NEP Proposal

Background: The NumPy interface is popular and used worldwide. It is well documented, with many examples, running code, a battery of tests, has highly flexible arrays, and a large support system. Achieving this success is the most difficult part, thus we need to preserve the front end NumPy interface while increasing the adaptability of the low level loop back end to better handle the ever increasing demand for more data.

Problem: Many NumPy algorithms are embedded in Python C code. These loops are not easily portable to other platforms; use **int, long, long long** instead of **int32_t, int64_t;** and are often difficult to maintain and debug due to the macro expansion style written over a decade ago.

The NumPy memory layout is standard and transferrable to most computer languages (both column major and row major are supported), and its loops are universal algorithms, *yet hundreds of useful algorithms are coupled to Python only.*

Although compilers have modernized over the past decade, NumPy low level loops have largely remained the same, with layers of code scaffolding built up around the loop kernels.

Example:

```
#if @elsize@ != 16
#  if !(@elsize@ == 1 && @dst_contig@)
    @type@ temp;
#  endif
#else
    npy_uint64 temp0, temp1;
#endif
    if (N == 0) {
        return 0;
    }
#if @is_aligned@ && @elsize@ != 16
    /* sanity check */
    assert(N == 0 || npy_is_aligned(dst, _UINT_ALIGN(@type@)));
    assert(N == 0 || npy_is_aligned(src, _UINT_ALIGN(@type@)));
#endif
#if @elsize@ == 1 && @dst_contig@
    memset(dst, *src, N);
#else
```

Current NumPy hooks provide no user callback argument, allow for just one hook, and are either too early in the processing cycle, or too late. For an example please see the stubs.h file in the pnumpy project (thousands of stubs because no user callback).

Solution: *More flexible NumPy hooks and portable templated low level loops.* This will architect NumPy for the future and handle the growing need for large and huge arrays.

Many other developers will contribute when the low level loops become more portable and hookable because there are many ways to improve existing loops: vector intrinsics, local threading, GPU, and distributed computing. New sorting improvements with SIMD registers have been published. Some compilers offer AVX2 versions for many cmath functions, from **cbrt** to **log10** – most of which are faster than the current numpy implementation. Masked operations could also be improved, to name just a few.

This NEP proposes that lower level loops:

1) Are moved into a new portable C++ templated lib with a C API that is not Python dependent
2) Have a new hook API which allows hooks in 3 locations

TODO: Draw picture of old way and new way…

A new lib is created to ensure that Python code does not creep into the new C++ library, which in turn ensures its portability. A new lib allows the NumPy core developers to refactor the interfaces, which is also desirable.

The plain C API allows other languages to interface also (Python, Java, C++, C#, Go, Rust) similar to Operating System libs. The C API could be broken into major subsystems which may include:

Create, Compare, Convert, Unary, Binary, Ternary, Reduce, Sort, Mask, RNG, and more.

This is a different approach from the current PyArray_API which is flat, has over 300 routines today, some of which are deprecated, and many are unhookable.

The new library needs a name, such as NCLib for Numpy C Lib (pick any suitable name). Function names are exported in the C++ lib, such as: NclCreate, NclCompare, NclConvert, NclUnary, NclBinary, NclReduce, etc.

A pre-known DLL entry point in the NCLib/dll/so can be used to get to the Master Head of all subsystems. This one memory address will exopse hundreds of future function calls.

The process of removing the existing embedded loops in NumPy into Ncl can be done in phases because it is a large task.

Better Hooks

The new hooks could be done Linux Kernel style (note: same style chosen by Windows Kernel) with doubly linked list: https://0xax.gitbooks.io/linux-insides/content/DataStructures/linux-datastructures-1.html

Just copy */include/linux/list.h* into new Numpy C Lib.

```
struct list_head {
    struct list_head* next, * prev;
};

struct NclCompare_node {
    struct NclCompare_node* next, *prev;
    NCL_COMPARE_FUNC* compare_func;
    void*             usercallback;
};

struct NclUnary_node {
    struct NclUnary_node* next, *prev;
    NCL_UNARY_FUNC*   unary_func;
    void*             usercallback;
};
```

The Linux Kernel style has advantages over PyArray_API style:

1) The hook can chain in front, middle, or back
2) The hook can always call the next hook (punt to next)
3) The hook can remove itself (self->prev->next = self->next, etc.)
4) The NumPy C Lib remains in charge of all hooks, and can reset all hooks or return what has been hooked
5) The hook can have extra information in its node

It has two disadvantages:

1) An extra level of indirection (extra memory touched)
2) Memory is often allocated for the nodes

For example instead of the API NclCompare being declared as

```
#define NclCompare *(int (*)(int subfunc, ArrayObject* in1, ArrayObject* in2, ArrayObject** out1,
CompParams* pparams, void *usercallback))NCL_API[3])
```

where `NCL_API` is a global memory pointer to call addresses, it might be declared

```
enum NCL_SUBSYTSTEM {
   SUBSYTEM_INFO = 0, SUBSYTEM_CREATE = 1,  SUBSYSTEM_CONVERT = 2,
   SUBSYSTEM_COMPARE = 3, SUBSYSTEM_UNARY = 4, SUBSYSTEM_BINARY = 5,…
};
list_head API_HEADS[SUBSYSTEM_LAST];

#define NclCompare (NclCompare_node)(API_HEADS[SUBSYSTEM_CONVERT]->next)->compare_func

#define NclUnary   (NclUnary_node)(API_HEADS[SUBSYSTEM_UNARY]->next)->unary_func
```

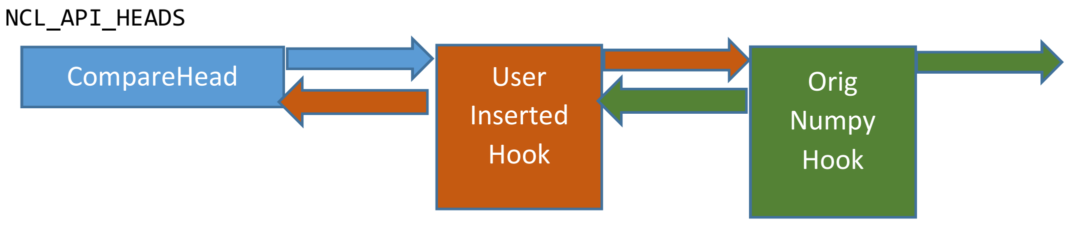
where `NCL_API_HEADS` is an array of ***list_heads*** that point to a ***NclCompare_node*** (struct) which then defines what the Compare function call signature is and provides additional storage for usercallback.

There are call tables per subsystem (like Convert, Unary, Binary..) and each call table is a ***list_head*** (per Linux Kernel design) allowing a new custom ***list_node*** to be inserted in front when hooked.

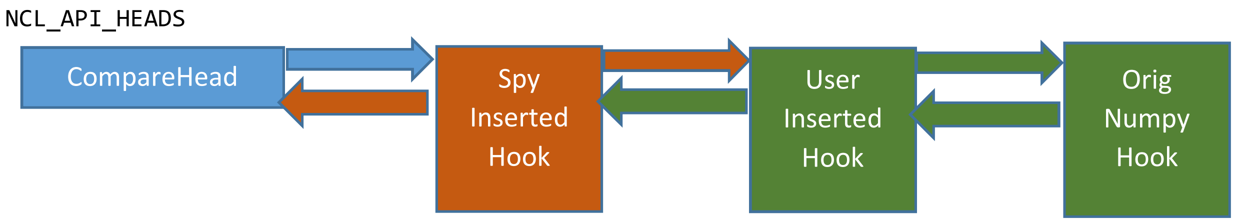
NclApiHook(subsystem, list_head* node_ptr);

NclKernelHook(subsystem, subfunction, * signature, list_head* node_ptr);

How a hook works, insert a user hook in front of the original NumPy routine



Insert a spy/debugger/ledger



If we wanted to time to see if new hook made things faster, we could insert yet another hook in front and time each operation. This could also buffer operations until it was forced to send them.

The user hook could also remove itself without calling any functions

Walk Through Example with compare operator

The user types in "a > b"

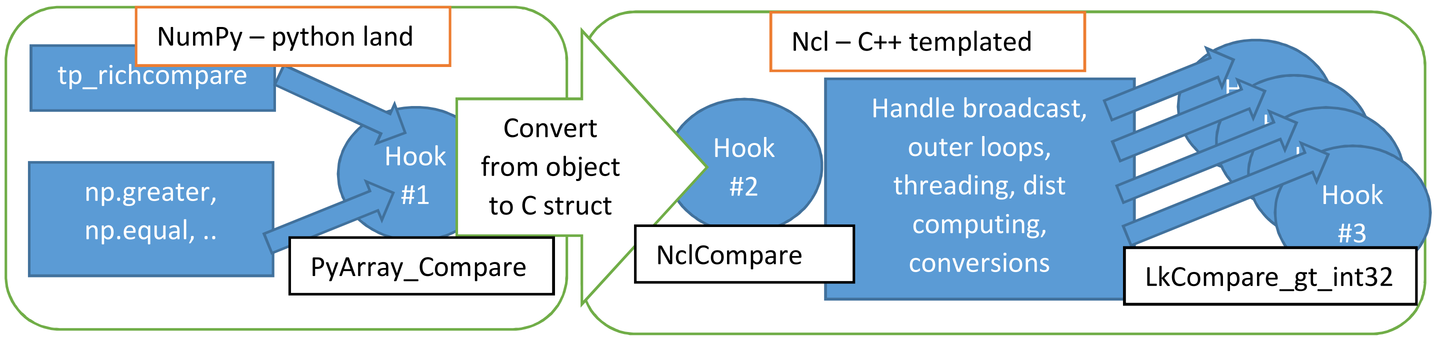
>>> **a > b**

In python this will call the compare hook in the NumPy array type object which will pass in two python objects and expects another object returned.

```
    richcmpfunc comparefunc = ArrayObject->ob_type->tp_richcompare;
```

The **`tp_richcompare`** is routed to the common compare routine (the same routine **`np.greater`** is routed to). A numpy user may hook early at this point –this is a python C level hook (we are still in python land and have not yet hopped over the wall into the NclLIB). A pure python callback can be done from C to Python. The Python Hook for Compare will process the inputs, report any errors, and convert it to C only structs – just like a java hook would if numpy arrays were ported to java.

For example in the line "**a > b**", b could be a scalar, which could be a Python scalar or NumPy scalar (**b=4.5**). It could be a list "**b=[3,4,5]**" or it could be a string, iterator, or any valid Python object.



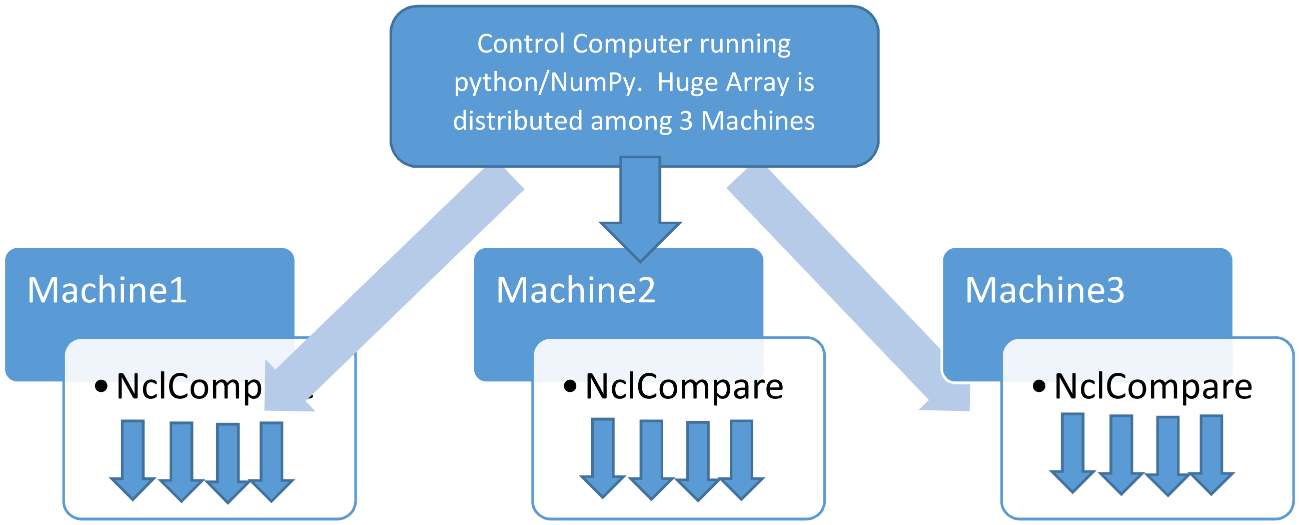
- ➔ Hook #1 PyArray_Compare -> convert python objects to C representation, call hook #2
- ➔ Hook #2 Call NclCompare(subfunction, input1, input2, output1, pparams ,usercallbackarg)
- ➔ Hook #3 Loop kernel LkCompare for dtype such as *cmp_gt_int32(input1, input2, output1, usercallbackarg)*

NOTE: There are 3 hook locations here (all doubly linked list Linux Kernel style hooks)

1) The first hook is for python to centralize and funnel all the ways comparison operations can come in (from both richcmpfunc '>' and the np.greater operations). This first stage also converts Python objects to C
2) The second hook is the portable C funnel. A hook here is important because
   a. All comparisons return a BOOL (centralize output alloc)
   b. All comparisons have a possible upcast stage
   c. All comparisons have to handle broadcasted and multi-dim strided loops
   d. Handle common functions with the SAME routine as opposed to duplicating code.
   e. This allows a user to hook comparisons for *int64 == uint64* to intercept the mantissa lossy upcast to float64 and include a proper comparison routine.
   f. Also allows a threader to break up large arrays and divide the work.
3) The third hook allows for optimized routines
   a. Such as vector intrincs mm256_cmp_gt
   b. This is the simplest hook
   c. Custom comparison routines can also go here

With distributed computing on a huge arrays, the output array is allocated on the remote computer. Any conversions are also done on the remote computer.

When threading is used, the work for large arrays is subdivided. Due to CPU memory hierarchy, conversions (or upcasting) should be done on the same thread performing the comparison as opposed to the main thread in a separate pass. Similarly, for distributed computing, the NclCompare operation for a huge array must also be subdivided and sent over the network to each machine. Each machine may then subdivide again using threads. The output array (in this case a Boolean array) will be allocated in partitions at each machine. Each machine reports back when the operation is completed. The control computer then returns back to PyArray_Compare, which wrap the result in a Python object and returns that object back to richcmpfunc, which returns control to python's main thread.


Control Computer running python/NumPy. Huge Array is distributed among 3 Machines
Machine1
• NclComp
Machine2
• NclCompare
Machine3
• NclCompare

NclCompare **has 6 arguments**

NclCompare(*NclCompare_node *self, enum subfunction, ArrayObject* input1, ArrayObject* input2, ArrayObject** output1, CompareParams *pparams)*

subfunction is an enum of >, <, >=, <=, ==, !=

input1/input2 is a pointer to the ArrayObject

output1 can be null and the caller can return an allocated array, thus the **

pparams can be null or specify parameters such as axis= or where=

self a pointer to the NclCompare_node which contains data on which hook to call next, any other fields the hook added to the end of the node.

Every argument can be serialized to send to threads, GPUs, or other distributed computers.

NclCompare knows how to resolve or broadcast loops (the C Lib side as opposed to the python side), for instance if input1 is 1 dimensional and input2 is 2 dimensional then input1 is a new array (came from a list for instance) it might be broadcast to all the second dimension. If input1 is small, it might have to be sent over to network to Machine1,2,3 so those machines can apply input1 to input2.

Python dtype.num is converted to Ncl dtypes. This decouples Ncl from the existing NumPy dtype system. Python "Object" arrays require more discussion. Some object array routines do not need to know about the object (for instance a Boolean mask can move data with no knowledge of what the data is). After the new array is created, it can be IncRef'd back in Python land at Hook #1 which understand PyObjects (Hooks #2 and #3 are in C Lib land).

The same is true for arrays allocated in Hook #2. In a compare function the output array returned is a Boolean, which is allocated in Hook #2 because this hook is aware of distributed computing. Hook #1 can then wrap the returned array as a python object. When the python object is DecRef'd to zero, it can then call back into the NclLib to delete the array.

This also solves the PyArray_IsSameDType problem where **long** and **long long** are the same on the gcc compiler because similar dtypes are impossible in the NcLib dtype system since based on bit widths (aka, int32_t instead of int).

NclCompare (Hook #2) will check for dtype conversions, it may call NclConvert and convert on the stack or small buffer before calling Hook #3.

We need more discussion on the GetConvertFunction(), but it is similar for GetDtypeTransfer functions.

The 'C' Struct used by NclLib can be the same as PyArrayObject and PyArray_Descr for now to simplify coding during phase 1. Example below.

```
typedef struct NclArrayObject {
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    OPAQUE   *base;
    NclArray_Descr *descr;
    /* Flags describing array -- see below */
    int flags;
    OPAQUE    *weakreflis;
}
```

TBD – show Descr struct.

<u>Walk Through Example with add operator</u>

Similar to the compare walkthough, addition can come in more than one way… from the + and from np.add(). Just like Compare subsystem has three hooks, so does the Binary subsystem. Just like compare has subfunctions, so does the binary subsystem (add, subtract, multiply, …).

Likewise there may exist an IsUnary subsystem where all the np.is*something* functions (which always return a Boolean) are funneled into.

Appendix

Large Arrays and Huge Arrays

Although data is doubling in size every 18 months for most data analytics fields (AI, web, social media, finance), the computer's tiered memory hierarchy has remained stable for decades.

The chart below is only meant as a rough estimate for a 2020 Intel i9 server computer. Times and speeds will vary as technology changes.

Memory Level	**Latency**	**Speed**	**Size**
L1 cache	< 2ns	1500 GB/sec	~32K code+data
L2 cache	~5ns	600 GB/sec	~256KB
L3 cache	~15ns	200 GB/sec	~16MB
DRAM/NUMA	~50ns	16 GB/sec	<1 TB
PCI BUS to GPU	~3000ns	+8 GB/sec *see note	GPU memory ~ 16GB
Network	~5000ns	+8 GB/sec	Infinite

- GPU contains DDR6(X) memory at ~ 1000GB/sec (1TB/sec)

Array Classification	**Size (number elements)**	**Speed up Technique**
Small Array	< 64K	Same thread
Large Array	+64K	Employ local threading
Huge Array	+2GB	Employ distributed computing

The cost to wake up additional threads makes threading often only useful for large arrays, or expensive operations such as sort.

The cost to communicate to other computers over the network makes distributed computing often only useful for huge arrays.