Distributed memory parallelization of Lax-Wendroff Flux Reconstruction

Devansh Tripathi
(SoM, IISER TVM)

Collaborator: Dr. Arpit Babbar
Advisor: Prof. Praveen Chandrashekar

TIFR CAM,
Bangalore, India

August 11, 2024

Outline

The presentation is consists of following parts:

1. Flux Reconstruction

Outline

The presentation is consists of following parts:

1. Flux Reconstruction
2. Lax-Wendroff Flux Reconstruction

Outline

The presentation is consists of following parts:

1. Flux Reconstruction
2. Lax-Wendroff Flux Reconstruction
3. Parallelization of TrixiLW.jl

Outline

The presentation is consists of following parts:

1. Flux Reconstruction
2. Lax-Wendroff Flux Reconstruction
3. Parallelization of TrixiLW.jl
4. Results

Outline

The presentation is consists of following parts:

1. Flux Reconstruction
2. Lax-Wendroff Flux Reconstruction
3. Parallelization of TrixiLW.jl
4. Results
5. MPI Remote Memory Access

Flux Reconstruction

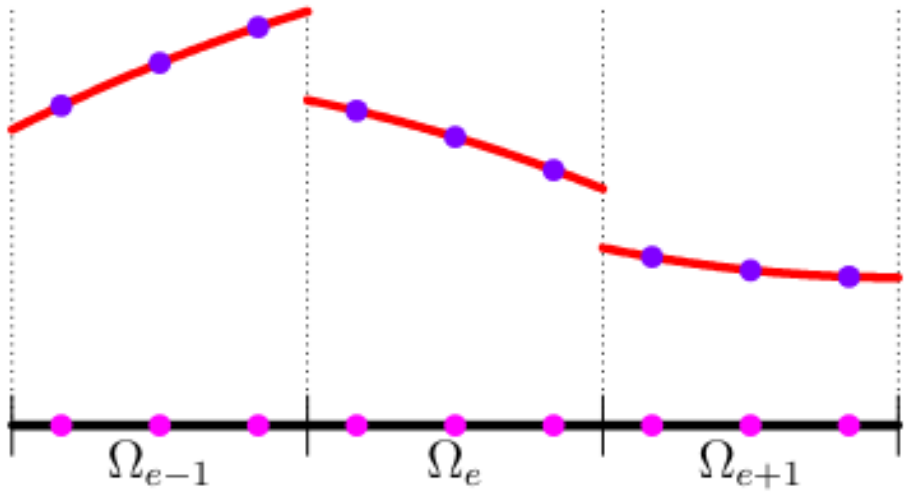
Conservation law in 1D

$$u_t + f(u)_x = 0$$

Flux Reconstruction

Conservation law in 1D

$$u_t + f(u)_x = 0$$



Flux Reconstruction

Main idea:

1. We need a continuous approximation to flux function with the help of *correction function* and discontinuous flux.

Flux Reconstruction

Main idea:

1. We need a continuous approximation to flux function with the help of *correction function* and discontinuous flux.
2. Use a collocation method to update nodal solution values.

Flux Reconstruction

Main idea:

1. We need a continuous approximation to flux function with the help of *correction function* and discontinuous flux.
2. Use a collocation method to update nodal solution values.

Discontinuous flux approximation f_h^δ is a polynomial of degree N:

$$f_h^\delta(\xi, t) = \sum_{j=0}^{N} f(u_j^e(t)) l_j(\xi) \tag{1}$$

where each $l_j(\xi)$ is Lagrange polynomial of degree N and u are the solution values (unknowns).

Flux Reconstruction

Main idea:

1. We need a continuous approximation to flux function with the help of *correction function* and discontinuous flux.
2. Use a collocation method to update nodal solution values.

Discontinuous flux approximation f_h^δ is a polynomial of degree N:

$$f_h^\delta(\xi, t) = \sum_{j=0}^{N} f(u_j^e(t)) l_j(\xi) \tag{1}$$

where each $l_j(\xi)$ is Lagrange polynomial of degree N and u are the solution values (unknowns).
$\xi \in [0, 1]$ is a point in reference element.

$$x \to \xi = \frac{x - x_{e-\frac{1}{2}}}{\Delta x_e}$$

Flux Reconstruction

Continuous flux approximation f_h with the help of correction function g_L and g_R:

$$f_h(\xi,t) = \left[f_{e-\frac{1}{2}}(t) - f_h^{\delta}(0,t)\right] g_L(\xi) + f_h^{\delta}(\xi,t) + \left[f_{e+\frac{1}{2}}(t) - f_h^{\delta}(1,t)\right] g_R(\xi) \tag{2}$$

where $f_{e-\frac{1}{2}}(t)$ is numerical flux function.

Flux Reconstruction

Continuous flux approximation f_h with the help of correction function g_L and g_R:

$$f_h(\xi,t) = \left[f_{e-\frac{1}{2}}(t) - f_h^{\delta}(0,t)\right] g_L(\xi) + f_h^{\delta}(\xi,t) + \left[f_{e+\frac{1}{2}}(t) - f_h^{\delta}(1,t)\right] g_R(\xi) \tag{2}$$

where $f_{e-\frac{1}{2}}(t)$ is numerical flux function.

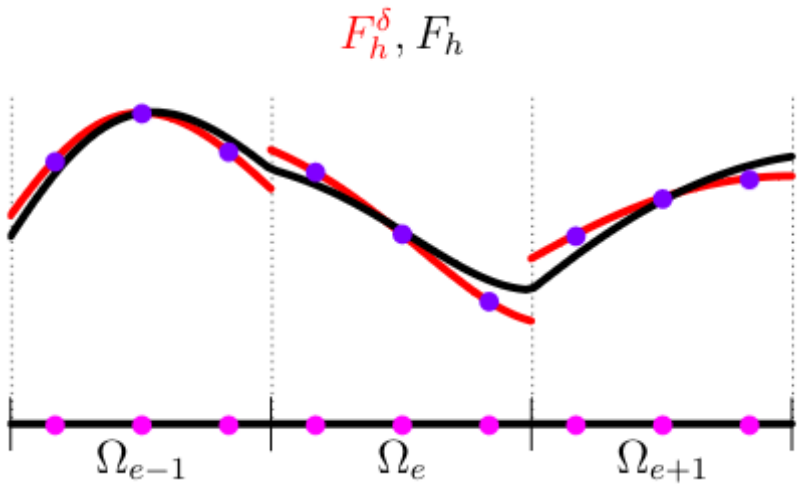


Figure: Discontinuous and continuous flux[1]

Correction function

Correction function should satisfy the following conditions:

- For domain $[0, 1]$:

$$g_L(0) = 1, \qquad g_R(0) = 0$$
$$g_L(1) = 0, \qquad g_R(1) = 1$$

Correction function

Correction function should satisfy the following conditions:

- For domain $[0, 1]$:

$$g_L(0) = 1, \qquad g_R(0) = 0$$
$$g_L(1) = 0, \qquad g_R(1) = 1$$

- Correction function should approximate 0 in some sense.

Example of correction functions: *Radau* and $g2$.

LWFR

- Single stage, high-order accurate method.

LWFR

- Single stage, high-order accurate method.
- Using Taylor expansion of u around $t = t^n$ and we use PDE, $\partial_t u = -\partial_x f$, to write the time derivatives as spatial derivatives.

$$\partial_t^m u = -\partial_t^{m-1} \partial_x f = -(\partial_t^{m-1} f)_x$$

LWFR

- Single stage, high-order accurate method.
- Using Taylor expansion of u around $t = t^n$ and we use PDE, $\partial_t u = -\partial_x f$, to write the time derivatives as spatial derivatives.

$$\partial_t^m u = -\partial_t^{m-1} \partial_x f = -(\partial_t^{m-1} f)_x$$

Taylor's expansion

$$\begin{aligned} u^{n+1} &= u^n + \sum_{m=1}^{N+1} \frac{(\Delta t)^m}{m!} \partial_t^m u^n + O(\Delta t^{N+2}) \\ &= u^n - \sum_{m=1}^{N+1} \frac{(\Delta t)^m}{m!} (\partial_t^{m-1} f)_x + O(\Delta t^{N+2}) \\ &= u^n - \Delta t \frac{\partial F}{\partial x}(u^n) + O(\Delta t^{N+2}) \end{aligned}$$

$$x \to \xi = \frac{x - x_{e-\frac{1}{2}}}{\Delta x_e}$$

- Update step looks like this:

$$(u_j^e)^{n+1} = (u_j^e)^n - \frac{\Delta t}{\Delta x_e} \frac{dF_h}{d\xi}(\xi_j), \quad 0 \leq j \leq N \tag{3}$$

where F_h is the continuous polynomial, used to reconstruct the time average flux F inside each element.

$$x \to \xi = \frac{x - x_{e-\frac{1}{2}}}{\Delta x_e}$$

- Update step looks like this:

$$(u_j^e)^{n+1} = (u_j^e)^n - \frac{\Delta t}{\Delta x_e} \frac{dF_h}{d\xi}(\xi_j), \quad 0 \leq j \leq N \tag{3}$$

where F_h is the continuous polynomial, used to reconstruct the time average flux F inside each element.

$$F(u^n) = \sum_{m=0}^{N} \frac{\Delta t^m}{(m+1)!} \partial_t^m f(u)$$

is an approximation to time average flux in the interval $[t^n, t^{n+1}]$
Order of accuracy in both **space** and **time**: $N + 1$

Solvers: TrixiLW and Trixi

`TrixiLW.jl` is an adaptive high-order numerical simulation framework for 2D conservation laws based on `Trixi.jl`[2].

Solvers: TrixiLW and Trixi

`TrixiLW.jl` is an adaptive high-order numerical simulation framework for 2D conservation laws based on `Trixi.jl`[2].

Features of `TrixiLW.jl`:

Solvers: TrixiLW and Trixi

`TrixiLW.jl` is an adaptive high-order numerical simulation framework for 2D conservation laws based on `Trixi.jl`[2].

Features of `TrixiLW.jl`:

- Cartesian and Curvilinear meshes

Solvers: TrixiLW and Trixi

`TrixiLW.jl` is an adaptive high-order numerical simulation framework for 2D conservation laws based on `Trixi.jl`[2].

Features of `TrixiLW.jl`:

- Cartesian and Curvilinear meshes
- Structured and Unstructured meshes

Solvers: TrixiLW and Trixi

`TrixiLW.jl` is an adaptive high-order numerical simulation framework for 2D conservation laws based on `Trixi.jl`[2].

Features of `TrixiLW.jl`:

- Cartesian and Curvilinear meshes
- Structured and Unstructured meshes
- AMR and Shock capturing

Solvers: TrixiLW and Trixi

`TrixiLW.jl` is an adaptive high-order numerical simulation framework for 2D conservation laws based on `Trixi.jl`[2].

Features of `TrixiLW.jl`:

- Cartesian and Curvilinear meshes
- Structured and Unstructured meshes
- AMR and Shock capturing
- High order accurate using LWFR

Solvers: TrixiLW and Trixi

`TrixiLW.jl` is an adaptive high-order numerical simulation framework for 2D conservation laws based on `Trixi.jl`[2].

Features of `TrixiLW.jl`:

- Cartesian and Curvilinear meshes
- Structured and Unstructured meshes
- AMR and Shock capturing
- High order accurate using LWFR
- Periodic and non-periodic boundary condition

Solvers: TrixiLW and Trixi

`TrixiLW.jl` is an adaptive high-order numerical simulation framework for 2D conservation laws based on `Trixi.jl`[2].

Features of `TrixiLW.jl`:

- Cartesian and Curvilinear meshes
- Structured and Unstructured meshes
- AMR and Shock capturing
- High order accurate using LWFR
- Periodic and non-periodic boundary condition
- Support for multi-threading

Solvers: TrixiLW and Trixi

`TrixiLW.jl` is an adaptive high-order numerical simulation framework for 2D conservation laws based on `Trixi.jl`[2].

Features of `TrixiLW.jl`:

- Cartesian and Curvilinear meshes
- Structured and Unstructured meshes
- AMR and Shock capturing
- High order accurate using LWFR
- Periodic and non-periodic boundary condition
- Support for multi-threading
- Support multi-node parallelization for Tree and P4est mesh

Idea of Parallelization

Data that needs to be communicated:

- u (solution)
- U (time average solution)
- F (time average flux)

Idea of Parallelization

Data that needs to be communicated:

- u (solution)
- U (time average solution)
- F (time average flux)

Types of interfaces:

- interface (on the same rank)
- mpi_interfaces (on different ranks)

Idea of Parallelization

Data that needs to be communicated:

- u (solution)
- U (time average solution)
- F (time average flux)

Types of interfaces:

- interface (on the same rank)
- mpi_interfaces (on different ranks)

At what places MPI communication is needed?: **mpi_interfaces**

Red lines: mpi_interfaces Grey lines: interfaces

Example Implementation

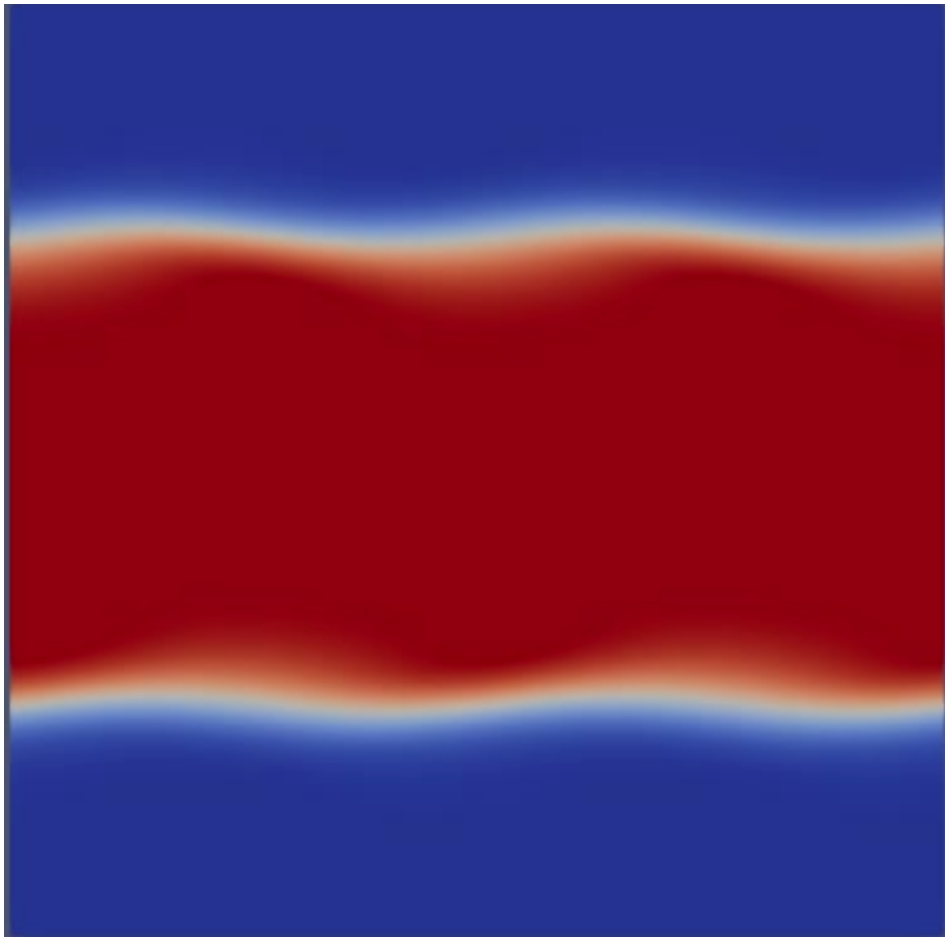
```
function rhs!(du, u, t, dt, ...)
    start_mpi_receive!(mpi_cache, ...)
    # Calculate volume integral
    calc_volume_integral!(du, u, t, dt, ... )
    # Prolong solution to MPI interfaces
    prolong2mpiinterfaces!(cache, u, mesh, equations, ...)
    start_mpi_send!(mpi_cache, mesh, equations, ...)
    # Serial computation
    # finish mpi communication
    finish_mpi_receive!(mpi_cache, mesh, equations,...)
    # Calculate MPI interface fluxes
    calc_mpi_interface_flux!(surface_flux_values,..)
    # Finish to send MPI data
    finish_mpi_send!(mpi_cache)
    return nothing
end
```

Data structures

Example `struct`:

```
mutable struct MPICache{uEltype <: Real}
    mpi_neighbor_ranks::Vector{Int}
    mpi_neighbor_interfaces::Vector{Vector{Int}}
    # contains data
    mpi_send_buffers::Vector{Vector{uEltype}}
    mpi_recv_buffers::Vector{Vector{uEltype}}
    # non-blocking communication
    mpi_send_requests::Vector{MPI.Request}
    mpi_recv_requests::Vector{MPI.Request}
    ...
end
```

Animation for Kelvin-Helmholtz



Scaling Results[1]

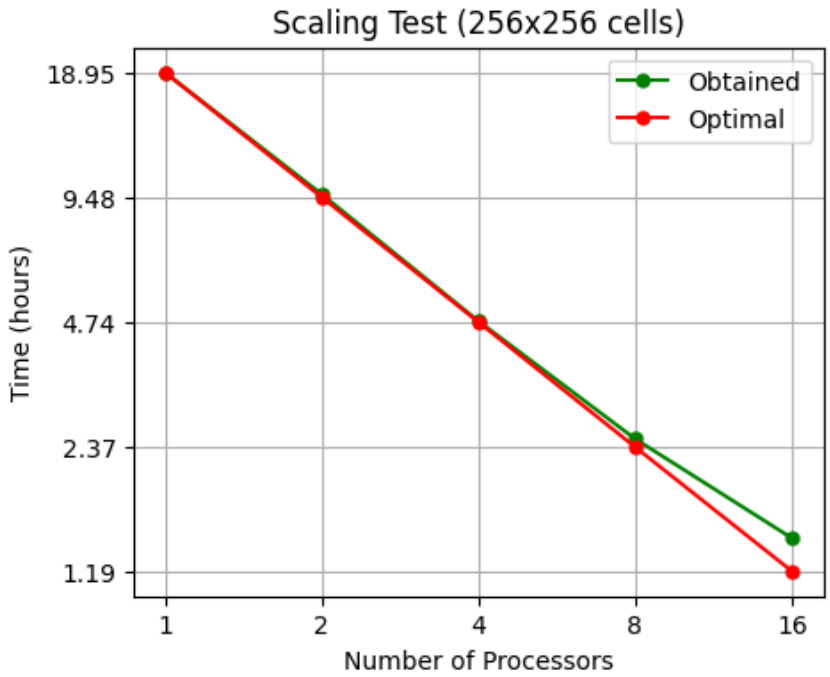


Figure: Isentropic Vortex problem

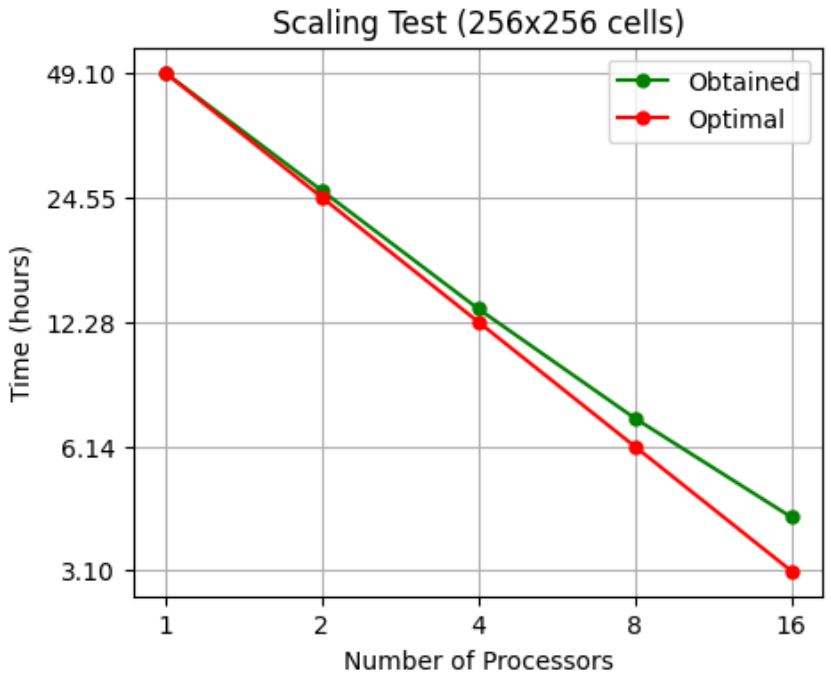


Figure: Kelvin Helmholtz problem

For isentropic vortex problem:
speed-up: 13.23
efficiency: 82.6 %

For Kelvin-Helmholtz problem:
speed-up: 11.86
efficiency: 74.12%

[1] on Dual Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz

Different Approach to parallelization: MPI RMA

One Sided Communication:

- We can move data without involving the remote process.

Different Approach to parallelization: MPI RMA

One Sided Communication:

- We can move data without involving the remote process.
- Each process exposes some part of its memory for RMA operations.

Different Approach to parallelization: MPI RMA

One Sided Communication:

- We can move data without involving the remote process.
- Each process exposes some part of its memory for RMA operations.
- Every other process can read and write data to this memory.

Different Approach to parallelization: MPI RMA

One Sided Communication:

- We can move data without involving the remote process.
- Each process exposes some part of its memory for RMA operations.
- Every other process can read and write data to this memory.
- No ordering.

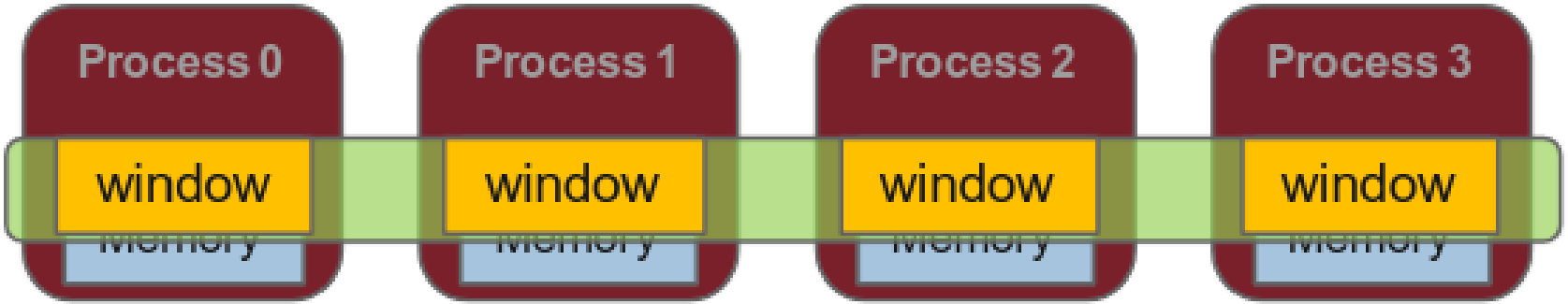


Figure: Memory Window

Window creation model

Four models exist:

- MPI_WIN_ALLOCATE
  - We want to create a buffer and directly make it remotely accessible.

Window creation model

Four models exist:

- MPI_WIN_ALLOCATE
  - We want to create a buffer and directly make it remotely accessible.
- MPI_WIN_CREATE
  - We already have an allocated buffer that we would like to make remote accessible.

Window creation model

Four models exist:

- MPI_WIN_ALLOCATE
  - We want to create a buffer and directly make it remotely accessible.
- MPI_WIN_CREATE
  - We already have an allocated buffer that we would like to make remote accessible.
- MPI_WIN_CREATE_DYNAMIC
  - Can allocate memory in future and then attach it.

Window creation model

Four models exist:

- MPI_WIN_ALLOCATE
  - We want to create a buffer and directly make it remotely accessible.
- MPI_WIN_CREATE
  - We already have an allocated buffer that we would like to make remote accessible.
- MPI_WIN_CREATE_DYNAMIC
  - Can allocate memory in future and then attach it.
  - Can dynamically add/remove buffers to/from the window.

Window creation model

Four models exist:

- MPI_WIN_ALLOCATE
  - We want to create a buffer and directly make it remotely accessible.
- MPI_WIN_CREATE
  - We already have an allocated buffer that we would like to make remote accessible.
- MPI_WIN_CREATE_DYNAMIC
  - Can allocate memory in future and then attach it.
  - Can dynamically add/remove buffers to/from the window.
- MPI_WIN_ALLOCATE_SHARED
  - We want multiple processes on the same node share a buffer.

Code Example

```
function collective_win_create(u)
    win = MPI.Win_create(u, comm)
    return win
end
```

```
function collective_win_free(win)
    MPI.free(win)
end
```

Data Movement

MPI RMA provides ability to read, write and atomically modify data in remotely accessible memory regions:

Data Movement

MPI RMA provides ability to read, write and atomically modify data in remotely accessible memory regions:

- MPI_PUT

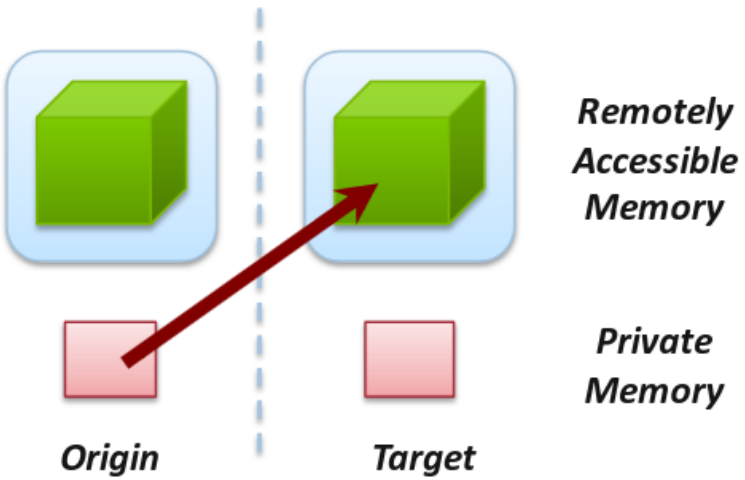


Figure: MPI_PUT[3]

Data Movement

- MPI_GET

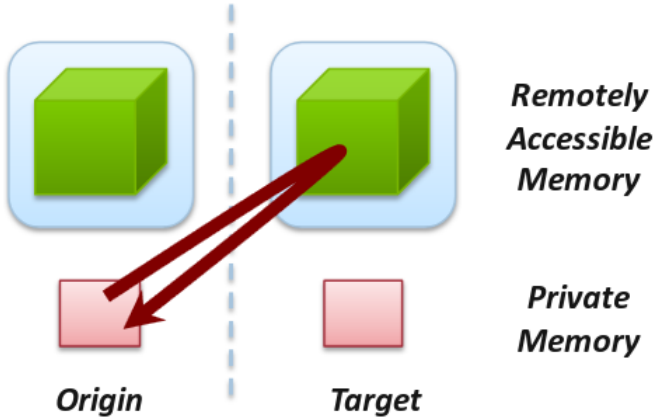


Figure: MPI_GET[3]

Data Movement

- MPI_GET

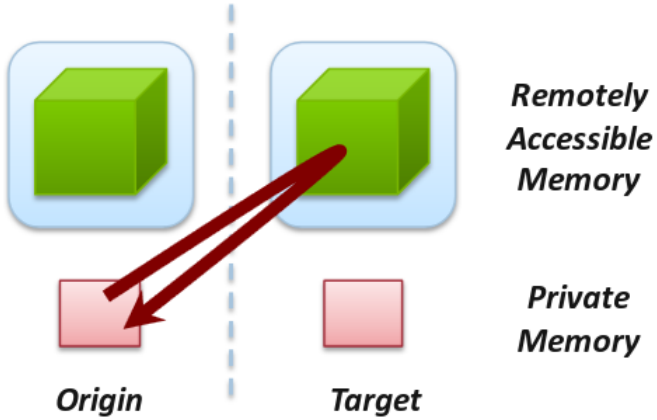


Figure: MPI_GET[3]

- MPI_ACCUMULATE (atomic)

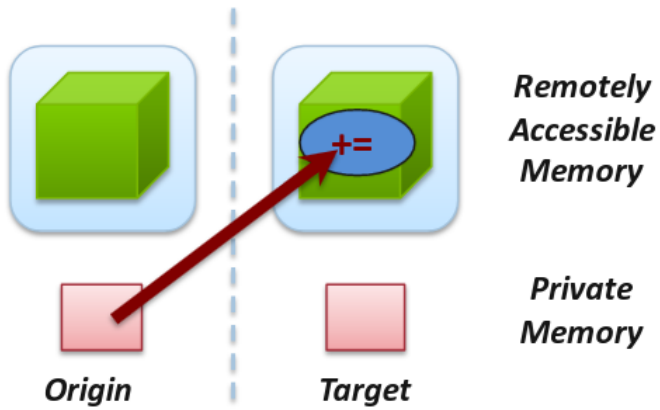


Figure: MPI_ACCUMULATE[3]

Data Movement

- MPI_GET

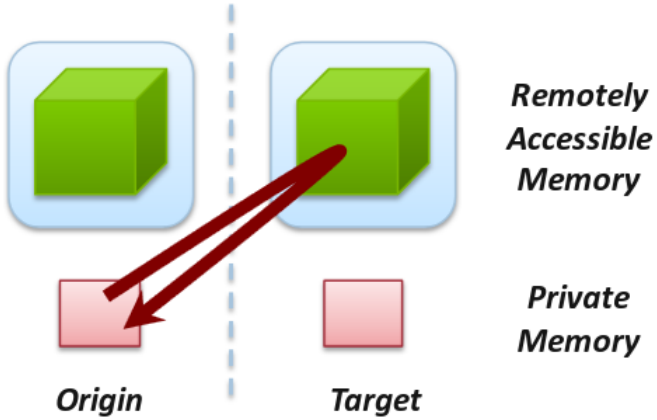


Figure: MPI_GET[3]

- MPI_ACCUMULATE (atomic)

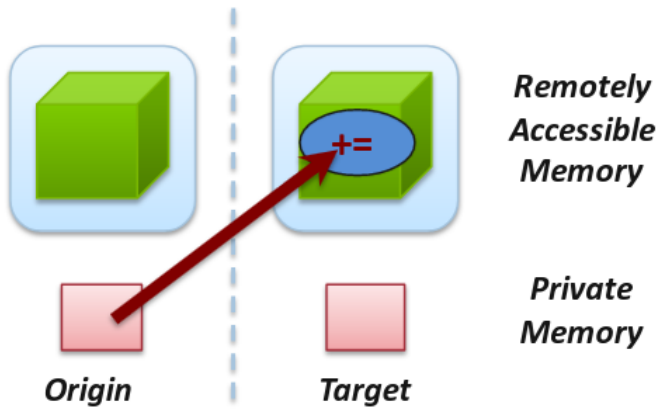


Figure: MPI_ACCUMULATE[3]

Atomic operations:

- These operations execute without the interruption of any other process in between their execution phase.

Window Synchronization Routines

Three synchronization models provided by MPI:

Window Synchronization Routines

Three synchronization models provided by MPI:

- Fence (active target)
- Lock/Unlock (passive target)
- Post-start-complete-wait (generalized active target)

Window Synchronization Routines

Three synchronization models provided by MPI:

- Fence (active target)
- Lock/Unlock (passive target)
- Post-start-complete-wait (generalized active target)

MPI_Win_fence: Active target synchronization

- Collective synchronization model.

Window Synchronization Routines

Three synchronization models provided by MPI:

- Fence (active target)
- Lock/Unlock (passive target)
- Post-start-complete-wait (generalized active target)

MPI_Win_fence: Active target synchronization

- Collective synchronization model.
- Between a pair of `MPI_Win_fence`, any number of RMA operations can be performed.

Window Synchronization Routines

Three synchronization models provided by MPI:

- Fence (active target)
- Lock/Unlock (passive target)
- Post-start-complete-wait (generalized active target)

MPI_Win_fence: Active target synchronization

- Collective synchronization model.
- Between a pair of `MPI_Win_fence`, any number of RMA operations can be performed.
- `assert` argument can be provided for optimization.

For example: `assert = MPI_MODE_NOPUT`

```
MPI_Win_fence(int assert, MPI_Win win)
```

Code Example

```
MPI.Win_fence(win)
if process_id !== size - 1
    MPI.Put!(u[1], win; rank=1, disp=0)
else
    MPI.Put!(u[N], win; rank=1, disp=0)
end
MPI.Win_fence(win)
```

Window Synchronization Routines

MPI_Win_lock/unlock: Passive target synchronization

- Target rank does not need to make corresponding MPI call for synchronization.

Window Synchronization Routines

MPI_Win_lock/unlock: Passive target synchronization

- Target rank does not need to make corresponding MPI call for synchronization.
- Two types of lock: **Shared** and **Exclusive**

Window Synchronization Routines

MPI_Win_lock/unlock: Passive target synchronization

- Target rank does not need to make corresponding MPI call for synchronization.
- Two types of lock: **Shared** and **Exclusive**
- **Shared:** Other process with same lock type can access concurrently.

Window Synchronization Routines

MPI_Win_lock/unlock: Passive target synchronization

- Target rank does not need to make corresponding MPI call for synchronization.
- Two types of lock: **Shared** and **Exclusive**
- **Shared:** Other process with same lock type can access concurrently.
- **Exclusive:** No other process can access concurrently.

Window Synchronization Routines

MPI_Win_lock/unlock: Passive target synchronization

- Target rank does not need to make corresponding MPI call for synchronization.
- Two types of lock: **Shared** and **Exclusive**
- **Shared:** Other process with same lock type can access concurrently.
- **Exclusive:** No other process can access concurrently.
- `MPI_Win_unlock` does the needed synchronization.

```
MPI_Win_lock(int locktype, int rank, int assert, MPI_Win win)
...
// RMA operations
...
MPI_Win_unlock(int rank, MPI_Win win)
```

Listing: Usage of lock/unlock

Example Implementation

```
function get_ghost_values!(param, u, win)
    if rank !== size - 1
        buf1 = fill(0.0, 1)
        MPI.Win_lock(win; rank=next, type=MPI.LOCK_SHARED)
        MPI.Put!(u[N+1], win; rank=next, disp=0)
        MPI.Get!(buf1, win; rank=next, disp=1)
        MPI.Win_unlock(win, rank=next)
        u[N+2] = buf1[1]
    else
        buf2 = fill(0.0, 1)
        MPI.Win_lock(win; rank=next, type=MPI.LOCK_SHARED)
        MPI.Put!(u[N+1], win; rank=next, disp=0)
        MPI.Get!(buf2, win; rank=next, disp=2)
        MPI.Win_unlock(win; rank=next)
        u[N+2] = buf2[1]
    end
end
```

FAQs

What is the difference between RMA and shared memory?

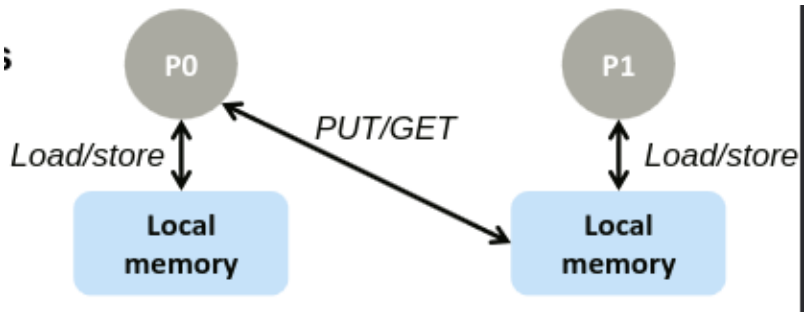


Figure: RMA

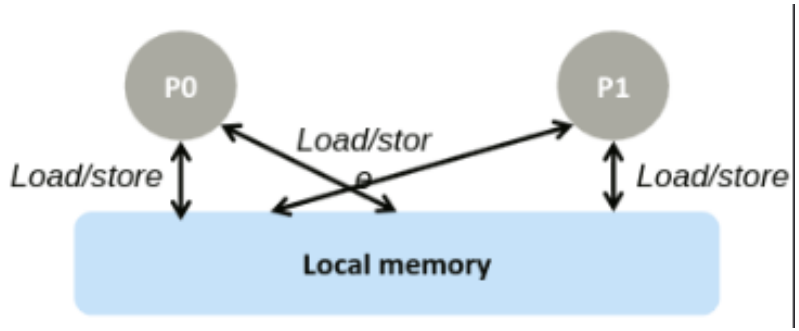


Figure: Shared memory

Does RMA work with the CPUs on different node?
Yes, it does. With the help of modern hardware such as **Infiniband** which follows (Remote direct memory access) RDMA protocols.

References

[1] Babbar, A., Kenettinkara, S., & Chandrashekar, P. (2022). Lax-Wendroff flux reconstruction method for hyperbolic conservation laws. Journal of Computational Physics, 467, 111423.

[2] Schlottke-Lakemper, M., Gassner, G. J., Ranocha, H., Winters, A. R., & Chan, J. (2024). Trixi.jl (v0.8.3). Zenodo. https://doi.org/10.5281/zenodo.12683615

[3] Hoefler, T., Balaji, P., Gropp, W., & Thakur, R. (2016, November). *Advanced MPI Programming* [Tutorial] https://web.cels.anl.gov/thakur/sc16-mpi-tutorial/slides.pdf