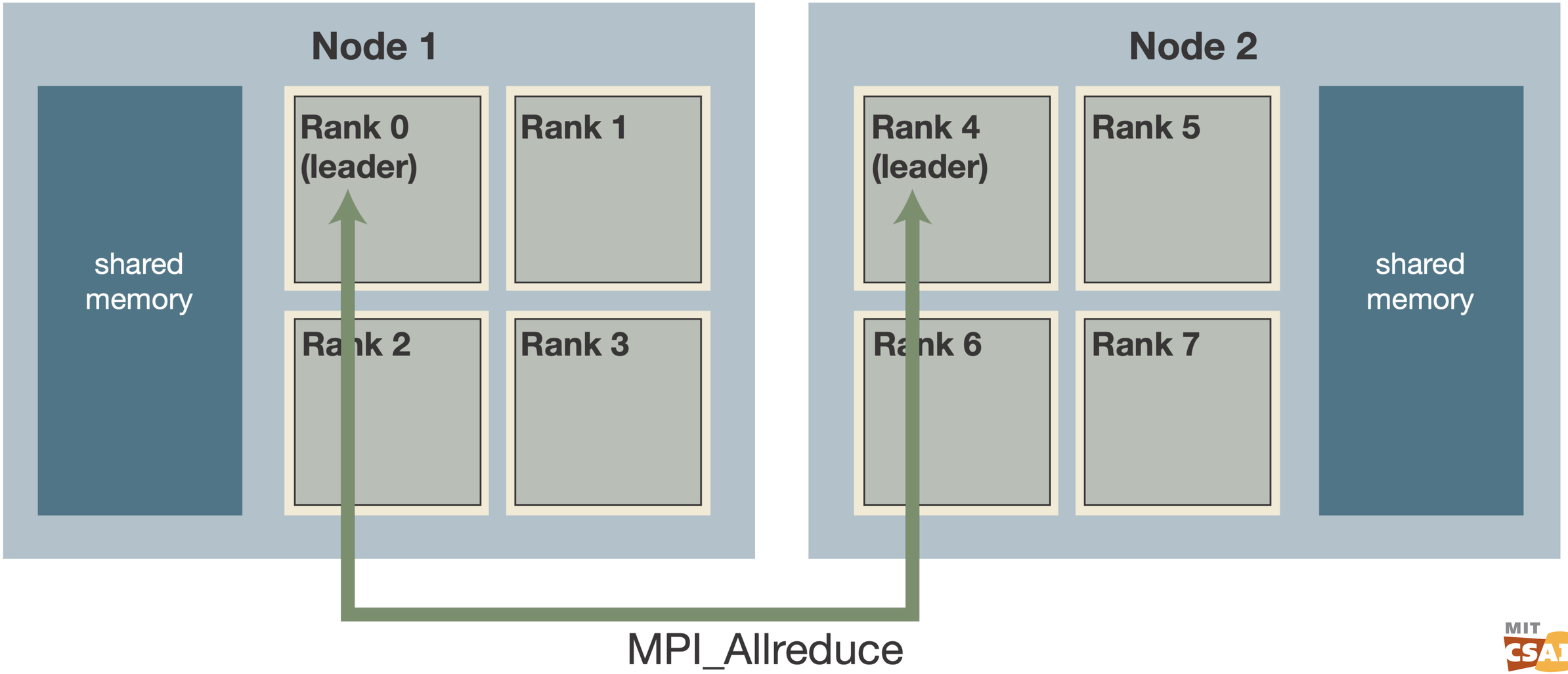
Hierarchical All-Reduce Approach

step 1: do reduction within each node (in Pure)



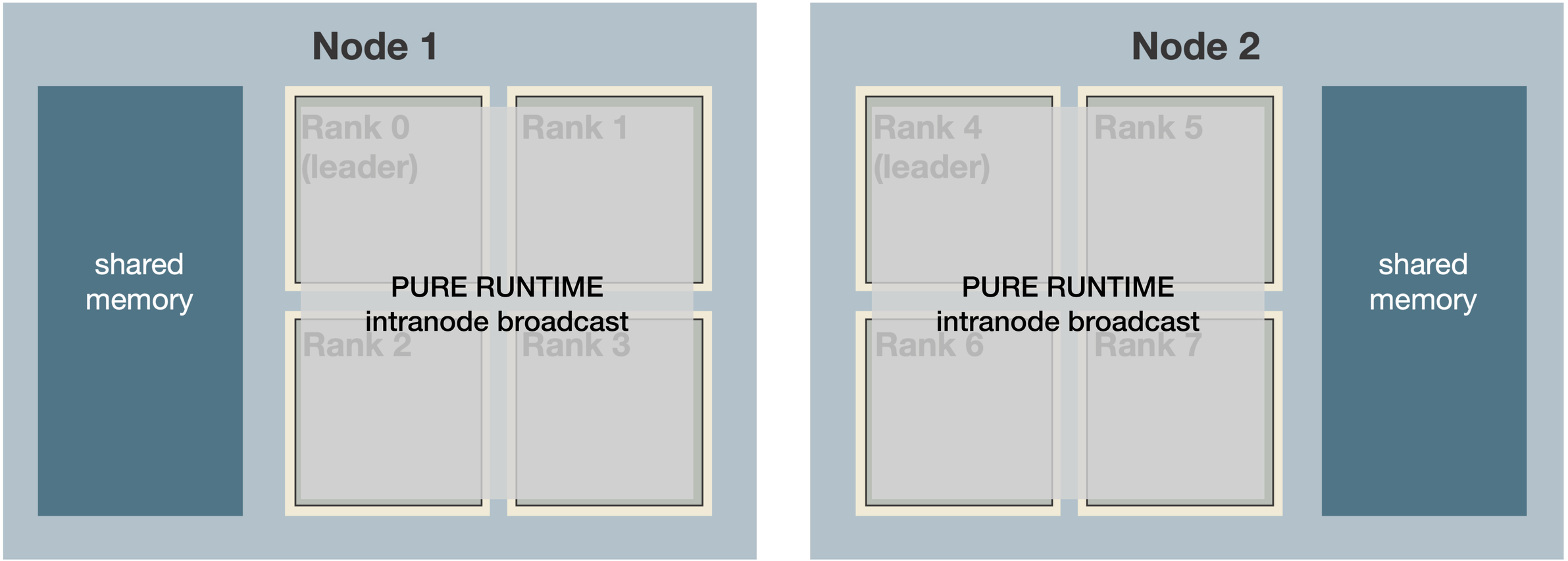
Hierarchical All-Reduce Approach

step 2: one thread per node does `MPI_Allreduce`

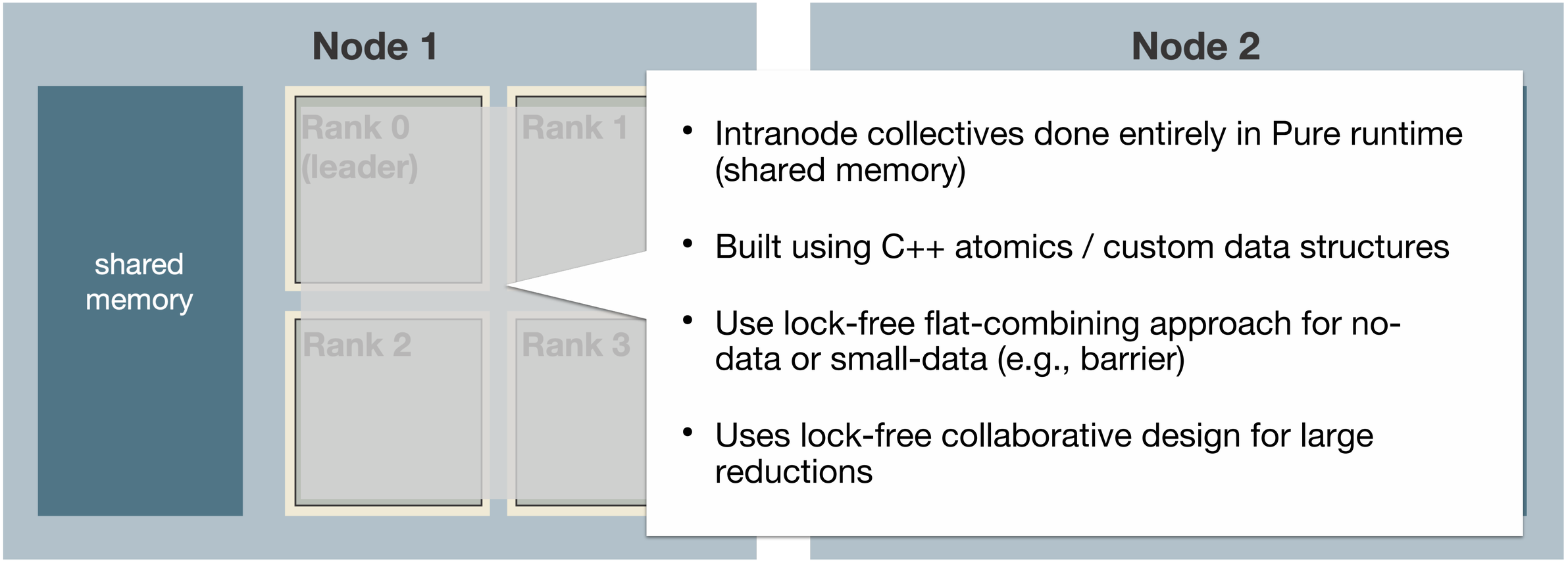


Hierarchical Allreduce Approach

step 3: do broadcast within each node (in Pure)

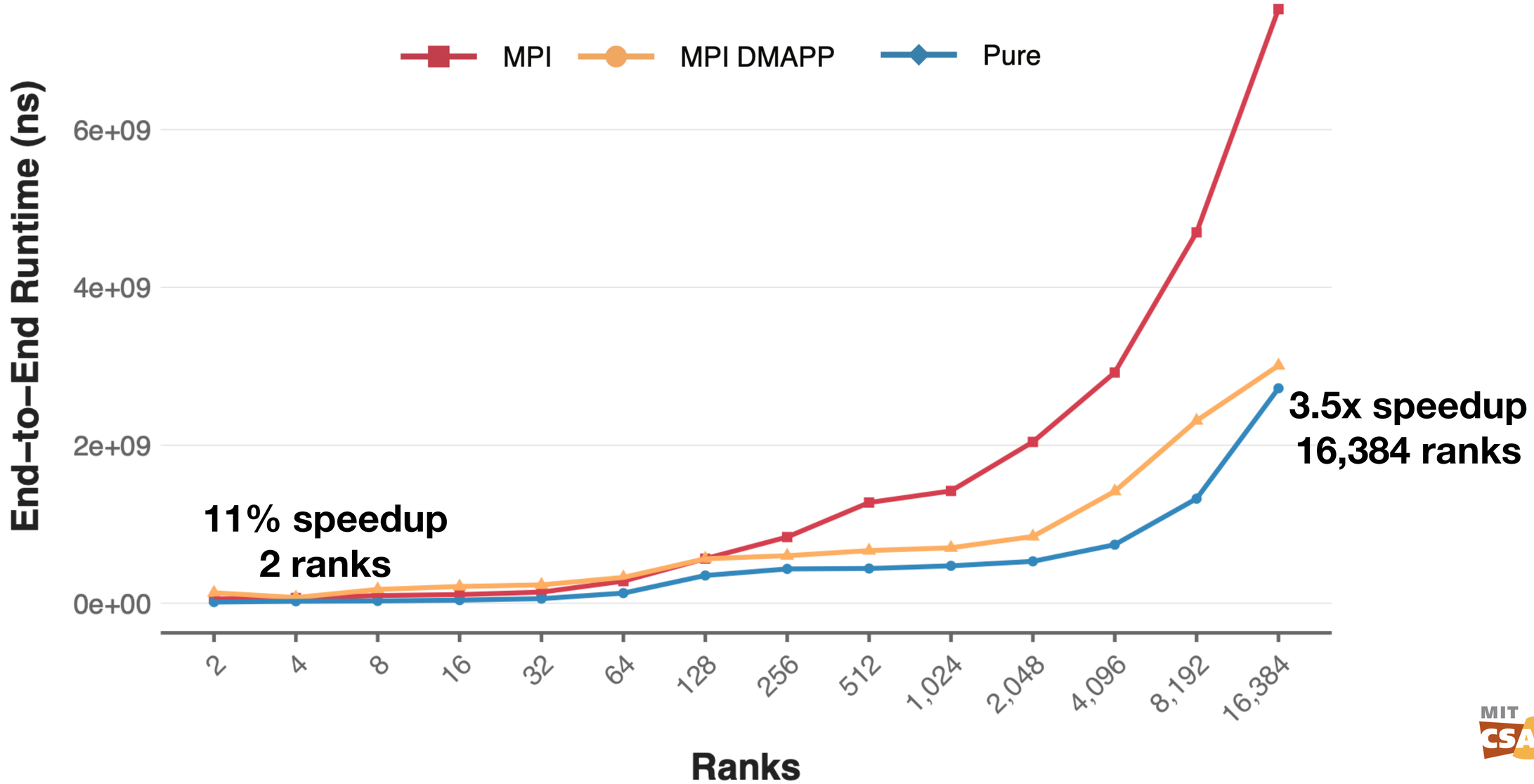


Hierarchical All-Reduce Approach

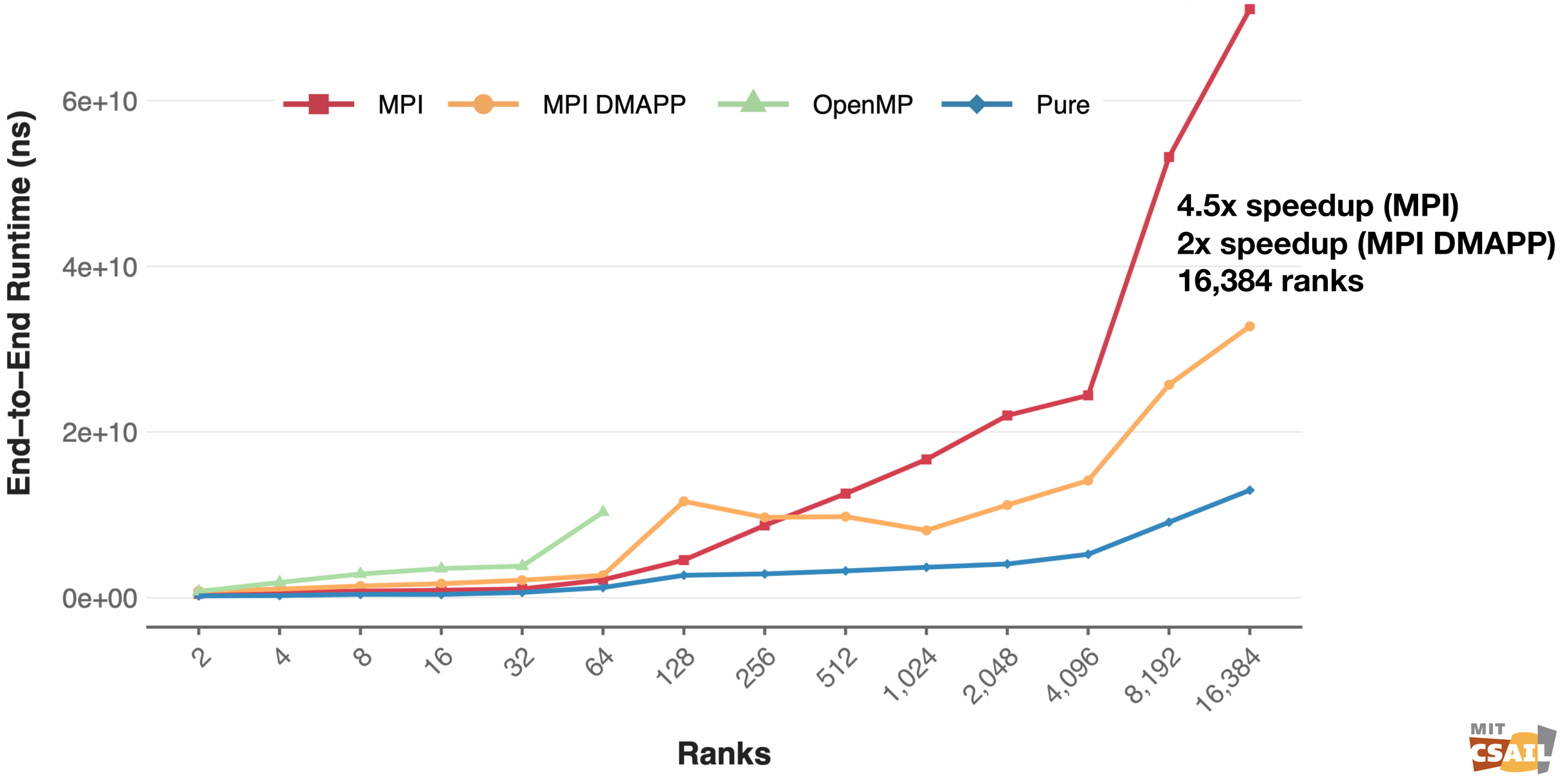


- Intranode collectives done entirely in Pure runtime (shared memory)
- Built using C++ atomics / custom data structures
- Use lock-free flat-combining approach for no-data or small-data (e.g., barrier)
- Uses lock-free collaborative design for large reductions

8B All-Reduce, 2—16k Ranks



Barrier: 2 – 16k ranks



Pure Collectives Speedups

Faster collectives: compared to Cray MPICH with XPMEM & DMAPP,
2 – 65,536 ranks

- 1.5x — 4.5x speedup for **barrier**
- 1.1x — 7x speedup for **broadcast**
- 1.5x — 100x speedup for **reduction**
- 1.1x — 4.5x speedup for **all-reduce**

Evaluation

Our Questions

1. Do we outperform MPI on real applications?
2. From where does the performance improvement come?
3. How challenging is programming in Pure? Porting from MPI?

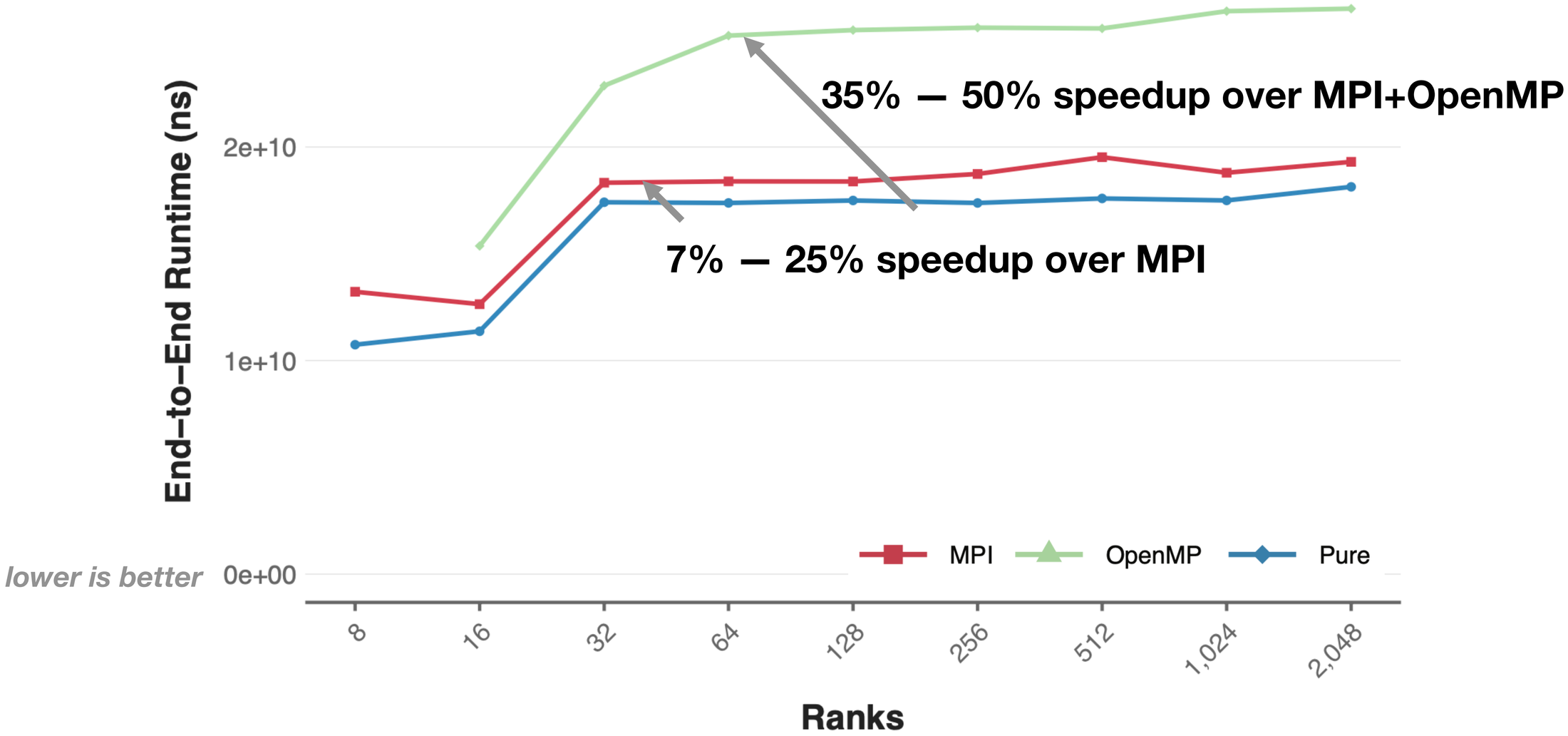
Experimental Setup

- **NERSC Cori** supercomputer at Berkeley Lab: Cray XC40, 30 PF peak performance. 2,388 nodes
- Each node: 2 sockets (NUMA regions), 16 core 2.3 GHz Haswell v3 processor, 128 GB memory, interconnected with Cray Aires
- Each core: 2 hyper-threads, and has two 256-bit-wide vector units
- Experiments generally run with 2 processes per node (one per NUMA region), 32 threads each
- Toolchain: Intel compiler, with Cray MPICH with DMAPP and XPMEM (baseline) (gcc and clang also supported)

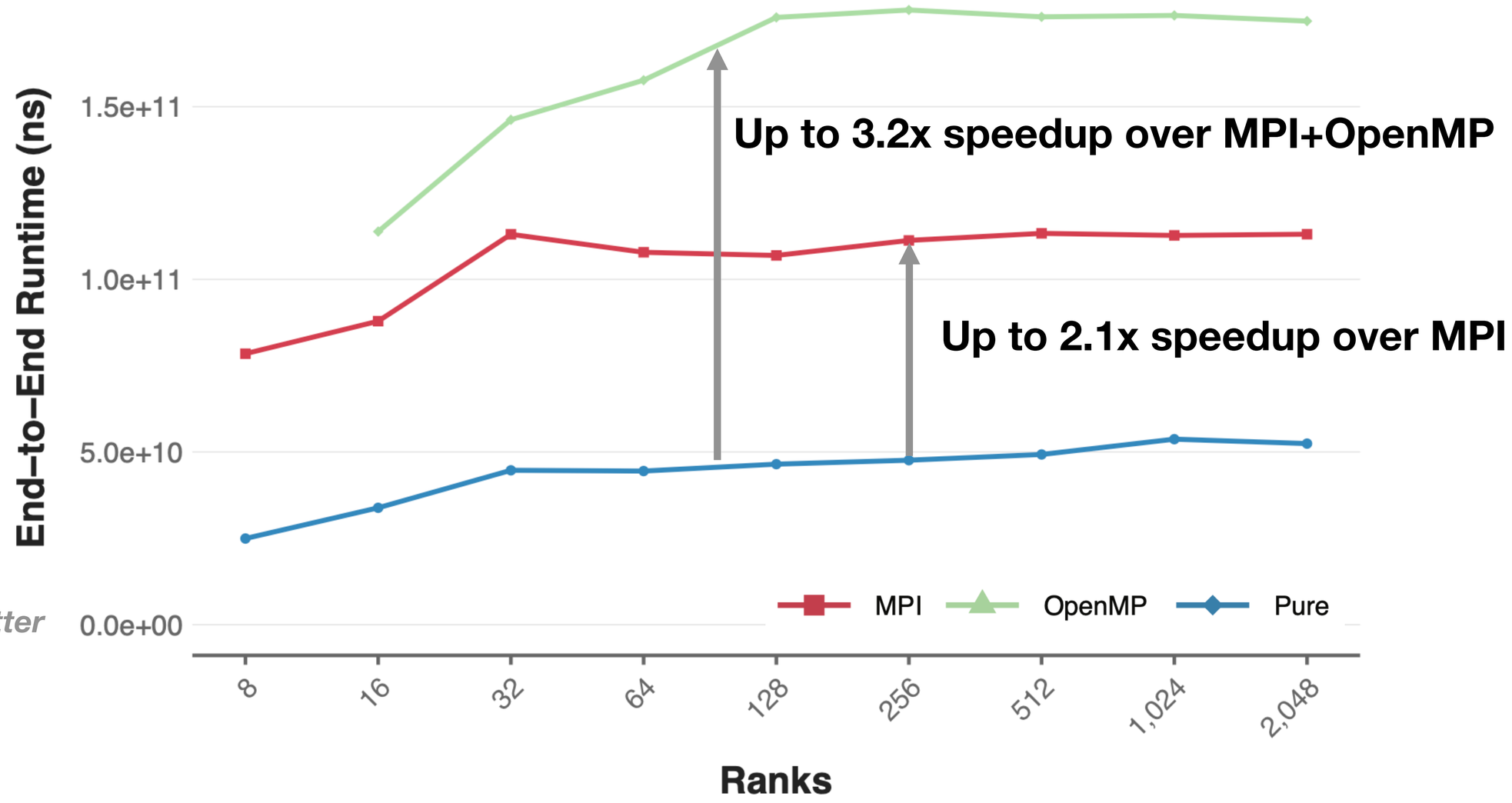
We Used Recommended MPI Baseline

- We compare Pure to the gold standard, recommended high performance MPI configuration on NERSC Cori:
  - **Cray XPMEM**: Linux kernel module that enables a process to map the memory of another process into its virtual address space
  - **Cray DMAPP**: special collective enhancements for 4-16B messages (MPI_Allreduce; MPI_Alltoall; MPI_Barrier)
  - **Cray UGNI** module includes optimizations for Aries network
  - **Hugepages** (2MB) enabled
  - **Rank reordering** for optimal rank-to-node layout via CrayPat profiler (MPI and Pure)
- Pure uses `MPI_THREAD_MULTIPLE`; baseline uses `MPI_THREAD_SINGLE`

CoMD (no Load Imbalance)

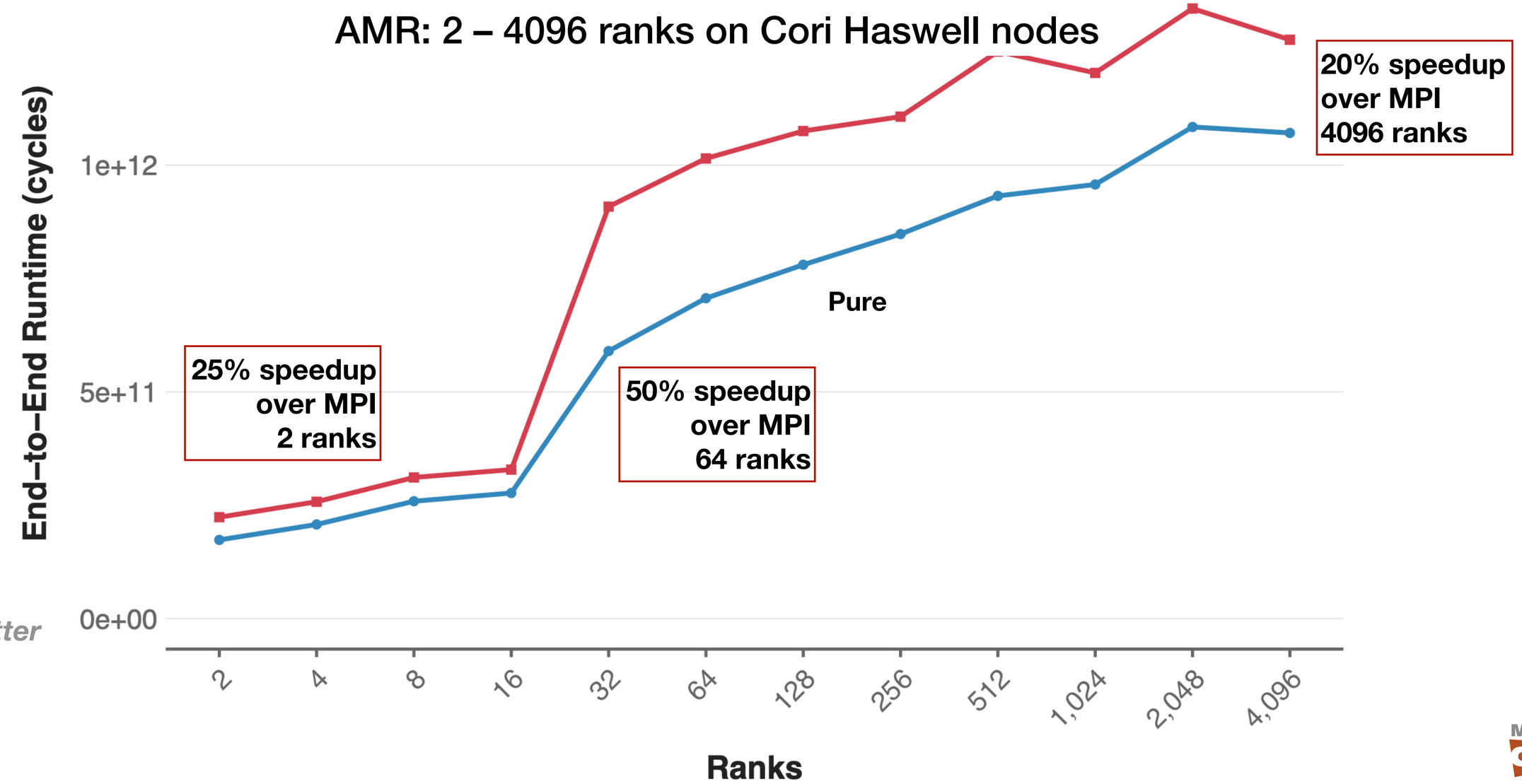


CoMD (with Load Imbalance)



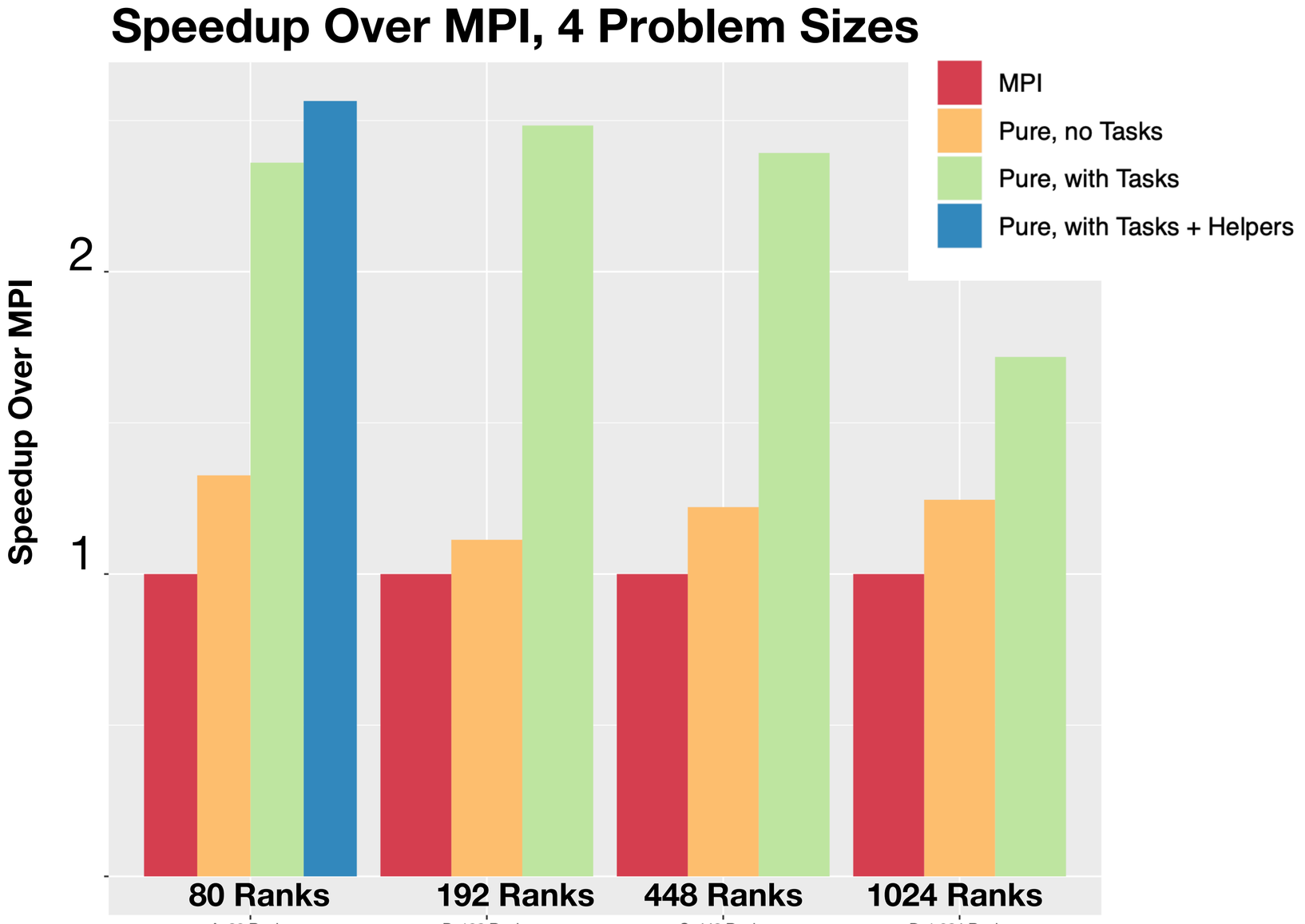
lower is better

miniAMR: 20% - 50% Speedup over MPI



NAS DT: 80 – 1,024 ranks

- NAS DT exhibits significant load imbalance
- 11 – 25% speedup from faster messages alone
- 1.7x – 2.6x speedup after adding Pure Tasks
- Helper threads on unused cores automatically added 9% speedup (had 24 spare cores per node)



Writing Programs with Pure

- Programming Pure feels mostly like programming MPI-only (one application rank per core, one runtime)
- Porting existing programs MPI to use Pure messaging calls:
  - single-digit hours per application
  - no global variables (use `thread_local`?)
- DT: had two Pure Tasks
- CoMD: three Pure Tasks (MPI+OMP version had 15 OMP blocks)
- Only one task in our experiments (in CoMD) required incorporating thread-safety (used atomics)
- Source-to-source (`mpi2pure`) translator tool

Getting performance from Pure programs is not too hard (more like MPI than MPI+x)

Pure Summary

- Programmer writes MPI-like SPMD application (one rank per core)
- Messages and collectives outperform MPI — get performance with no tasks
- Optionally add Pure Tasks for hotspots / load imbalance (C++ lambdas)
- Automatically uses application ranks as temporary helper threads when blocking on communication
- Implementation: 22k SLOC C++17, lock-free. Modern C++ interface
- 20% - 2.5x speedup on 3 miniapps

Pure: Evolving Message Passing To Better Leverage Shared Memory Within Nodes

James Psota
Armando Solar-Lezama