MPI.jl: Julia bindings for the Message Passing Interface

Simon Byrne[1], Lucas C. Wilcox[2], and Valentin Churavy[3]

[1]California Institute of Technology
[2]Naval Postgraduate School
[3]Massachusetts Institute of Technology

ABSTRACT

MPI.jl is a Julia package for using the Message Passing Interface (MPI), a standardized and widely-supported communication interface for distributed computing, with multiple open source and proprietary implementions. It roughly follows the C MPI interface, with some additional conveniences afforded by the Julia language such as automatic handling of buffer lengths and datatypes.

Keywords

Julia, MPI, distributed computing

1. Introduction

Now over 25 years old, MPI is the stalwart of high-performance computing communication, supported on everything from single machines to billion-dollar supercomputers. Despite its age, it supports several models of communication, and significant engineering effort goes into optimizing performance and supporting the latest networking hardware.
Although Julia provides its own suite of distibuted computing tools via the Distributed standard library, it is based on a controller/-worker model and is currently unable to leverage fast networking hardware such as InfiniBand, which limits its scalability to large problems. MPI.jl leverages the well-established and proven technology, including extensions such as the CUDA-aware interfaces for multi-GPU communication. It is being used by multiple Julia projects, including the CliMA Earth system modelling project [4].

2. Simple example and running MPI programs

Most MPI programs utilise a single-program, multiple-data (SPMD) model where multiple processes all run the same program and communicate via messages, with their data determined by the process rank (a 0-based ordering of the processes).
An example of this is is a simple "round-robin" communication pattern in which each process sends a message containing its rank to its next neighbor using non-blocking point-to-point operations:

```
# sendrecv.jl
# initialize and set global variables
using MPI
MPI.Init()

comm = MPI.COMM_WORLD
rank = MPI.Comm_rank(comm)
N = MPI.Comm_size(comm)

# non-blocking receive from previous rank
recv_buf = Array{Float64}(undef, 2)
recv_req = MPI.Irecv!(recv_buf, mod(rank-1, N), 0,
                      comm)

# non-blocking send to next rank
send_buf = Float64[rank, rank]
send_req = MPI.Isend(send_buf, mod(rank+1, N), 0,
                     comm)

# block until communication is completed
MPI.Waitall!([recv_req, send_req])
print("$rank: Received $recv_buf\n")
```

This can be run using the MPI launcher (typically called `mpiexec`.

```
$ mpiexec -n 3 julia sendrecv.jl
0: Received [2.0, 2.0]
2: Received [1.0, 1.0]
1: Received [0.0, 0.0]
```

3. Implementation details and challenges

Although MPI.jl mirrors the C MPI interface quite closely, it does take advantage of several features of the Julia language to improve usability. The C and Fortran MPI interfaces require that users manually check the error code returned by each function; MPI.jl is able to use Julia's exception handling machinery to automatically check error codes and print readable error messages. This allows functions to return their results via return values instead of via additional functions arguments. For example, non-blocking operations return `Request` objects; blocking receive operations return their output buffers.

3.1 Allocation and serialization

For communication operations which receive data, MPI.jl typically defines two separate functions:

—one function in which the output buffer is supplied by the user: as it mutates this value, it adopts the Julia convention of suffixing with `!` (e.g. `MPI.Recv!`, `MPI.Reduce!`).

—one function which allocates the buffer for the output (`MPI.Recv`, `MPI.Reduce`).

Additionally, we adopt the convention from the mpi4py Python MPI bindings [2] of using lowercase names for functions which are able to handle arbirary objects. These are typically slower as they rely on serialization and are not type-stable, but can be convenient as they don't require that the object type or size be known by

the receiver. Currently only a small number of these functions are provided.

3.2 Buffers, datatypes and operators

In C and Fortran, MPI communication functions require three arguments (address, count and element datatype) to specify their input and/or output buffers e.g. the `MPI_Send` signature in C has six arguments:

```
int MPI_Send(const void* buf, int count,
      MPI_Datatype datatype, int dest, int tag,
      MPI_Comm comm)
```

In Julia, these can all be determined from an `Array` object, so the corresponding function in MPI.jl only requires 4 arguments:

```
MPI.Send(buf, dest::Integer, tag::Integer,
      comm::MPI.Comm)
```

An intermediate `Buffer` type is defined that captures the necessary properties, and allows defining MPI communication operations for other Julia objects without requiring that additional methods for every communication function. For example, to support the CUDA-aware MPI interface across all MPI functions, only a single additional `Buffer(arr::CUDA.CuArray)` method was required.
If the element type of the buffer is not one of the predefined MPI datatypes, then MPI.jl will automatically build and commit the corresponding MPI user-defined type.
Similarly, for collective reduction operations (`MPI.Reduce`, `MPI.Scan`, etc.), MPI.jl will convert Julia functions to MPI operator objects, either mapping to predefined operators (e.g. `+` to `MPI.SUM`), or wrapping functions to form custom operators.
Both of these are illustrated in the pooled variance example in section 4.3.

3.3 Application binary interface

The MPI standard specifies C and Fortran appilication programming interfaces (API), but not an application binary interface (ABI). Consequently, datatypes and enum values vary between different implementations, and require parsing C headers to extract their precise values. We use two approaches to work around this problem:

—Attempt to identify the MPI implementation by querying `MPI_Get_library_version`, and use predefined constants and types if known to be compatible with MPICH, Open MPI or Microsoft MPI.

—Otherwise, at build time it compiles a small C program that outputs the type sizes and constants. One complication is that the opaque C handles might only be defined at link time: in this case, we convert to the Fortran handle values (which are required to be integers), and convert back to C handles when calling `MPI.Init()`. A similar approach is used by the MPI bindings for Rust [3].

3.4 Binary support

Similar to many Julia packages, MPI.jl uses BinaryBuilder and the Artifacts system to automatically install an MPI implementation when the package is installed (currently Microsoft MPI on Windows, MPICH on other platforms), which simplifies the installation procedure for users on single machines.
On high-performance computing systems one would typically want to use system or other externally-provided binaries. To aid this, MPI.jl provides additional hooks to enable switching this at build time via environment variables, and a warning is show if a user appears to be using the default MPI binary on a HPC system. Challenges remain on how to make it easier to switch implementations (when multiple are present), or how to deal with binaries which depend on MPI.

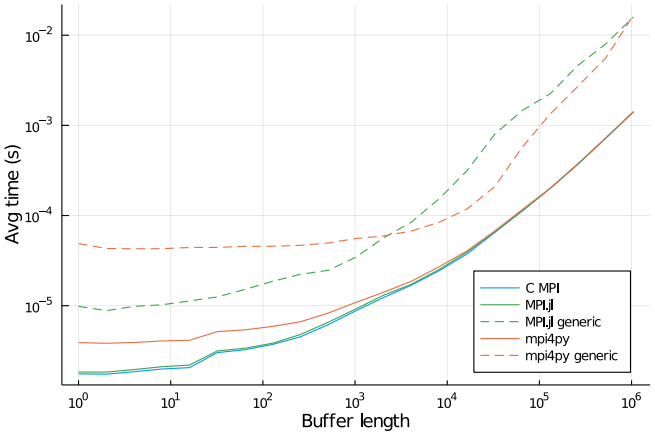

Fig. 1. MPI ping pong benchmark in C, Julia (MPI.jl) and Python (mpi4py) using arrays of 64-bit floating-point numbers. Benchmarks were performed using Open MPI 4.0.4, using two processes on different nodes connected by EDR InfiniBand.

4. Examples

4.1 Ping pong benchmark

The "ping pong" benchmark consists of two MPI processes which alternate sending messages between each other, and is a useful measure of how function call overhead affects communication latency. A simple Julia implementation is:

```
function pingpong(T, bufsize, iters)
    buffer = zeros(T, bufsize)
    comm = MPI.COMM_WORLD
    rank = MPI.Comm_rank(comm)
    tag = 0

    MPI.Barrier(MPI.COMM_WORLD)
    tic = MPI.Wtime()
    for i = 1:iters
        if rank == 0
            MPI.Send(buffer, 1, tag, comm)
            MPI.Recv!(buffer, 1, tag, comm)
        else
            MPI.Recv!(buffer, 0, tag, comm)
            MPI.Send(buffer, 0, tag, comm)
        end
    end
    toc = MPI.Wtime()

    avgtime = (toc-tic)/iters
    return avgtime
end
```

Figure 1 compares the ping pong benchmark implemented in C, Julia using MPI.jl, and Python using mpi4py. The MPI.jl benchmark exhibits similar performance to C, whereas mpi4py is notable slower for smaller message sizes, likely due to the intepreter overhead of Python.
In addition, for MPI.jl and mpi4py we also compare the lowercase "generic" `MPI.send`and `MPI.recv` functions, which are able to handle arbitrary objects. Here MPI.jl is still faster than mpi4py for small messages, but slower for medium-sized messages. We suspect this is due to garbage collection pauses in the Julia runtime.

4.2 Minimum-spanning tree broadcast

Julia syntax is close to pseudo-code found in the literature to describe parallel algorithms. For example, consider the minimum-spanning tree broadcast algorithm in Figure 3a of [1]. A Julia implementation is given as:

```
function MSTBcast(x, root, left, right, comm)
    me = MPI.Comm_rank(comm)
    tag = 999

    if left == right
        return x
    end
    mid = div((left + right),  2)
    dest = root <= mid ? right : left

    if me == root
        MPI.send(x, dest, tag, comm)
    end
    if me == dest
        (x, _) = MPI.recv(root, tag, comm)
    end

    if me <= mid && root <= mid
        MSTBcast(x, root, left, mid, comm)
    elseif me <= mid && root > mid
        MSTBcast(x, dest, left, mid, comm)
    elseif me > mid && root <= mid
        MSTBcast(x, dest, mid + 1, right, comm)
    elseif me > mid && root > mid
        MSTBcast(x, root, mid + 1, right, comm)
    end
end
```

This is nearly identical to the pseudo-code and can be called for all of the datatypes supported by `MPI.send` and `MPI.recv`, for example arrays, functions, and dictionaries.

4.3 Pooled variance using custom datatypes and operators

The following example uses both custom MPI datatypes and custom reduction operators to compute the pooled variance of a distributed dataset in a numerically stable way, using a single communication operation:

```
# Custom struct containing the summary statistics
# (mean, variance, count)
struct SummaryStat
    mean::Float64
    var::Float64
    n::Float64
end

function SummaryStat(X::AbstractArray)
    m = mean(X)
    v = varm(X,m, corrected=false)
    n = length(X)
    SummaryStat(m,v,n)
end

# Custom reduction operator, computing pooled mean,
# variance and length
function pool(S1::SummaryStat, S2::SummaryStat)
    n = S1.n + S2.n
    m = (S1.mean*S1.n + S2.mean*S2.n) / n
    v = (S1.n * (S1.var + S1.mean * (S1.mean-m)) +
         S2.n * (S2.var + S2.mean *
(S2.mean-m)))/n
    SummaryStat(m,v,n)
end

# Perform a scalar reduction to `root`
summ = MPI.Reduce(SummaryStat(X), pool, root, comm)
```

5. Acknowledgements

We thank the many contributors to MPI.jl over the years: Erik Schnetter, Jared Crean, Jake Bolewski, Davide Lasagna, Katharine Hyatt, Jeremy Kozdon, Andreas Noack, Bart Janssens, Amit Murthy, Steven G. Johnson, David Anthoff, Thomas Bolemann, Joey Huchette, Seyoon Ko, Juan Ignacio Polanco, Tristan Konolige, Samuel Omlin, Mosè Giordano, Filippo Vicentini, Keno Fischer, Maurizio Tomasi, Yuichi Motoyama, Tom Abel, Jane Herriman, Ernesto Vargas, Elliot Saba, Rohan McLure, Randy Lai, Mike Nolta, Josh Milthorpe, Michel Schanen, Kiran Pamnany, Joaquim Dias Garcia, Jonathan Goldfarb, Chris Hill, Balazs Nemeth, Alberto F. Martin, Ali Ramadhan, Viral Shah, Sacha Verweij, Kristoffer Carlsson, Joel Mason and Yao Lu.
This research was made possible by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program, Mountain Philanthropies, the Paul G. Allen Family Foundation, and the National Science Foundation (NSF award AGS-1835860).

6. References

[1] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[2] Lisandro D Dalcin, Rodrigo R Paz, Pablo A Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

[3] rsmpi developers. rsmpi: MPI bindings for Rust, 2020.

[4] Tapio Schneider, Shiwei Lan, Andrew Stuart, and João Teixeira. Earth system modeling 2.0: A blueprint for models that learn from observations and targeted high-resolution simulations. *Geophysical Research Letters*, 44(24):12,396–12,417, 2017.