PADERBORN UNIVERSITY

HLRS
High-Performance Computing Center Stuttgart

NHR NATIONALES HOCHLEISTUNGS RECHNEN

PC² Paderborn Center for Parallel Computing

# Julia for High Performance Computing


## Carsten Bauer

Paderborn Center for Parallel Computing (PC2)


20-23 September 2022

# Schedule

| | Tuesday | Wednesday | Thursday | Friday |
|---|---|---|---|---|
| | **Foundations** | **"Serial"** | **Parallel** | **Accelerators** |
| 9:00- 11:00 (2h) | Introduction | Performance Optimisation | Distributed- | Exercises |
| | Types & Dispatch | | | GPU |
| | Short break | | | |
| 11:15- 12:45 (1,5h) | Exercises | Exercises | Computing | Programming |
| | Specialisation & | Microbench. | Exercises | Q&A |
| | Lunch break | | | |
| 14:15- 15:15 (1h) | Generic Prog. | SIMD & LinAlg | Experience Rep. | |
| | Short break | | | |
| 15:30 – 16:30 (1h) | Workflow & Pkgs | Profiling | Multithreading | |

# Course Material

github.com/carstenbauer/JuliaHLRS22

# Live Survey

# Why Julia for HPC?

# The Julia Programming Language

- **Great for domain science**
  - "Python-like" syntax
  - Dynamic & interactive
    - powerful REPL
    - Jupyter notebooks (Julia + Python + R)
  - Powerful linear algebra + ecosystem
  - Package manager, environments

- **Great for HPC**
  - 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
}
```

# Bridging the Gap

Domain Science


HPC

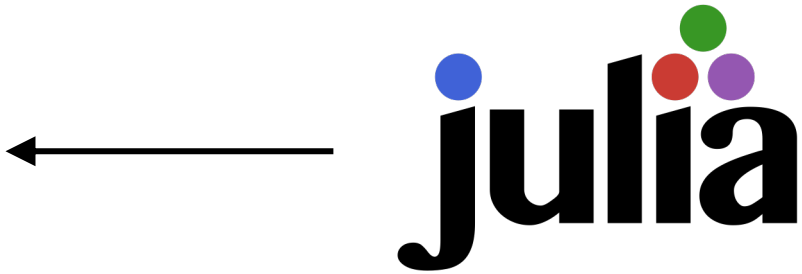
nature

”Julia: come for the syntax, stay for the speed”

DOI: 10.1038/d41586-019-02310-3

# Bridging the Gap

prototyping / slow



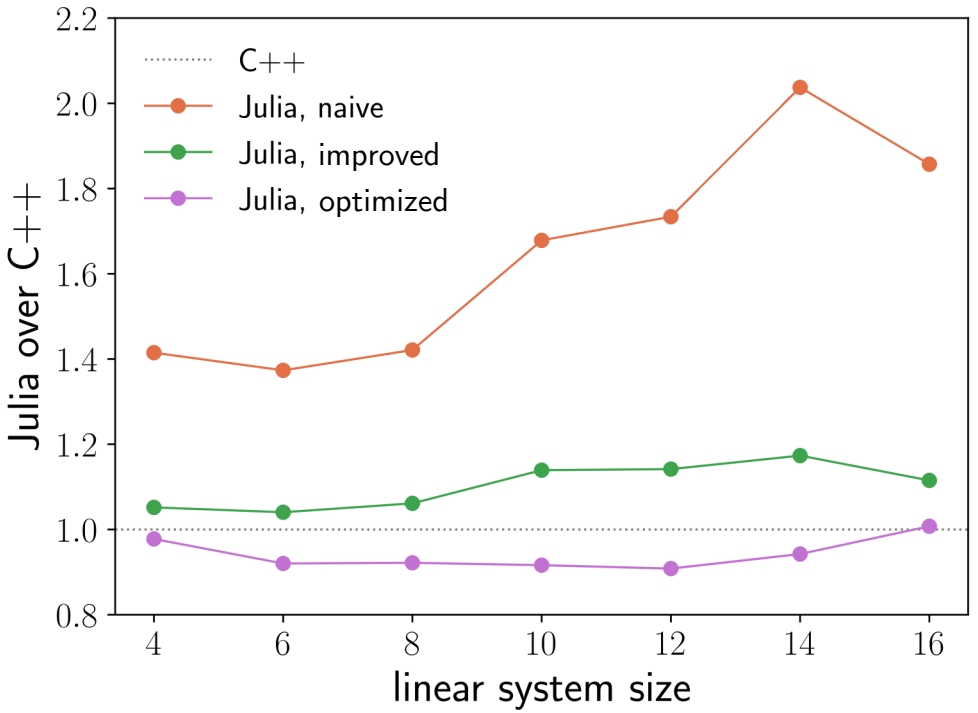

production / fast

**Julia covers the spectrum between those two “limits”**
**and can often provide the best of both worlds!**

# Personal Experience: Large-Scale Quantum Monte Carlo

- Condensed matter physics application
- Prototypical Julia implementation
  - ~3x faster than Python (not shown)
  - Within a factor of ~2x compared to C++

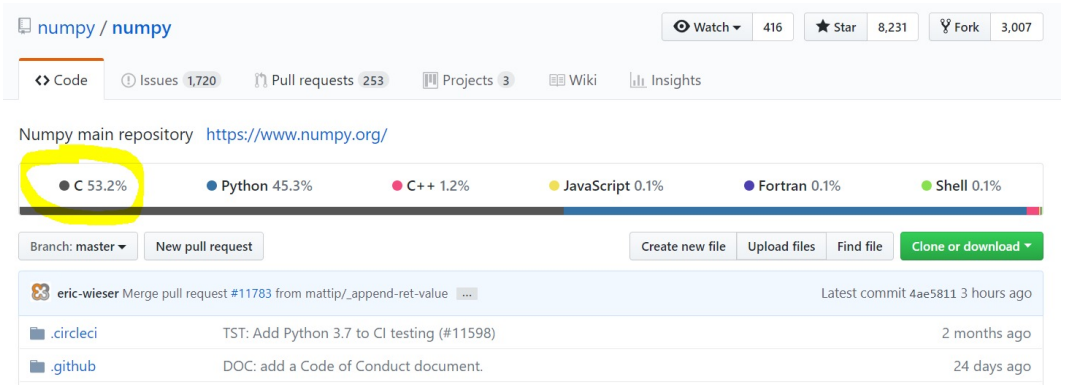
- Could **gradually** improve performance
  - No disruptive language change necessary!
  - A lot of fun!



# Bridging the Gap


**Start with HPC on your laptop and transition to clusters!**

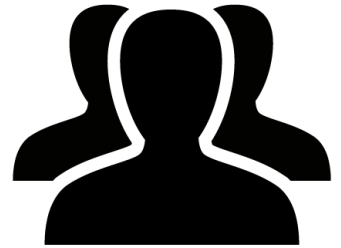
# Bridging the Gap





Developer

User



# Vandermonde matrix

$$V = \begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \dots & \alpha_2^{n-1} \\ 1 & \alpha_3 & \alpha_3^2 & \dots & \alpha_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_m & \alpha_m^2 & \dots & \alpha_m^{n-1} \end{bmatrix}$$

`vander(x)`

```
numpy.vander(x)
```

# Python

```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
```

calls

# C code

```c
PyUFunc_Accumulate(PyUFuncObject *ufunc, PyArrayObject *arr, PyArrayObject *out,
                   int axis, int otype)
{
    PyArrayObject *op[2];
    PyArray_Descr *op_dtypes[2] = {NULL, NULL};
    int op_axes_arrays[2][NPY_MAXDIMS];
    int *op_axes[2] = {op_axes_arrays[0], op_axes_arrays[1]};
    npy_uint32 op_flags[2];
    int idim, ndim, otype_final;
    int needs_api, need_outer_iterator;

    NpyIter *iter = NULL, *iter_inner = NULL;

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

    const char *ufunc_name = ufunc_get_name_cstr(ufunc);

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

    NPY_BEGIN_THREADS_DEF;

    NPY_UF_DBG_PRINT1("\nEvaluating ufunc %s.accumulate\n", ufunc_name);

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

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

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

    otype_final = otype;
    if (get_binary_op_function(ufunc, &otype_final,
                               &innerloop, &innerloopdata) < 0) {
        PyArray_Descr *dtype = PyArray_DescrFromType(otype);
        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;
    }

    ndim = PyArray_NDIM(arr);
```

```c
#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_arrays[0][idim] = idim;
        op_axes_arrays[1][idim] = idim;
    }

    /* The per-operand flags for the outer loop */
    op_flags[0] = NPY_ITER_READWRITE |
                  NPY_ITER_NO_BROADCAST |
                  NPY_ITER_ALLOCATE |
                  NPY_ITER_NO_SUBTYPE;
    op_flags[1] = NPY_ITER_READONLY;

    op[0] = out;
    op[1] = arr;

    need_outer_iterator = (ndim > 1);
    /* We can't buffer, so must do UPDATEIFCOPY */
    if (!PyArray_ISALIGNED(arr) || (out && !PyArray_ISALIGNED(out)) ||
            !PyArray_EquivTypes(op_dtypes[0], PyArray_DESCR(arr)) ||
            (out &&
             !PyArray_EquivTypes(op_dtypes[0], PyArray_DESCR(out)))) {
        need_outer_iterator = 1;
    }
    /* If input and output overlap in memory, use iterator to figure it out */
    else if (out != NULL && solve_may_share_memory(out, arr,
NPY_MAY_SHARE_BOUNDS) != 0) {
        need_outer_iterator = 1;
    }

    if (need_outer_iterator) {
        int ndim_iter = 0;
        npy_uint32 flags = NPY_ITER_ZEROSIZE_OK|
                           NPY_ITER_REFS_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
```

```c
    /* Get the output */
    if (out == NULL) {
        if (iter) {
            op[0] = out = NpyIter_GetOperandArray(iter)[0];
            Py_INCREF(out);
        }
        else {
            PyArray_Descr *dtype = op_dtypes[0];
            Py_INCREF(dtype);
            op[0] = out = (PyArrayObject *)PyArray_NewFromDescr(
                                    &PyArray_Type, dtype,
                                    ndim, PyArray_DIMS(op[1]), NULL,
NULL,
                                    0, NULL);
            if (out == NULL) {
                goto fail;
            }

        }
    }

    /*
     * 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_DIM(op[1], axis) == 0) {
        goto finish;
    }
    else if (PyArray_SIZE(op[0]) == 0) {
        goto finish;
    }

    if (iter && NpyIter_GetIterSize(iter) != 0) {
        char **dataptr_copy[3];
        npy_intp stride_copy[3];
        npy_intp count_m1, stride0, stride1;

        NpyIter_IterNextFunc *iternext;
        char **dataptr;

        int itemsize = op_dtypes[0]->elsize;

        /* Get the variables needed for the loop */
        iternext = NpyIter_GetIterNext(iter, NULL);
        if (iternext == NULL) {
            goto fail;
        }
        dataptr = NpyIter_GetDataPtrArray(iter);

        /* Execute the loop with just the outer iterator */
        count_m1 = PyArray_DIM(op[1], axis)-1;
```

```c
    if (count_m1 > 0) {
        /* Turn the two items into three for the inner loop */
        dataptr_copy[1] = stride1;
        dataptr_copy[2] = stride0;
        NPY_UF_DBG_PRINT1("iterator loop count %d\n",
                          (int)count_m1);
        innerloop(dataptr_copy, &count_m1,
                  stride_copy, innerloopdata);
    }
} while (iternext(iter));

NPY_END_THREADS;
}
else 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[1],
axis);

    NPY_UF_DBG_PRINT("UFunc: Reduce loop with no
iterators\n");

    if (PyArray_NDIM(op[0]) != PyArray_NDIM(op[1]) ||
            !PyArray_CompareLists(PyArray_DIMS(op[0]),
                                  PyArray_DIMS(op[1]),
                                  PyArray_NDIM(op[0]))) {
        PyErr_SetString(PyExc_ValueError,
                "provided out is the wrong size "
                "for the reduction");
        goto fail;
    }
    stride0 = PyArray_STRIDE(op[0], axis);

    stride_copy[0] = stride0;
    stride_copy[1] = stride1;
    stride_copy[2] = stride0;

    /* 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[0]);

    /*
     * Copy the first element to start the reduction.
     *
     * Output (dataptr[0]) and input (dataptr[1]) may point
to the
     * same memory, e.g. np.add.accumulate(a, out=a).
     */
    if (otype == NPY_OBJECT) {
```

uses

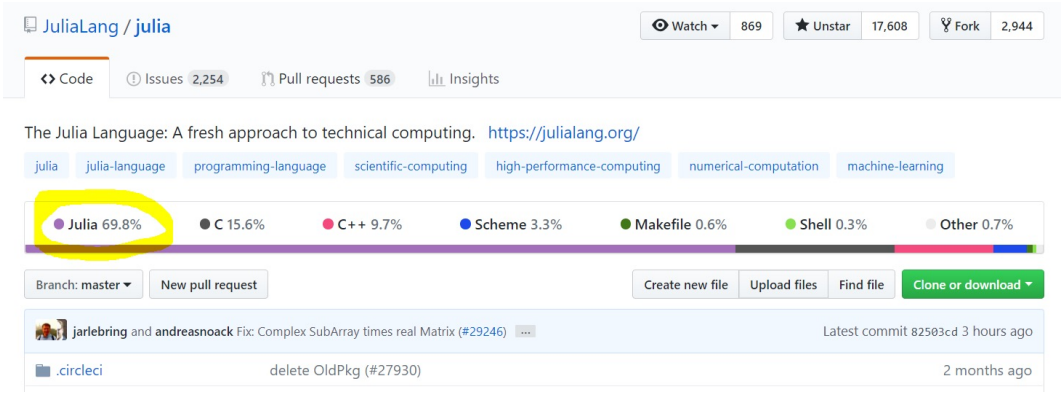
```c
@TYPE@_@kind@(char **args, npy_intp *dimensions, npy_intp *steps, void *NPY_UNUSED(func))
{
    if (IS_BINARY_REDUCE) {
#if @PW@
        @type@ * iop1 = (@type@ *)args[0];
        npy_intp n = dimensions[0];

        *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@ *)op1) = in1 @OP@ in2;
        }
    }
}
```

C template

# Julia

```
function vander(x::AbstractVector{T}) where T
    m = length(x)
    V = Matrix{T}(undef, m, m)
    for j = 1:m
        V[j,1] = one(x[j])
    end
    for i= 2:m
        for j = 1:m
            V[j,i] = x[j] * V[j,i-1]
            end
        end
    return V
end
```

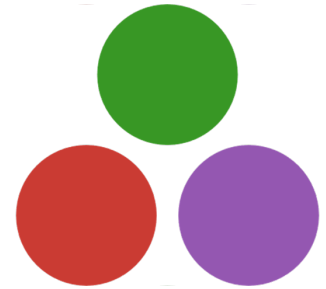
# Bridging the Gap





Developer

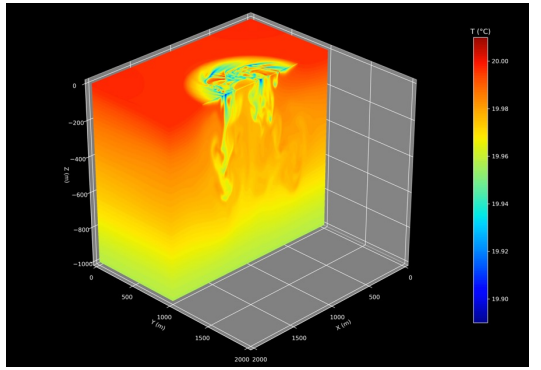
User



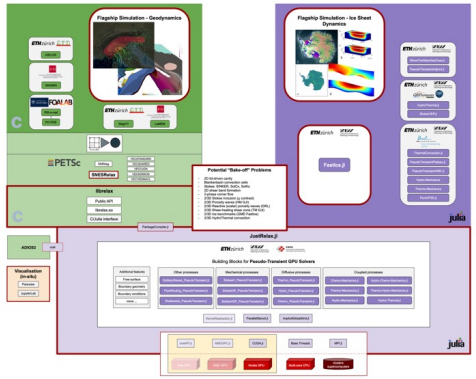
# Julia for HPC

# International Initiatives

- CliMA @ Caltech, MIT
  - 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

# Julia HPC Position Paper

## USA / Canada

- **MIT** Massachussets 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

(in preparation)

Bridging HPC Communities through the Julia Programming Language

Journal Title
XX(X):1–14
©The Author(s) 2022
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Valentin Churavy[1], William F Godoy[2], Carsten Bauer[3], Hendrik Ranocha[4], Michael Schlottke-Lakemper[5], Ludovic Räss[6,7], Johannes Blaschke[8], Mosè Giordano[9], Erik Schnetter[10,11,12], Samuel Omlin[13], Jeffrey S. Vetter[2], Alan Edelman[1]

**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 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] Massachussetts 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, University College London, Gower Street, London, WC1E 6BT, 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

# Julia HPC Community

- Julia Slack
  - Channel: #hpc
  - julialang.org/slack/


- Monthly JuliaHPC Call
  - julialang.org/community/#events

- JuliaCon 2022
  - 27 – 29 July
  - Recording: https://www.youtube.com/watch?v=fog1x9rs71Q


BROWN UNIVERSITY
CORNELL UNIVERSITY
FOUNDED A.D. 1865
EMORY UNIVERSITY
NORTHEASTERN UNIVERSITY
LVX VERITAS VIRTVS
BOSTON, MASSACHUSETTS
PSU
PENN STATE UNIVERSITY
PURDUE
SZKOŁA GŁÓWNA HANDLOWA W WARSZAWIE
Stanford University
UNIVERSIDAD NACIONAL AUTONOMA D MEXICO
THE UNIVERSITY OF EDINBURGH
UNIVERSITY of GLASGOW
UNIVERSITY of SOUTH FLORIDA
1956
TRUTH WISDOM
UNIVERSITE PAUL SABATIER
TOULOUSE III
UNIVERSITY OF CONNECTICUT
1881
Universitas Ledviensis
FIDE SPLENDET ET SCIENTIA
THE UNIVERSITY OF CALIFORNIA BERKELEY
1868
Western
UNIVERSITY · CANADA
ARIZONA STATE UNIVERSITY
1885
MŰEGYETEM 1782
ILLINOIS STATE UNIVERSITY
1857
Gladly we learn and teach
SIDERE MENS EADEM MUTATO
CUNY
THE CITY UNIVERSITY OF NEW YORK
EPFL
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
TOKYO METROPOLITAN UNIVERSITY
首都大学東京
Ucla
AGH

# Let’s get started! ☺

github.com/carstenbauer/JuliaHLRS22

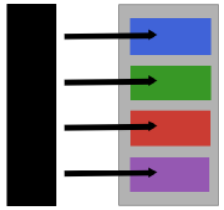
# Attribution / Copyright

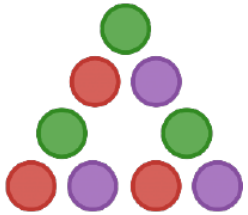
- Laptop: Laptop Vectors by Vecteezy
- Cluster: Web Vectors by Vecteezy

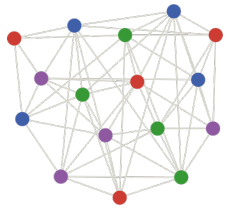
# HPC Ecosystem

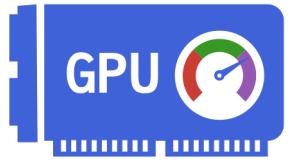
# HPC Ecosystem

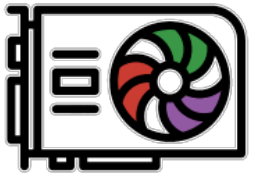
- JuliaSIMD
  - LoopVectorization.jl, Polyester.jl
- JuliaParallel
  - Dagger.jl, MPI.jl
- JuliaFolds
  - Transducers.jl, FLoops.jl
- JuliaPerf
  - LIKWID.jl, PProf.jl
- JuliaGPU
  - CUDA.jl, AMDGPU.jl







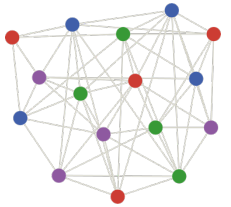



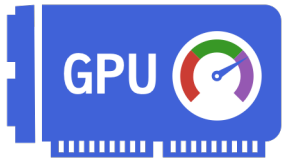
Paderborn Center for Parallel Computing

pc2.de/go/jlpkg

# Julia Packages by PC2

- ThreadPinning.jl
  - Dynamically pin Julia threads to CPU threads
- STREAMBenchmark.jl & BandwidthBenchmark.jl
  - Memory bandwidth measurements
- LIKWID.jl
  - Hardware performance monitoring
- GPUInspector.jl
  - Benchmark max. GPU performance
  - GPU node stresstest: monitor temperature & power




pc2.de/go/jlpkg