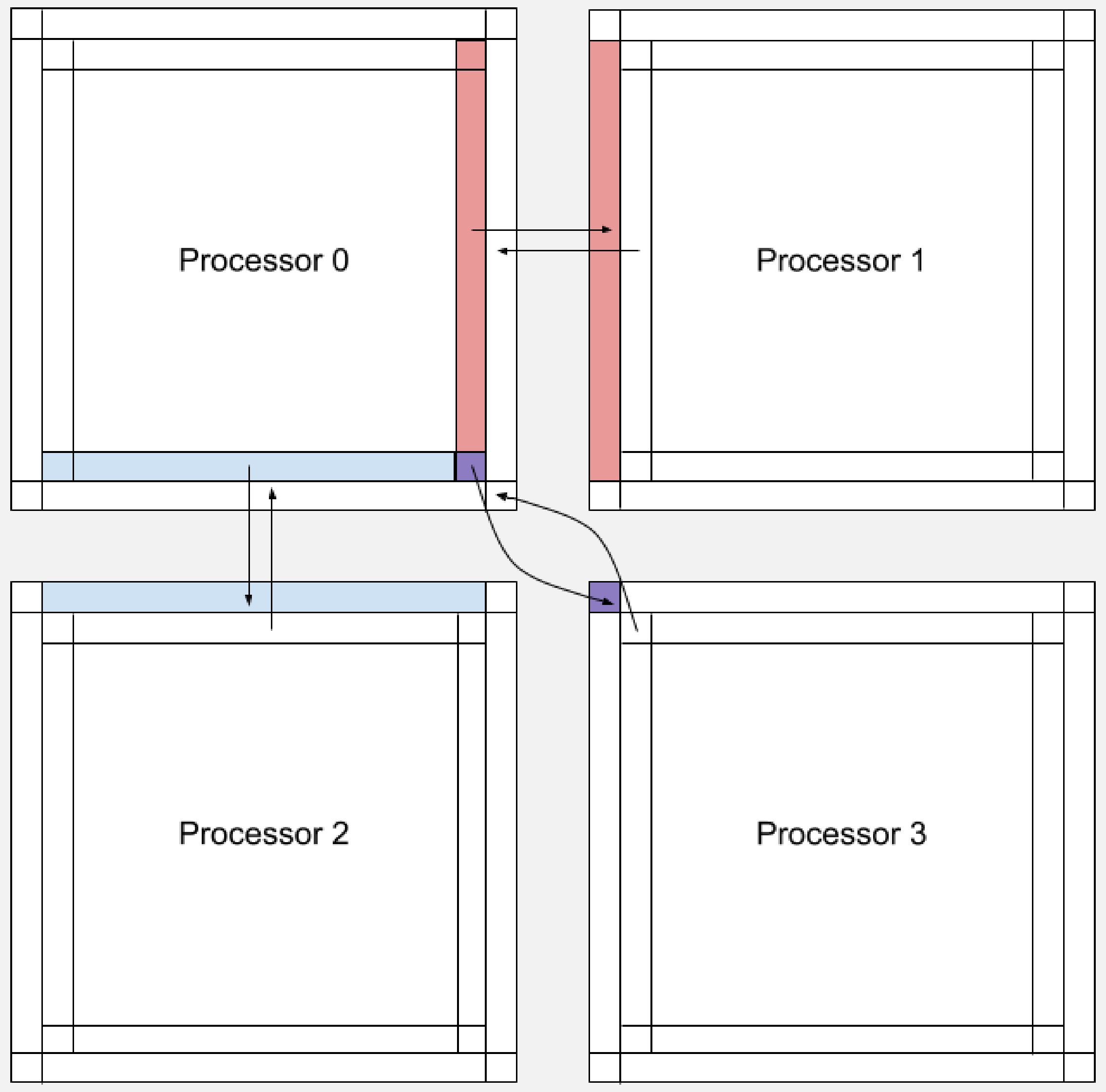
Center for Understandable
Performant Exascale
Communication Systems

THE UNIVERSITY OF
NEW MEXICO

Persistent and Partitioned MPI for Stencil Communication

Gerald Collom and Amanda Bienz, UNM Dept. of Computer Science

Background



Stencil Communication

- Halo Exchange: the communication of boundary values between neighboring processes
- Packing: copying data into a contiguous message buffer

Persistent MPI

- Amortizes setup costs by e.g. caching arguments

Partitioned MPI

- Partition messages to mark portions as ready to send or check if received
- MPI + threads
- Early work, early communication

Algorithms

Point-to-Point Baseline

```
Algorithm 1: Exchange
Input: msgs_info, requests
    {message information (process pairs, data sources), requests storage array}

parallel region                                              {OpenMP Region}
    // Copy data from mesh to contiguous message buffer
    pack(msgs_info)

// Begin communication
for $i \leftarrow 0$ to $n_{send}$ do
    MPI_Isend(msgs_info[$i$], requests[i])
for $i \leftarrow 0$ to $n_{recv}$ do
    MPI_Irecv(msgs_info[$i + n_{send}$],
     requests[$i + n_{send}$])

// Wait on messages
MPI_Waitall(requests)

parallel region
    // Copy values from message buffer into mesh
    unpack(msgs_info, msgs_info)
```

Persistent MPI Interfaces

1) Initialization:
   - Provide message info
   - Once before exchanges
   - Returns persistent request
2) Communication exchanges:
   - Start persistent request and wait for completion
   - Repeat for each exchange
3) Cleanup:
   - Free persistent requests

Persistent MPI

```
Algorithm 2: Persistent Init
Input: msgs_info, requests
    {message information (process pairs, data sources), requests storage array}

// Initialize messages
for $i \leftarrow 0$ to $n_{send}$ do
    MPI_Send_init(msgs_info[$i$],
     requests[$i$])
for $i \leftarrow 0$ to $n_{recv}$ do
    MPI_Recv_init(msgs_info[$i + n_{send}$],
     requests[$i + n_{send}$])
```

```
Algorithm 3: Persistent Exchange
Input: msgs_info, requests
    {message information (process pairs, data sources), requests storage array}

parallel region                                              {OpenMP Region}
    // Copy data from mesh to contiguous message buffer
    pack(msgs_info)

// Begin communication
MPI_Startall(requests)

// Wait on messages
MPI_Waitall(requests)

parallel region
    // Copy values from message buffer into mesh
    unpack(msgs_info)
```

```
Algorithm 4: Persistent Destroy
Input: requests                                     {requests storage array}
for $i \leftarrow 0$ to $n_{recv} + n_{send}$ do
    MPI_Request_free(requests[$i$])
```

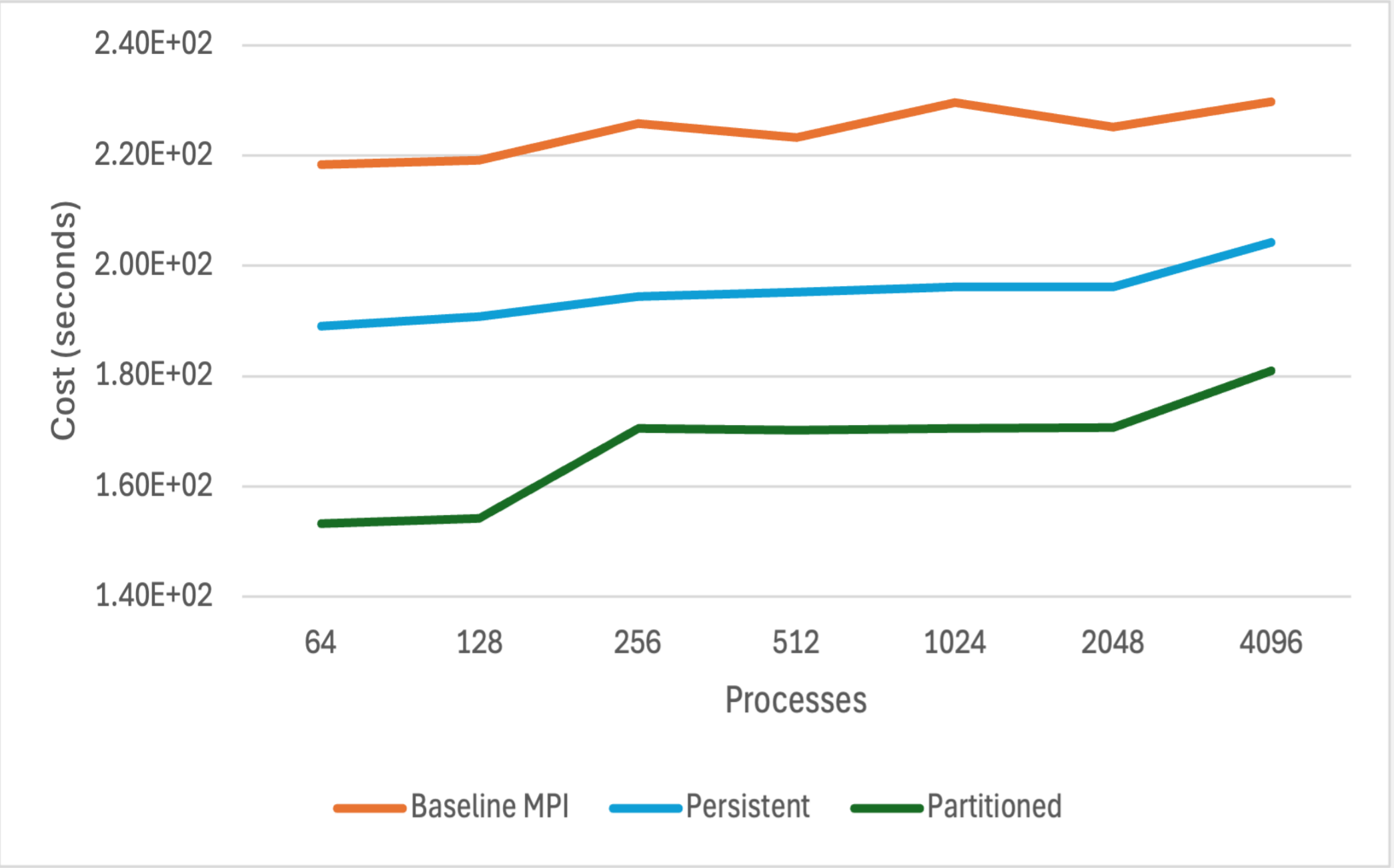
Partitioned MPI

```
Algorithm 5: Partitioned Init
Input: n_parts, msgs_info, requests
    {number of partitions, message information (process pairs, data sources),
  requests storage array}

// Initialize messages
for $i \leftarrow 0$ to $n_{send}$ do
    MPI_Psend_init(n_parts,
     msgs_info[$i$], requests[$i$])
for $i \leftarrow 0$ to $n_{recv}$ do
    MPI_Precv_init(n_parts,
     msgs_info[$i + n_{send}$],
     requests[$i + n_{send}$])
```

```
Algorithm 6: Partitioned Exchange
Input: msgs_info, requests
    {message information (process pairs, data sources), requests storage array}

// Begin communication
MPI_Startall(requests)

parallel region                                              {OpenMP Region}
    // Copy data from mesh to contiguous message buffer
    pack(msgs_info)
    // Mark partition of current thread as ready
    MPI_Pready(partition)

// Wait on messages
MPI_Waitall(requests)

parallel region
    // Copy values from message buffer into mesh
    unpack(msgs_info)
```

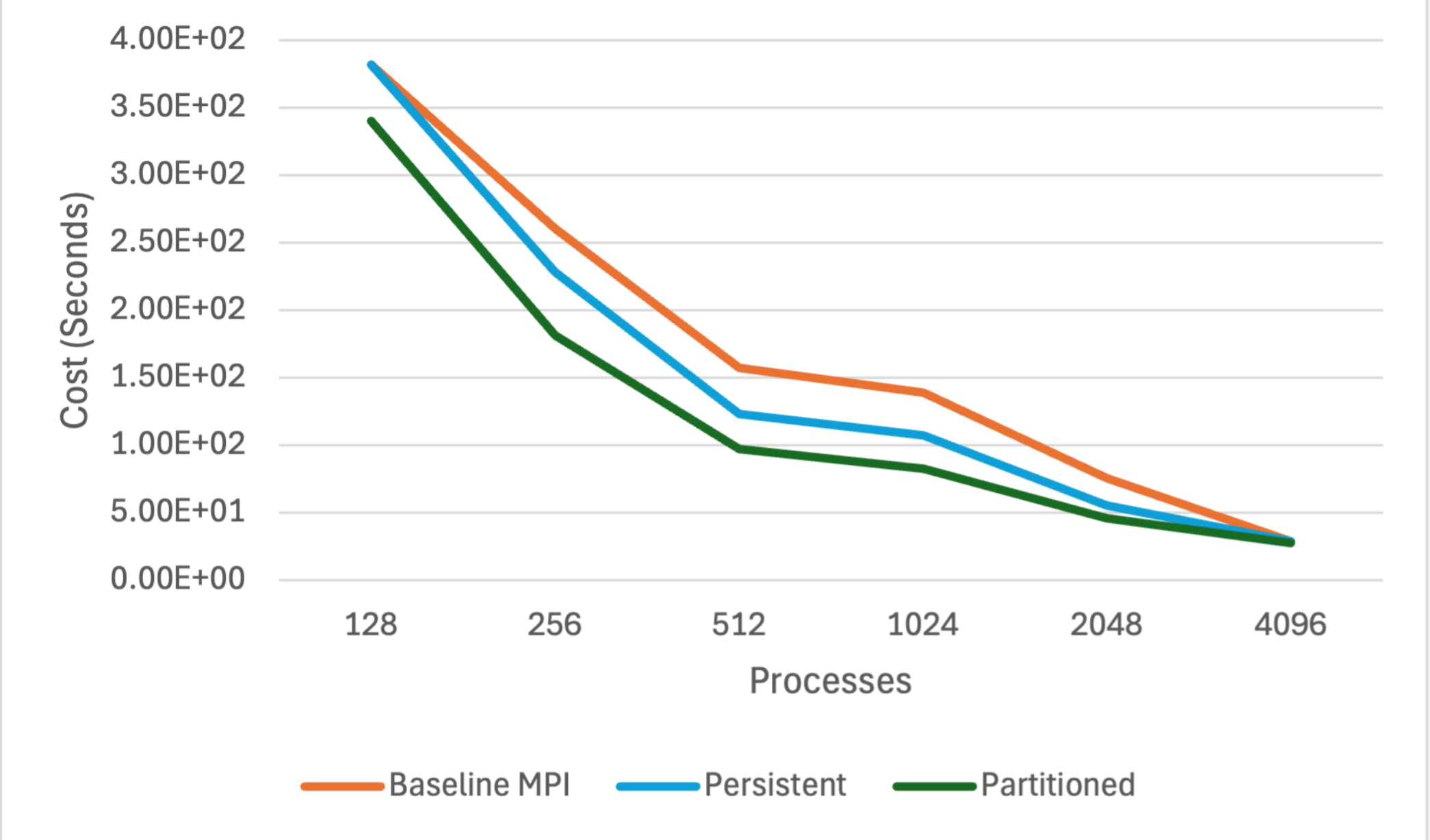
```
Algorithm 7: Partitioned Destroy
Input: requests                                     {requests storage array}
for $i \leftarrow 0$ to $n_{recv} + n_{send}$ do
    MPI_Prequest_free(requests[$i$])
```

Results



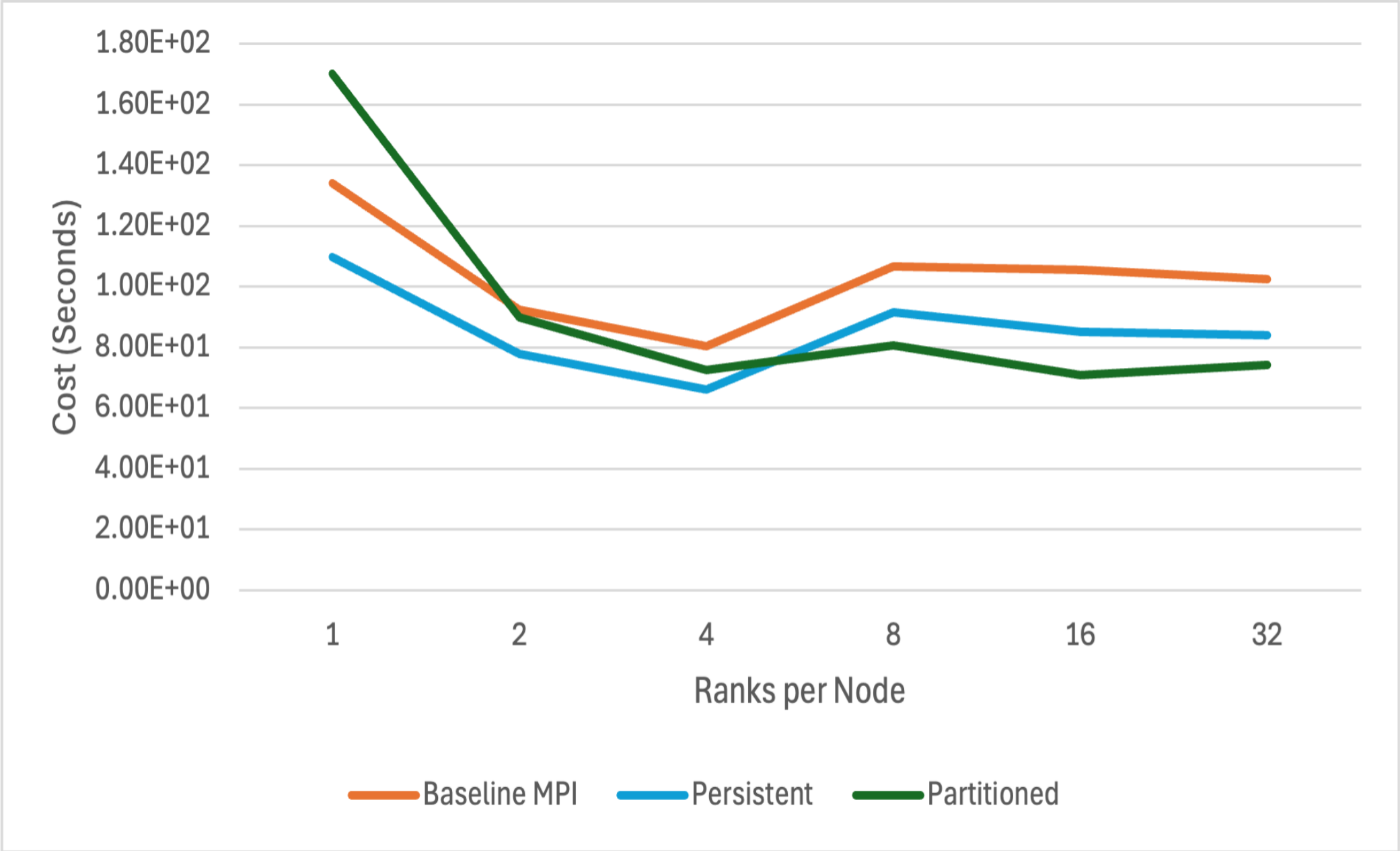
Weak Scaling

- MPIPCL, System: Quartz (Intel Xeon system at LLNL)
- Timed 1000 exchanges, 3 runs
- One core, two threads per MPI Process
- 512^3 cells per process, 3 mesh variables
- messages sizes of 524,288 doubles
- Persistent MPI: 16% speedup
- Partitioned MPI: 42% speedup



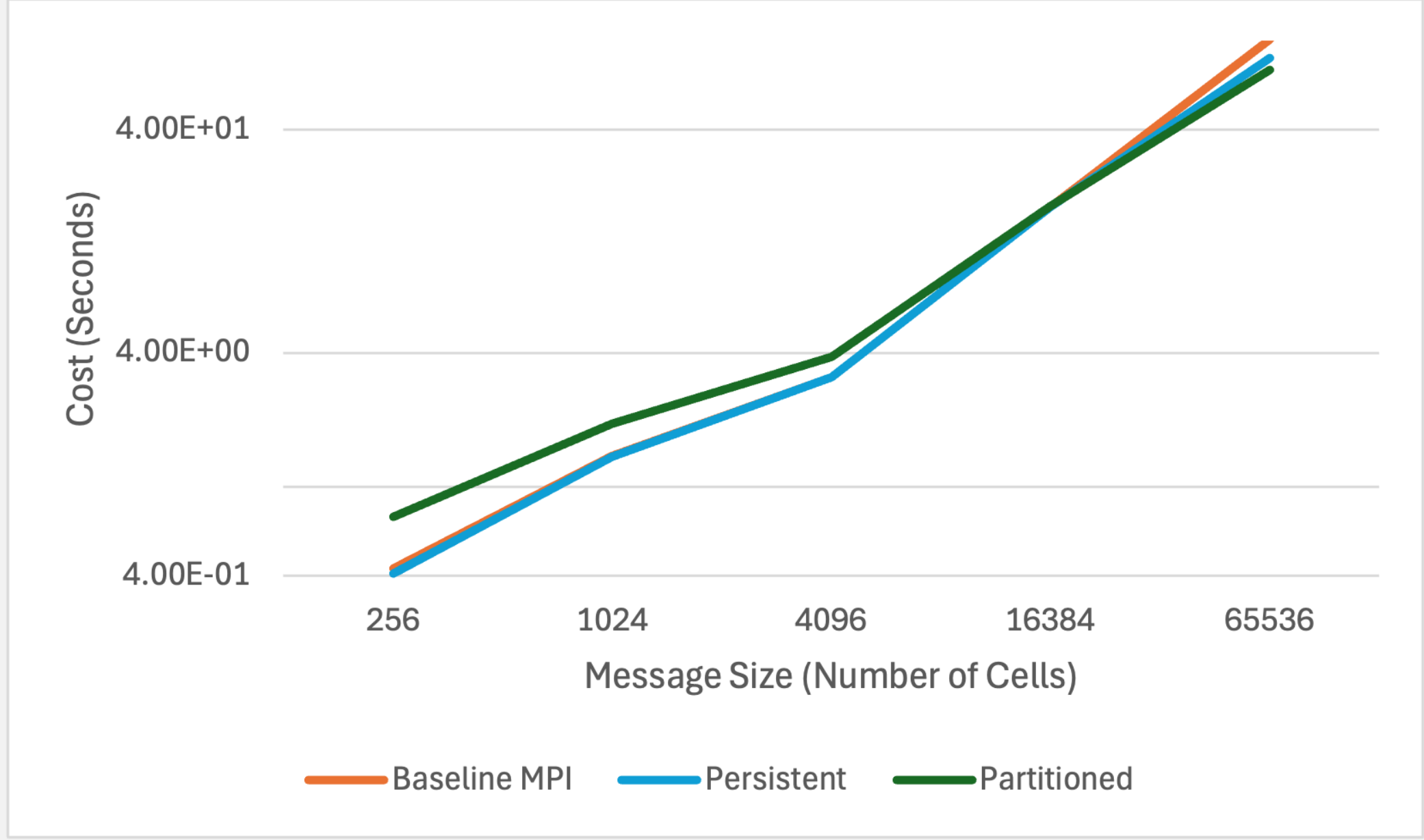
Strong Scaling

- 2048^3 cell problem with message sizes of 260k doubles, decreasing with scale
- Persistent MPI: 37% speedup
- Partitioned MPI: up to 68% total speedup



Scaling Ranks/Node

- 64 nodes, 32 active cores per node, 64 OpenMP threads per node
- 2048 x 4096 x 4096 cells
- Higher ranks/node → lower threads/rank
- Persistent MPI: ~20% speedup
- Partitioned MPI: slowdown before overtaking both other methods



Message Size Scaling

- 4096 processes on 128 nodes
- Persistent MPI: matches baseline for smaller messages, 21% faster for largest tested message
- Partitioned MPI: baseline is 73% faster for smaller messages, partitioned MPI is 37% faster for larger messages