



Julia for High Performance Computing

Carsten Bauer

Paderborn Center for Parallel Computing (PC2)
National High-Performance Computing Alliance (NHR)

Course Objectives

- Learn about **Julia** with a focus on HPC
- Learn about **HPC** concepts in and with Julia
- **Have some fun** 😊

Schedule

	Tuesday	Wednesday	Thursday	Friday
	Foundations	"Serial"	Parallel	GPU
09:00 - 11:00 (2h)	Julia for HPC Fundamentals	Performance Optimisation	Multithreading	Experience Report GPU
11:00 - 11:15 (15m)	Break	Break	Break	Break
11:15 - 12:45 (1.5h)	Compilation & Fast User Code	Hands-On	Hands-On	Hands-On
12:45 - 14:15 (1.5h)	Lunch	Lunch	Lunch	Lunch
14:15 - 15:45 (1.5h)	Hands-On	SIMD & Profiling	Distributed Computing	Buffer / Hands-On Outro + Q&A
15:45 - 16:00 (15m)	Break	Break	Break	
16:00 - 17:00 (1h)	Workflow	Hands-On	Hands-On	

Course Material

github.com/carstenbauer/JuliaHLRS23

Preparation

- Make sure that you have the latest state

git pull

- Make sure everything is precompiled (will take a few minutes if not)

julia --project precompile.jl

Quick Live Survey

The Julia Programming Language

- **Domain science**

- “Python-like” syntax
- Dynamic & interactive
 - powerful REPL
 - Jupyter notebooks ([Julia](#) + [Python](#) + [R](#))
- Powerful linear algebra + ecosystem
- Package manager, environments

- **High-performance computing**

- Compiled via LLVM (just ahead of time)
- Parallel computing
- Hardware accelerator programming
 - High-level abstractions + low-level kernels
- Interactive HPC

```
julia> function sumup()
           x = 0
           for i in 1:100
               x += i
           end
           return x
       end
sumup (generic function with 1 method)
```

```
julia> @code_llvm debuginfo=:none sumup()
define i64 @julia_sumup_177() #0 {
top:
    ret i64 5050
}
```

Where I Come From

- PhD: Condensed matter physics / quantum field theory
 - Large-scale Quantum Monte Carlo simulations
 - $O(15$ million CPU-core hours)
 - $O(100$ TB of data)

Where I Come From

- PhD: Condensed matter physics / quantum field theory
 - Large-scale Quantum Monte Carlo simulations
 - $O(15$ million CPU-core hours)
 - $O(100$ TB of data)



Where I Come From

- PhD: Condensed matter physics / quantum field theory
 - Large-scale Quantum Monte Carlo simulations
 - $O(15$ million CPU-core hours)
 - $O(100$ TB of data)



Domain Science



HPC

Where I Come From

- PhD: Condensed Matter Physics
 - Large-scale QM calculations
 - O(15 million CPU hours)
 - O(100 TB of data)

nature
"Julia: come for the syntax, stay for the speed"
[DOI: 10.1038/d41586-019-02310-3](https://doi.org/10.1038/d41586-019-02310-3)



Domain Science



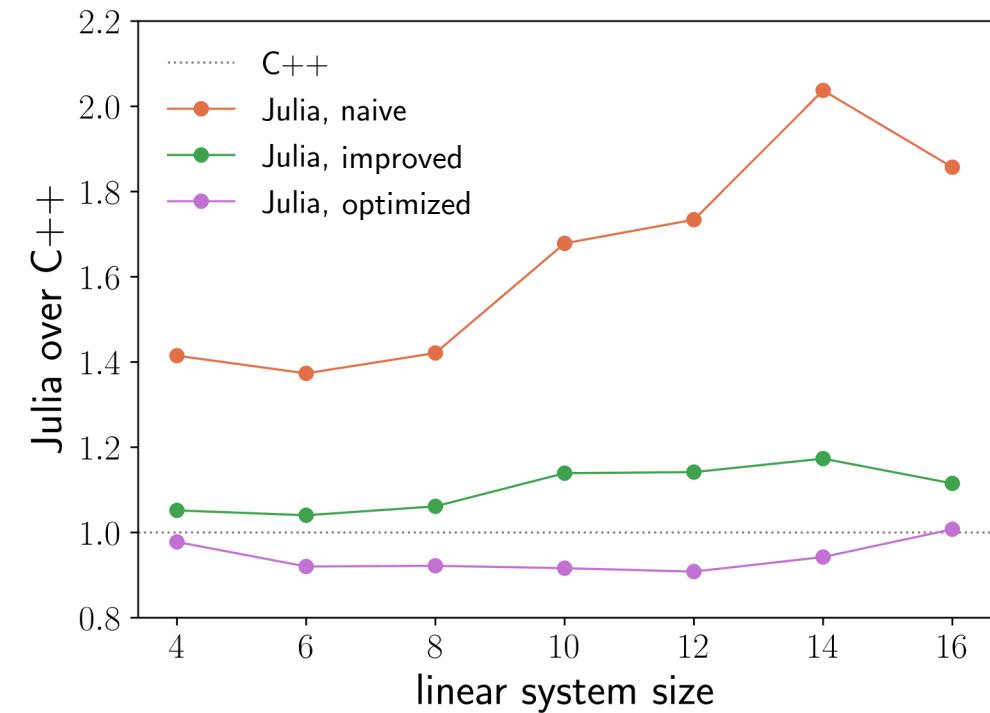
julia



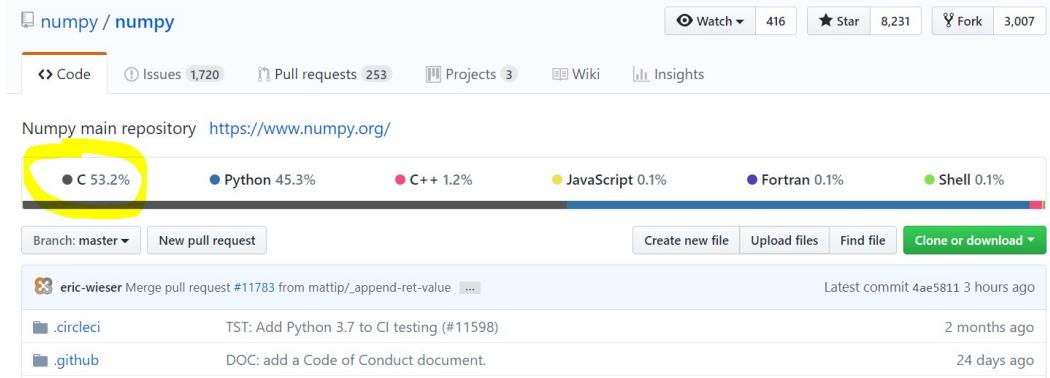
HPC

Where I Come From: Large-Scale Quantum Monte Carlo

- Prototypical Julia implementation
 - ~3x faster than Python (not shown)
 - Within a factor of ~2x compared to C++
- Could **gradually** improve performance
 - No disruptive language change necessary!
 - A lot of fun!



Social Barrier



Developer

User



Social Barrier



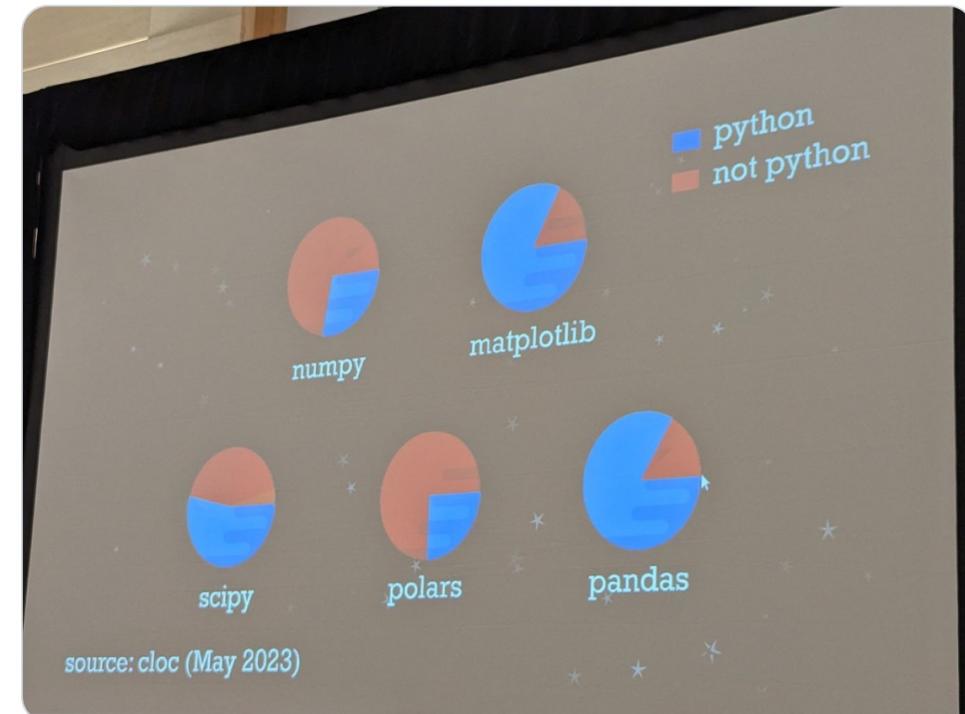
Dillon is @ #SciPy2023
@dillonniederhut

...

One of the weird things about working in the [#Python](#) community is that our core technologies are mostly not written in Python.

If someone wants to contribute code, now they have two problems.

- Michael Droettboom at [#SciPy2023](#)



4:48 PM · Jul 12, 2023 · 100.2K Views

Python

```
def vander(x, N=None, increasing=False):
    x = asarray(x)
    if x.ndim != 1:
        raise ValueError("x must be a one-dimensional array or sequence.")
    if N is None:
        N = len(x)

    v = empty((len(x), N), dtype=promote_types(x.dtype, int))
    tmp = v[:, ::-1] if not increasing else v

    if N > 0:
        tmp[:, 0] = 1
    if N > 1:
        tmp[:, 1:] = x[:, None]
        multiply.accumulate(tmp[:, 1:], out=tmp[:, 1:], axis=1)

    return v

@TYPE@_kind@(char **args, npy_intp *dimensions, npy_intp *steps, void *NPY_UNUSED(func))
{
    if (IS_BINARY_REDUCE) {
        #if @TYPE@
            @type@ * iop1 = (@type@ *)args[0];
            npy_intp n = dimensions[0];

            *iop1 @OP@= pairwise_sum_@TYPE@(args[1], n, steps[1]);
        #else
            BINARY_REDUCE_LOOP(@type@) {
                io1 @OP@= *(@type@ *)ip2;
            }
            *((@type@ *)iop1) = io1;
        #endif
    }
    else if (!run_binary_simd_@kind@_@TYPE@(args, dimensions, steps)) {
        BINARY_LOOP {
            const @type@ in1 = *(@type@ *)ip1;
            const @type@ in2 = *(@type@ *)ip2;
            *((@type@ *)oip1) = in1 @OP@ in2;
        }
    }
}
```

calls

C code

```
PyFunc_Accumulate(PyFuncObject *ufunc, PyArrayObject *arr, PyArrayObject *out,
                   int axis, int dtype)
{
    PyArrayObject *op[2];
    PyArray_Descr *op_dtypes[2] = {NULL, NULL};
    int op_axes[2] = {op_axes_array[0], op_axes_array[1]};
    npy_uint32 op_flags[2];
    int ndim, ndim, dtype_final;
    int needs_api, need_outer_iterator;

    PyObject *iter = NULL, *iter_inner = NULL;

    /* These parameters come from extobj or from a TLS global */
    int buffersize = 0, errormask = 0;

    NPY_BEGIN_THREADS_DEF;

    NPY_IF_DBG_PRINT(("Evaluating ufunc %s.accumulate(%s", ufunc_name);

    #if 0
    printf("(Doing %s.accumulate on array with dtype : ", ufunc_name);
    PyObject_Print((PyObject *)PyArray_DESCR(arr), stdout, 0);
    printf("%s)", ufunc_name);
    #endif

    if (_get_buffersize_errormask(NULL, "accumulate", &buffersize, &errormask) < 0) {
        return NULL;
    }

    /* Take a reference to out for later returning */
    Py_XINCREF(out);

    dtype_final = dtype;
    if (get_binary_op_function(ufunc, &otype_final,
                               &innerloop, &innerloopdata) < 0) {
        PyErr_Format(PyExc_ValueError,
                    "could not find a matching type for %s.accumulate.",
                    "requested type has type code '%c'.",
                    ufunc_name, dtype ? dtype->type : '-');

        Py_XDECREF(dtype);
        goto fail;
    }

    #if NPY_UF_DBG_TRACING
    printf("Found %s.accumulate inner loop with dtype : ", ufunc_name);
    PyObject_Print((PyObject *)op_dtypes[0], stdout, 0);
    printf("\n");
    #endif

    /* Set up the op_axes for the outer loop */
    for (idim = 0; idim < ndim; ++idim) {
        op_axes_array[0][idim] = idim;
        op_axes_array[1][idim] = idim;
    }

    /* The selected inner loop */
    PyFuncGenericFunction innerloop = NULL;
    void *innerloopdata = NULL;

    const char *ufunc_name = ufunc_get_name_cstr(ufunc);

    /* If the reduction axis has size zero, either return the reduction
     * unit for UFUNC_REDUCE, or return the zero-sized output array
     * for UFUNC_ACCUMULATE.
     */
    if ((PyArray_ISALIGNED(arr)) || (out == PyArray_ISALIGNED(out))) {
        if (PyArray_EquivTypes(op_dtypes[0], PyArray_DESCR(arr))) {
            out = NULL;
        }
        else if (PyArray_ISALIGNED(op_dtypes[0], PyArray_DESCR(out))) {
            need_outer_iterator = 1;
        }
        else if (PyArray_SIZE(op[0]) == 0) {
            goto finish;
        }
        else if (PyArray_SIZE(op[1]) == 0) {
            goto finish;
        }
    }

    /* If input and output overlap in memory, use iterator to figure it out */
    else if (out == NULL || solve_my_share_memory(out, arr,
                                                   NPY_MAX_SHARE_MEMORY) < 0) {
        need_outer_iterator = 1;
    }

    /* If (out != NULL) & solve_my_share_memory(out, arr, 0) < 0 {
        need_outer_iterator = 1;
    }

    if (need_outer_iterator) {
        int ndim_iter = 0;
        npy_uint32 flags = NPY_ITER_ZERO_SIZE_OK |
                           NPY_ITER_REF_OK |
                           NPY_ITER_COPY_IF_OVERLAP;
        PyArray_Descr *op_dtypes_param = NULL;

        /* The way accumulate is set up, we can't do buffering,
         * so make a copy instead when necessary.
         */
        ndim_iter = ndim;
        flags |= NPY_ITER_MULTI_INDEX;
    }

    /* Add some more flags.
     */

    /* The accumulation outer loop is 'elementwise' over the array, so turn
     * (ndim - 1) into three for the inner loop */
    if (count_m1 > 0) {
        /* Turn the two items into three for the inner
         * loop */
        dataptr_copy[0] += stride1;
        dataptr_copy[1] += stride2;
        NPY_UF_DBG_PRINT(("Iterator loop count %d\n",
                          (int)count_m1));
    }

    innerloop(dataptr_copy, &count_m1,
              stride_copy, innerloopdata);

    NPY_END_THREADS;
}

/* Get the output */
if (out == NULL) {
    /* Turn the two items into three for the inner
     * loop */
    dataptr_copy[0] += stride1;
    dataptr_copy[1] += stride2;
    NPY_UF_DBG_PRINT(("Iterator loop count %d\n",
                      (int)count_m1));
    innerloop(dataptr_copy, &count_m1,
              stride_copy, innerloopdata);
}

else {
    PyArray_Descr *dtype = op_dtypes[0];
    Py_INCRREF(dtype);
    op[0] = out = (PykarrayObject *)Pykarray_NewFromDescr(
        Pykarray_Type, dtype,
        ndim, Pykarray_DIMS(op[1]), NULL,
        0, NULL);
}

/* Execute the loop with no iterators */
npy_intp count = PyArray_DIM(op[1], axis);
npy_intp stride0 = 0, stride1 = PyArray_STRIDE(op[0]),
        stride2 = PyArray_STRIDE(op[1], axis);
NPY_UF_DBG_PRINT(("Func: Reduce loop with no
iterators\n"));

if ((PyArray_ISDIN(op[0]) != PyArray_ISDIN(op[1])) ||
    PyArray_CompareLists(Pykarray_DIMS(op[0]),
                         Pykarray_DIMS(op[1]),
                         PyErr_SetString(PyExc_ValueError,
                                         "provided out is the wrong size "
                                         "for the reduction"));
    goto fail;
}

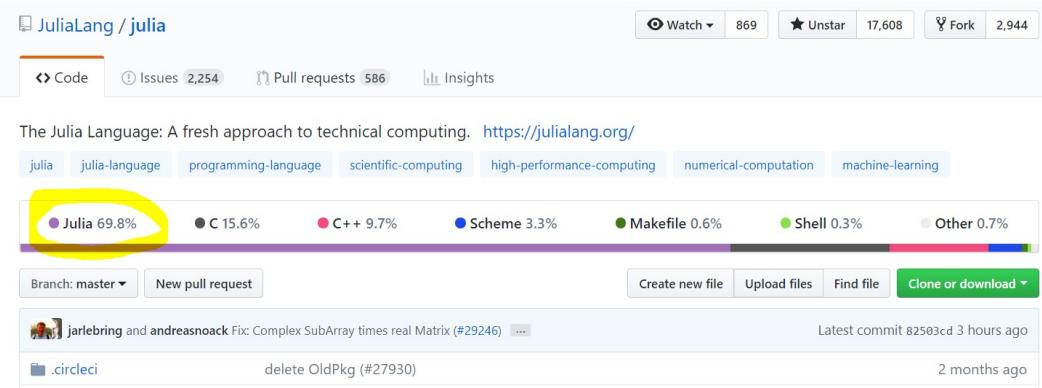
# if (iter == NULL) {
#     char *dataptr_copy[3];
#     npy_intp stride_copy[3];
#     int itemsize = op_dtypes[0]->elsize;
#     /* Execute the loop with no iterators */
#     npy_intp count = PyArray_DIM(op[1], axis);
#     npy_intp stride0 = 0, stride1 = PyArray_STRIDE(op[0]),
#             stride2 = PyArray_STRIDE(op[1], axis);
#     NPY_UF_DBG_PRINT(("Func: Reduce loop with no
# iterators\n"));
#     if ((PyArray_ISDIN(op[0]) != PyArray_ISDIN(op[1])) ||
#         PyArray_CompareLists(Pykarray_DIMS(op[0]),
#                             Pykarray_DIMS(op[1]),
#                             PyErr_SetString(PyExc_ValueError,
#                                             "provided out is the wrong size "
#                                             "for the reduction"));
#         goto fail;
#     }

#     npy_intp count_m1, stride, stride;
#     stride_copy[0] = stride0;
#     stride_copy[1] = stride;
#     stride_copy[2] = stride;
#     /* Turn the two items into three for the inner loop */
#     dataptr_copy[0] = PyArray_BYTES(op[0]);
#     dataptr_copy[1] = PyArray_BYTES(op[1]);
#     dataptr_copy[2] = PyArray_BYTES(op[2]);
#     /* Get the variables needed for the loop */
#     iternext = Pykiter_GetterNext(iter, NULL);
#     if (iternext == NULL) {
#         goto fail;
#     }
#     /* Copy the first element to start the reduction.
#      */
#     dataptr = Pykiter_GetDataPtrArray(iter);
#     /* Output (dataptr[0]) and input (dataptr[1]) may point
#      * to the same memory, e.g. np.add.accumulate(a, out).
#      */
#     if (dtype == NPY_OBJECT) {
#         if (count_m1 = PyArray_DIM(op[1], axis)-1;
```

uses

C template

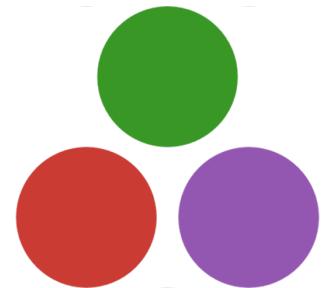
Social Barrier



User

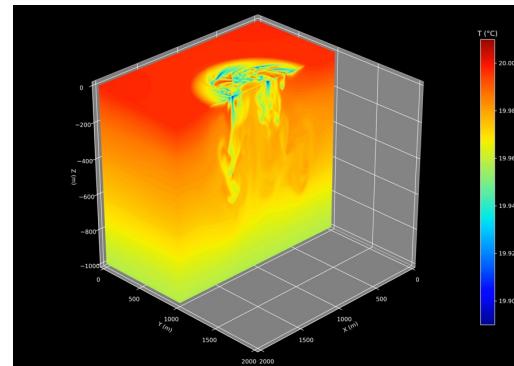


Developer



Julia HPC: International Initiatives

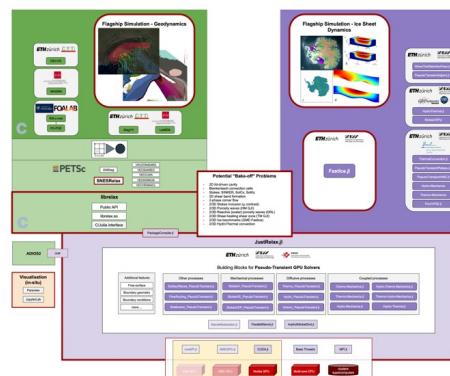
- CliMA @ Caltech
 - Climate Modeling Alliance
- CESMIX @ MIT
 - Center for the exascale simulation of materials in extreme environment
- ExaSGD @ LLNL, ORNL, ...
 - National power grid optimization
- GPU4GEO @ ETH / CSCS
 - Computational earth science
- ...



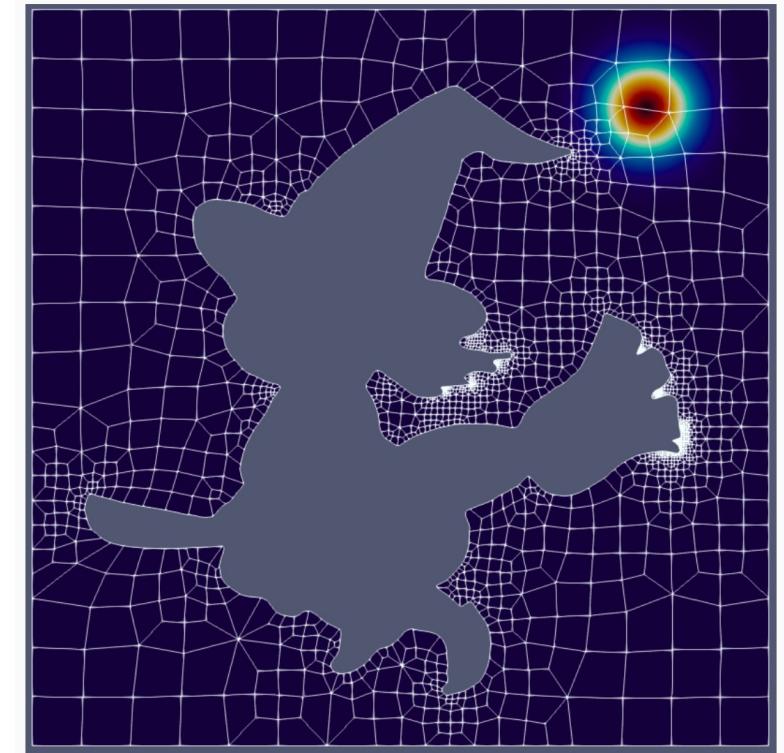
Oceananigans.jl



EXASCALE COMPUTING PROJECT



- Adaptive simulation framework for hyperbolic PDEs
 - Hybrid: MPI + Multithreading
- Authors:
 - **Michael Schlottke-Lakemper (HLRS, RWTH Aachen)**
 - Hendrik Ranocha (Uni Hamburg)
 - ...
- Competitive performance + easier development/extensibility



Credit: Andrew R. Winters

Counting Flops

```
• x = rand(10_000);  
  
• function computation(x)  
•     x .+ x  
• end;
```

Counted Flops: 10000

How?

```
• using LIKWID  
  
• metrics, events = @perfmon "FLOPS_DP" computation(x);
```

Compute from derived metrics

```
10000  
• begin  
•     flops_per_second = metrics["FLOPS_DP"][1]["DP [MFLOP/s]"] * 1e6  
•     runtime = metrics["FLOPS_DP"][1]["Runtime (RDTSC) [s]"]  
•     flops = round(Int, flops_per_second * runtime)  
• end
```

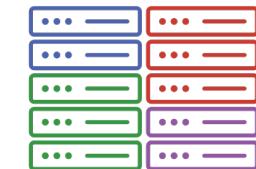


Paderborn
Center for
Parallel
Computing



Julia HPC Community

- Julia Slack
 - Channel: #hpc
- Monthly JuliaHPC Call
 - julialang.org/community/#events
- JuliaCon 2023 @ MIT, USA
 - 25 – 29 July



JULIA FOR HPC
JuliaCon 2023 Minisymposium



JuliaCon 2023 Hackathon @ MIT



Julia HPC Position Paper

arxiv.org/abs/2211.02740

USA / Canada

- **MIT** Massachusetts Institute of Technology
- **NERSC** National Energy Research Scientific Computing
- **ORNL** Oak Ridge National Laboratory
- **PI** Perimeter Institute for Theoretical Physics

Europe

- **PC2** Paderborn Center for Parallel Computing
- **HLRS** High-Performance Computing Center Stuttgart
- **UHH** University of Hamburg
- **CSCS** Swiss National Supercomputing Centre
- **ETH** Swiss Federal Institute of Technology Zürich
- **ARC@UCL** University College London

Bridging HPC Communities through the Julia Programming Language

Journal Title
XX(X):1–14
OT To Be Assigned (s) 2022
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/TobBeAssigned
www.sagepub.com/


Valentin Churavy¹, William F Godoy², Carsten Bauer³, Hendrik Ranocha⁴, Michael Schlotte-Lakemper⁵, Ludovic Räss^{6,7}, Johannes Blaschke⁸, Mosè Giordano⁹, Erik Schnetter^{10,11,12}, Samuel Omlin¹³, Jeffrey S. Vetter², Alan Edelman¹

Abstract

The Julia programming language has evolved into a modern alternative to fill existing gaps in the requirements of scientific computing and data science applications. Julia's single-language paradigm, and its proven track record at achieving high-performance without sacrificing user productivity, makes it a viable single-language alternative to the existing composition of high-performance computing (HPC) languages (Fortran, C, C++) and higher-level languages (Python, R, Matlab) suitable for data analysis and simulation alike. Julia's rapid growth in language capabilities, package ecosystem, and community make it a promising new universal language for HPC similar to C++ or Python – an achievable goal if the community is given the necessary resources. This paper presents the views of a multidisciplinary group of researchers in academia, government, and industry advocating for the use of Julia and its ecosystem in HPC centers. We examine the current practice and role of Julia as a common programming model to address major challenges in scientific reproducibility, data-driven artificial intelligence/machine learning (AI/ML), co-design, and in-situ workflows, scalability and performance portability in heterogeneous computing, network, data management, and community education. As a result, we consider necessary the diversification of current investments to fulfill the needs of the upcoming decade as more supercomputing centers prepare for the Exascale era.

Keywords

High Performance Computing, Julia, Programming Language

1 Programming Languages and HPC

The development of programming languages for high performance computing (HPC) has a rich and varied history. Early on, the needs of HPC and main-stream computing were still mostly aligned, leading to the development of Fortran as the first high-level language. Today, we see a bifurcation of programming languages designed for HPC, like Chapel ([Chamberlain et al. 2007](#)), and generalist programming languages, like Python and C++, that are being retooled for HPC use cases. Meanwhile, we see many new languages and programming models being deployed to help users manage the increased heterogeneity of contemporary systems: OpenCL, CUDA, HIP, OpenMP, OpenACC, SYCL, Kokkos, and many others.

Specialist languages lead to a smaller community and make it harder for attracting and recruiting developers who were educated in a more generalist language. We identify these smaller communities as a distinct risk for a language to grow to a sustainable size and to keep attracting users and investments. As an example of the three DARPA HPCS languages Chapel, X10, and Fortress only Chapel is still going.

In generalist languages, the needs of the HPC community are often not addressed or not deemed as important as other concerns. This leads to the development of HPC dialects, based around frameworks.

We will present Julia, a modern generalist programming language, that from the beginning was designed for

¹ Massachusetts Institute of Technology, USA

² Oak Ridge National Laboratory, USA

³ Paderborn Center for Parallel Computing, Paderborn University, Germany

⁴ Department of Mathematics, University of Hamburg, Germany

⁵ High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Germany

⁶ Laboratory of Hydraulics, Hydrology and Glaciology (VAW), ETH Zurich, Switzerland

⁷ Swiss Federal Institute for Forest, Snow and Landscape Research (WSL), Birmensdorf, Switzerland

⁸ National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, USA

⁹ Centre for Advanced Research Computing, United Kingdom

¹⁰ Perimeter Institute, 31 Caroline St. N., Waterloo, ON, Canada N2L 2Y5

¹¹ Department of Physics and Astronomy, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

¹² Center for Computation & Technology, Louisiana State University, Baton Rouge, LA 70803, USA

¹³ Swiss National Supercomputing Centre (CSCS), ETH Zurich, Switzerland

Corresponding author:
Valentin Churavy, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA
Email: vchuravy@mit.edu

Let's get started! 😊

github.com/carstenbauer/JuliaHLRS23

Attribution / Copyright

- Laptop: [Laptop Vectors by Vecteezy](#)
- Cluster: [Web Vectors by Vecteezy](#)