The Journal of Open Source Software

mpi4jax: Zero-copy MPI communication of JAX arrays

Dion Häfner[*1] **and Filippo Vicentini**[†2]

1 Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark **2** Institute of Physics, École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland

DOI: 10.21105/joss.03419

Software

- Review
- Repository
- Archive

Editor: Kelly Rowland

Reviewers:

- @1313e
- @Himscipy

Submitted: 10 June 2021
Published: 30 August 2021

Summary

The tensor framework JAX (Bradbury et al., 2018) combines expressivity and performance while retaining an accessible pure Python interface. Expressivity is achieved by treating functions as first-class objects, while efficiency is obtained by compiling to machine code just-ahead-of-time.

However, machine learning and (high-performance) scientific computing are often conducted on different hardware stacks: Machine learning is typically done on few highly parallel units (GPUs or TPUs) connected to a single host CPU, while scientific models tend to run on clusters of dozens to thousands of CPUs. Unfortunately, support from JAX and the underlying compiler XLA is more mature in the former case. Notably, there is so far no built-in solution to communicate data between different nodes that is as sophisticated as the widely used MPI (Message Passing Interface) libraries (Forum, 1994).

We attempt to fill this gap and introduce `mpi4jax`, a Python library bringing first-class support for the most important MPI operations to JAX. We achieve this by defining a set of primitive functions matching MPI's operations, instructing JAX how to transform them and providing a native implementation to execute them. This has the result that users can communicate arbitrary JAX data without performance or usability penalties. In particular, `mpi4jax` is able to communicate data without copying from CPU and GPU memory (if built against a CUDA-aware MPI library) between one or multiple hosts (e.g. via an Infiniband network on a cluster).

This also means that existing applications using NumPy and `mpi4py` can be ported seamlessly to the JAX ecosystem for potentially significant performance gains.

Statement of Need

For decades, high-performance computing has been done primarily in low-level programming languages like Fortran or C. But the ubiquity of Python is starting to spill into this domain as well, thanks to its strong library ecosystem and wide adoption throughout the sciences.

With a combination of NumPy (Harris et al., 2020) and `mpi4py` (Dalcín et al., 2005), Python users can already build massively parallel applications without delving into low-level programming languages, which is often advantageous when human time is more valuable than computing time. However, such high-level frameworks are not always able to achieve peak performance, especially in more *niche* workloads.

Google's JAX library leverages the XLA compiler and supports just-in-time compilation (JIT) of (a subset of) Python code to XLA primitives. The result is highly competitive performance on

[*] Contributed equally, order determined by coin flip.
[†] Contributed equally, order determined by coin flip.

both CPU and GPU (Häfner, 2020). This approach, conceptually similar to Julia's deferred compilation model (Bezanson et al., 2017), achieves low-level performance in a high-level language. With a strong performance baseline on single devices, the only thing missing is easy scalability to massively parallel hardware stacks, which we supply here.

Two real-world use cases for `mpi4jax` are the ocean model Veros (Häfner et al., 2018) and the machine learning toolkit for many-body quantum systems NetKet (Carleo et al., 2019):

- In the case of Veros, MPI primitives are needed to communicate overlapping grid cells between processes. Communication primitives are buried deep into the physical subroutines. Therefore, refactoring the codebase to leave `jax.jit` every time data needs to be communicated would severely break the control flow of the model and incur a hefty performance loss (in addition to the cost of copying data from and to JAX). Through `mpi4jax`, it is possible to apply the JIT compiler to whole subroutines to avoid this entirely.
- In the case of NetKet, a high efficiency algorithm for natural gradient optimization requires finding the solution of a large linear system $Ax = y$. The matrix A is determined by running automatic differentiation on a neural network model whose inputs might be distributed across several computing nodes and GPUs. Therefore, the need to differentiate through distributed reduction operations inside of a linear solver arises.

Implementation

`mpi4jax` combines JAX's custom call mechanism with `mpi4py.libmpi` (which exposes MPI C primitives as Cython callables).

The implementation of a primitive in `mpi4jax` consists of two parts:

1. A Python module, registering a new primitive with JAX. JAX primitives consist of an *abstract evaluation* rule and several *translation rules*. Abstract evaluation rules are used by the compiler to infer the output shapes and data types without running the actual computation, while *translation rules* determine the specific computational kernel and prepare the input buffers.

   In particular, we need to ensure that all numerical input data is of the expected type (e.g., by converting Python integers to the C type `uintptr_t`) before passing it on to XLA. A different translation rule is necessary for every type of backend, such as CPUs, GPUs and TPUs.

   On specific primitives we also define a transposition and JVP (Jacobian-vector product) rule to support forward and reverse mode automatic differentiation.

2. A Cython (Behnel et al., 2011) function that casts raw input arguments passed by XLA to their true C type, so they can be passed on to MPI. On CPU, arguments are given in the form of arrays of void pointers, `void**`, so we use static casts for conversion. On GPU, input data is given as a raw char array, `char*`, which we deserialize to a custom Cython `struct` whose fields represent the input data.

   On GPU, our Cython bridge also supports copying the data from device to host and back before and after calling MPI (by linking `mpi4jax` to the CUDA runtime library). This way, we support the communication of GPU data via main memory if the underlying MPI library is not built with CUDA support (at a minor performance penalty).

This is sufficient for our primitives to be callable from compiled JAX code. However, there is one additional complication: we need to take special care to ensure that MPI statements are not re-ordered. Consider the following example:

```python
@jax.jit
def exchange_data(arr):
    if rank == 0:
        # rank 0 sends, then receives
        mpi4jax.send(arr, dest=1)
        newarr = mpi4jax.recv(arr, source=1)
    else:
        # rank 1 receives, then sends
        newarr = mpi4jax.recv(arr, source=0)
        mpi4jax.send(arr, dest=0)
    return newarr
```

As JAX and XLA operate on the assumption that all primitives are pure functions without side effects, the compiler is in principle free to re-order the `send` and `recv` statements above. This would typically lead to a deadlock or crash, as both processes might wait for each others' input at the same time.

The solution to this in JAX is a token mechanism that involves threading a dummy token value as input and output through each primitive. This introduces a fake data dependency between subsequent calls using the token, which prevents XLA from re-ordering them relative to each other.

The example above, using proper token management, reads:

```python
@jax.jit
def exchange_data(arr):
    if rank == 0:
        token = mpi4jax.send(arr, dest=1)
        newarr, token = mpi4jax.recv(arr, source=1, token=token)
    else:
        newarr, token = mpi4jax.recv(arr, source=0)
        token = mpi4jax.send(arr, dest=0, token=token)
    return newarr
```

As a result, we are successfully able to execute MPI primitives just as if they were JAX primitives. This incurs minimal overhead, as no data is copied between JAX and MPI. All `mpi4jax` primitives operate directly on device memory addresses (this is what we refer to as *zero-copy*).

We can quantify this overhead by comparing the runtime of a JAX function with and without using an `mpi4jax` call. We also exclude the time spent inside the MPI library by using `mpi4jax`'s debug logging mechanism (benchmark script available online). This reveals an overhead of about 1µs, which is negligible in virtually any real-world application.

As of yet, `mpi4jax` supports the MPI operations `allgather`, `allreduce`, `alltoall`, `bcast`, `gather`, `recv`, `reduce`, `scan`, `scatter`, `send`, and `sendrecv` (Forum, 1994). Most still unsupported operations such as `gatherv` could be implemented with little additional work if needed by an application.

Example & Benchmark: Non-linear Shallow Water Solver

As a demo application, and to use as a benchmark, we have ported a non-linear shallow water solver to JAX and parallelized it with `mpi4jax` (Figure 1).

In this simple example, all workers operate on a rectangular region of the 2D domain (split evenly into `[nproc / 2, 2]` chunks). All communication is handled via halo exchanges, i.e.,

each chunk contains 1 extra cell around the perimeter of the domain and exchanges it with its neighbors when needed. As this is a simple problem on a regular grid, the mesh partitioning is done manually.

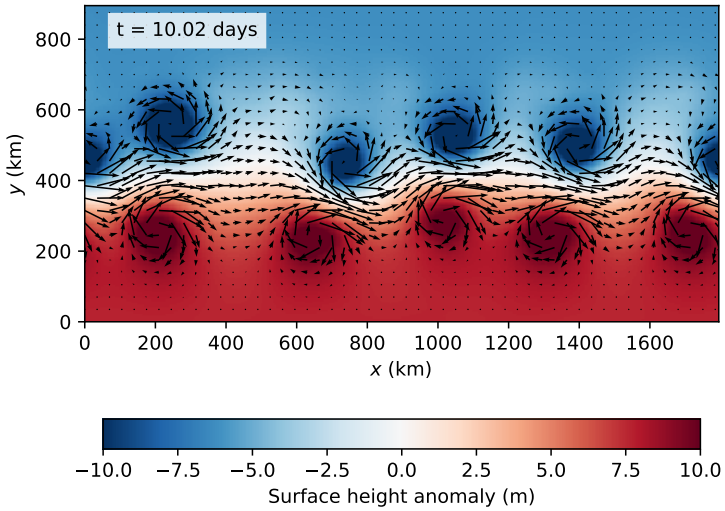

Figure 1: Output snapshot of the non-linear shallow water model. Shading indicates surface height, quivers show the current's velocity field.

The full example is available in the `mpi4jax` repository. It defines a function `enforce_boundaries` where we use `mpi4jax` to handle halo exchanges between all MPI processes. The core of it reads similar to this (plus some special cases to take care of processes at the edges of the domain):

```
@jax.jit
def enforce_boundaries(arr, grid, token):
    # start sending west / receiving east, go clockwise
    send_order = ("west", "north", "east", "south")
    recv_order = ("east", "south", "west", "north")

    # loop over neighbors
    for send_dir, recv_dir in zip(send_order, recv_order):
        # determine neighboring processes
        send_proc = proc_neighbors[send_dir]
        recv_proc = proc_neighbors[recv_dir]

        # determine data to send
        send_idx = overlap_slices_send[send_dir]
        send_arr = arr[send_idx]

        # determine where to place received data
        recv_idx = overlap_slices_recv[recv_dir]
        recv_arr = jnp.empty_like(arr[recv_idx])

        # execute send-receive operation
        recv_arr, token = mpi4jax.sendrecv(
            send_arr,
            recv_arr,
```

```
            source=recv_proc,
            dest=send_proc,
            comm=mpi_comm,
            token=token,
        )

        # update array with received data
        arr = arr.at[recv_idx].set(recv_arr)

    return arr
```

Then, it can be used in the physical simulation like this:

```
@jax.jit
def shallow_water_step(state):
    token = jax.lax.create_token()
    # ...
    fe = fe.at[1:-1, 1:-1].set(
      0.5 * (hc[1:-1, 1:-1] + hc[1:-1, 2:]) * u[1:-1, 1:-1]
    )
    fn = fn.at[1:-1, 1:-1].set(
      0.5 * (hc[1:-1, 1:-1] + hc[2:, 1:-1]) * v[1:-1, 1:-1]
    )
    fe, token = enforce_boundaries(fe, "u", token)
    fn, token = enforce_boundaries(fn, "v", token)
    # ...
```

Note how we are able to mix boundary communication with numerical computation in the same `jax.jit` block. This would not be possible without `mpi4jax`.

To verify the performance scaling of the solver with additional processes, we performed a rudimentary benchmark by running a bigger version of this example (shape 3600×1800) on several combinations of platform (CPU / GPU) and number of processes.

Platform	# processes	Elem. per worker	Time (s)	Rel. speedup
CPU	1 (NumPy)	6.5M	770	1
CPU	1	6.5M	112	6.9
	2	3.2M	90	8.6
	4	1.6M	39	19
	6	0.8M	29	27
	8	0.4M	21	37
	16	0.2M	16	48
GPU	1	6.5M	6.3	122
	2	3.2M	3.9	197

(The test hardware consists of 2x Intel Xeon E5-2650 v4 CPUs and 2x NVIDIA Tesla P100 GPUs.)

As we can see, switching from NumPy to JAX already yields a substantial speedup, which we can then amplify by scaling to additional CPUs or GPUs.

More in-depth benchmarks on larger architectures are available for the Veros ocean model,

which uses `mpi4jax` to parallelize its JAX backend.

Outlook

In this paper, we introduced `mpi4jax`, which allows zero-copy communication of JAX-owned data. `mpi4jax` provides an implementation of the most important MPI operations in a way that is usable from JAX compiled code.

However, JAX is much more than just a JIT compiler. It is also a full-fledged differentiable programming framework by providing tools for automatic differentiation (e.g. via `jax.grad`, `jax.vjp`, and `jax.jvp`). Differentiable programming is a promising new paradigm to combine advances in machine learning and physical modelling (Degrave et al., 2018; Li et al., 2021), and being able to freely distribute those models among different nodes will allow for even more powerful applications.

Combining automatic-differentiation with the multi-process nature of `MPI` workloads is not trivial, and right now `mpi4jax` allows to differentiate only few communication primitives. An interesting future development for the library will be to properly support the whole set of operations, enabling fully differentiable, distributed physical simulations without additional user code.

Acknowledgements

We thank all JAX developers, in particular Matthew Johnson and Peter Hawkins, for their outstanding support on the many issues we opened. We also thank Himanshu Sharma and Ellert van der Velden for their insightful review comments.

DH acknowledges funding from the Danish Hydrocarbon Research and Technology Centre (DHRTC).

FV acknowledges support from G. Carleo and funding from the Simons Foundation and Ecole Polytechnique Federale de Lausanne.

References

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., & Smith, K. (2011). Cython: The best of both worlds. *Computing in Science and Engg.*, *13*(2), 31–39. https://doi.org/10.1109/MCSE.2010.118

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.2.5) [Computer software]. http://github.com/google/jax

Carleo, G., Choo, K., Hofmann, D., Smith, J. E. T., Westerhout, T., Alet, F., Davis, E. J., Efthymiou, S., Glasser, I., Lin, S.-H., Mauri, M., Mazzola, G., Mendl, C. B., Nieuwenburg, E. van, O'Reilly, O., Théveniaut, H., Torlai, G., Vicentini, F., & Wietek, A. (2019). NetKet: A machine learning toolkit for many-body quantum systems. *SoftwareX*, *10*, 100311. https://doi.org/10.1016/j.softx.2019.100311

Dalcín, L., Paz, R., & Storti, M. (2005). MPI for python. *Journal of Parallel and Distributed Computing*, *65*(9), 1108–1115. https://doi.org/https://doi.org/10.1016/j.jpdc.2005.03.010

Degrave, J., Hermans, M., Dambre, J., & wyffels, F. (2018). *A differentiable physics engine for deep learning in robotics*. http://arxiv.org/abs/1611.01652

Forum, M. P. (1994). *MPI: A message-passing interface standard*. University of Tennessee.

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., R'ıo, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Häfner, D. (2020). PyHPC benchmarks. In *GitHub repository*. GitHub. https://github.com/dionhaefner/pyhpc-benchmarks

Häfner, D., Jacobsen, R. L., Eden, C., Kristensen, M. R. B., Jochum, M., Nuterman, R., & Vinter, B. (2018). Veros v0.1 – a fast and versatile ocean simulator in pure Python. *Geoscientific Model Development*, *11*(8), 3299–3312. https://doi.org/10.5194/gmd-11-3299-2018

Li, L., Hoyer, S., Pederson, R., Sun, R., Cubuk, E. D., Riley, P., & Burke, K. (2021). Kohn-sham equations as regularizer: Building prior knowledge into machine-learned physics. *Phys. Rev. Lett.*, *126*, 036401. https://doi.org/10.1103/PhysRevLett.126.036401