GAIN: DISTRIBUTED ARRAY COMPUTATION WITH PYTHON

By

JEFFREY ALAN DAILY

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2009

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of JEFFREY ALAN DAILY find it satisfactory and recommend that it be accepted.

Robert R. Lewis, Ph.D., Chair

Li Tan, Ph.D.

Kevin Glass, Ph.D.

ACKNOWLEDGEMENT

I would like to thank Dr. Lewis for his encouragement and for always setting such high expectations. Also, Battelle Memorial Institute for funding my way through graduate school.

GAIN: DISTRIBUTED ARRAY COMPUTATION WITH PYTHON

Abstract

by Jeffrey Alan Daily, M.S.
Washington State University
May 2009

Chair: Robert R. Lewis

Scientific computing makes use of very large, multidimensional numerical arrays – typically, gigabytes to terabytes in size – much larger than can fit on even the largest single compute node. Such arrays must be distributed across a "cluster" of nodes.

Global Arrays is a cluster-based software system from Battelle Pacific Northwest National Laboratory that enables an efficient, portable, and parallel shared-memory programming interface to manipulate these arrays. Written in and for the C and FORTRAN programming languages, it takes advantage of high-performance cluster interconnections to allow any node in the cluster to access data on any other node very rapidly.

The "numpy" module is the de facto standard for numerical calculation in the Python programming language, a language whose use is growing rapidly in the scientific and engineering communities. numpy provides a powerful N-dimensional array class as well as other scientific computing capabilities. However, like the majority of the core Python modules, numpy is inherently serial.

Our system, GAiN (Global Arrays in NumPy), is a parallel extension to Python that accesses Global Arrays through numpy. This allows parallel processing and/or larger problem sizes to be harnessed almost transparently within new or existing numpy programs.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Dedication

To Nicole, Abigail, and Grace – for without my family I would have nothing.

CHAPTER ONE

INTRODUCTION

Scientific computing with Python typically involves using the NumPy package. NumPy provides an efficient multi-dimensional array and array processing routines. Unfortunately, like many Python programs, NumPy is serial in nature. This limits both the size of the arrays as well as the speed with which the arrays can be processed to the available resources on a single compute node.

NumPy programs are written, debugged, and run on single machines. This may be sufficient for certain problem domains. However, NumPy may also be used to develop prototype software. Such software is usually ported to a different, compiled language and/or explicitly parallelized to take advantage of additional hardware.

GAiN is an extension to Python and provides parallel, distributed processing of arrays. It implements a subset of the NumPy API so that for some programs, by simply importing `gain` in place of `numpy` they may be able to take advantage of parallel processing automatically. Other programs may require slight modification. This allows those programs to take advantage of the additional cores available on single compute nodes and to increase problem sizes by distributing across clustered environments.

Chapter 2 provides all of the background information necessary to understand GAiN. Chaptper 3 highlights systems similar to GAiN. Chapter 4 describes all of the subsystems and concepts that went into GAiN. Chapter 5 briefly documents the difficulties encountered while carrying out the design. Chapter 6 evaluates the performance of GAiN compared to NumPy as well as other attempts to make NumPy more efficient. Chapter 7 concludes and Chapter 8 describes how GAiN can be extended or used as the basis of future research.

CHAPTER TWO

BACKGROUND

Like any complex piece of software, GAiN builds on many other foundational ideas and implementations. This background is not intended to be a complete reference of the subjects herein, rather only what is necessary to understand the design and implementation of GAiN. Further details may be found by examining the references or as otherwise noted.

2.1 Python

Python[28, 46] is a machine-independent, bytecode interpreted, object-oriented programming (OOP) language. It can be used for rapid application development, shell scripting, or scientific computing to name a few. It gives programmers and end users the ability to extend the language with new features or functionality. Such extensions can be written in C[45], C++[15], FORTRAN[36], or Python. It is also a highly introspective language, allowing code to examine various features of the Python interpreter at run-time and adapt as needed.

2.1.1 Operator Overloading

User-defined classes may implement special methods that are then invoked by built-in Python functions. This allows any user-defined class to behave like Python built-in objects. For example, if a class defines `__len__()` it will be called by the built-in `len()`. There are special methods for object creation, deletion, attribute access, calling objects as functions, and making objects act like Python sequences, maps, or numbers. Classes need only implement the appropriate overloaded operators.

2.1.2 object construction

User-defined classes control their instance creation using either the overloaded `__new__()`, `__init__()`, or both. `__new__()` is a special-cased static method that takes the class of which an instance was requested as its first argument. It must return a class instance, but it need not be an instance of the class that was requested. If `__new__()` returns an instance of the requested class, then the instance's `__init__()` is called. If some other class instance is returned, then `__init__()` is not called on that instance. `__new__()` was designed to allow for subclasses of Python's immutable types but can be used for other purposes.

2.1.3 Slicing

Python supports two types of built-in sequences, immutable and mutable. The immutable sequences are the `string`s, `unicode`s, and `tuple`s while the only mutable type is the `list`. Python sequences generally support access to their items via bracket "[]" notation and accept either an integer k where $0 <= k < N$ and N is the length of the sequence, or a `slice` object. Out-of-bounds indices will result in an error, however negative indices are supported by the built-in Python sequences by calculating offsets from the end of the sequence. It is up to the implementing class whether to support negative indices when overloading the sequence operators.

`slice` objects are used for describing *extended slice syntax*. `slice` objects describe the beginning, end, and increment of a subsequence via the `start`, `stop`, and `step` attributes, respectively. When `slice`s are used within the bracket notation, they can be represented by using the `slice()` constructor or as colon-separated values. The start value is inclusive of its index while the stop value is not. If the start value is omitted it defaults to 0. Similarly, stop defaults to the length of the sequence and step defaults to 1. The `slice` can be used in lieu of an index for accessing a subsequence of the sequence

object. Slicing a built-in Python sequence always returns a copy of the returned subsequence. User-defined classes are free to abandon that convention. Further, `slice`s that are out-of-bounds will silently return an in-bounds sequence which is different behavior than simple single-item indexing.

To illustrate both index and `slice` access to Python sequences assume we have the `list` of `int`s `A=[0,1,2,3,4,5,6,7,8,9]`. An example of single-item access would be `A[1]=1`. Using `slice` syntax would looke like `A[1:2]=[1]`. A negative single-item index looks like `A[-1]=9`. Finally, an example of `slice` syntax that includes a `step` is `A[2:9:3]=[2,5,8]`. Note in these examples how single-item access returns an `int` whereas slicing notation returns a `list`.

2.1.4 Function Decorators

Everything in Python is an `object`, including functions. That means functions can be passed as arguments to other functions. This allows functions to wrap other functions quite easily and has always been available to the Python programmer. As of Python 2.4, a function that takes another function as its first argument and returns a function is given special importance as a *decorator*.

Decorators allow a form of metaprogramming[37], allowing functions to be wrapped by one or more additional functions at the time of their definition. They are defined like any other Python function. The decorator's label is then placed immediately above the function to be decorated and denoted by the "@" symbol. Python uses decorators internally to implement class and static methods, `@classmethod` and `@staticmethod` respectively. One use for decorators besides those already mentioned would be for logging function calls. Other uses might be to enforce the types of inputs or of outputs since Python is dynamically typed. See the sample code in Appendix A.1 for clarity.

```
def my_decorator(function_to_wrap):
    def new_function():
        print "Hello World"
        return function_to_wrap
    return new_function
@my_decorator
def function_getting_wrapped():
    pass
```

Figure 2.1: Example of Python Function Decorators.

2.1.5 Bytecode

Python is a machine-independent bytecode-interpreted language. When a Python program is written, the source code is compiled by Python into an intermediate, internal Python representation called bytecode. The bytecode is machine-independent and consists of instructions for the Python interpreter to carry out. This bytecode is cached as *.pyc or *.pyo files so that running the same code is faster after the first time. Bytecode can be examined or replaced at run-time. For a function, the bytecode is accessible from the attribute `func_code` as a `code object`. The returned object can be replaced with any other `code object`, allowing functions to change their behavior at run-time.

2.1.6 Object Serialization

Many languages have the need for object serialization and de-serialization. Serialization is the process by which a class instance is transformed into another, often succinct, representation (like a byte stream) which can persist or be transmitted. It can then be de-serialized back into its original object. This is also commonly known as marshalling and unmarshalling. In Python this is also called pickling and unpickling.

`pickle`[28] is part of the Python Standard Library and can serialize nearly everything in Python. It is designed to be backwards compatible with earlier versions of Python. Certain Python objects, such as modules or functions, are serialized by name only rather than

by their state. They can only be de-serialized so long as the same module or function exists within the scope where the de-serialization occurs. There is no guarantee that the same function will result from the de-serialization or that the function exists in that namespace.

There is another module for performing serialization called `marshal`[28]. `marshal` does not support as many classes as `pickle` nor is it version independent. However, `marshal` supports serializing Python code objects while `pickle` does not. Recall from 2.1.5 that `code objects` represent Python bytecode and are the instructions of defined functions. By using the `marshal` module one can serialize Python functions.

2.1.7 ctypes

`ctypes`[46] is a Python module that allows functions in shared libraries to be called directly from Python. This allows libraries written in C to be wrapped in pure Python versus other wrapping options[6]. `ctypes` also has representations of the C data types which are are more efficient yet less flexible than the Python built-ins. All functions called via `ctypes` must have their parameters converted to one of these C data types. Automatic conversion is only available for integers and strings. `ctypes` also exposes the otherwise internal Python C-API to native Python programs. `ctypes` became a part of the Python Standard Library as of Python 2.5.

2.2 NumPy

`numpy`[32, 31] is a Python extension module which adds a powerful multidimensional array class `ndarray` to the Python language. NumPy also provides scientific computing capabilities such as basic linear algebra and Fourier transform support. NumPy is the de facto standard for scientific computing in Python and the successor of the other numerical Python packages Numarray[42] and numeric[2].

2.2.1 ndarray

The primary class defined by NumPy is the `ndarray`. The `ndarray` is implemented as a contiguous memory segment that is either FORTRAN- or C-ordered. Recall that in FORTRAN, the first dimension changes the fastest while it is the opposite (last) dimension in C. All `ndarray`s have a pointer to the location of the first element as well as the attributes `shape`, `ndim`, and `strides`. `ndim` describes the number of dimensions in the array, `shape` describes the number of elements in each dimension, and `strides` describes the number of bytes between consecutive elements per dimension. `shape` can be modified while `ndim` and `strides` are read-only and used internally, although their exposure to the programmer may help in developing certain algorithms.

The `ndarray` does not implement Python's `__init__()` object constructor. Instead, `ndarray`s use the `__new__()` `classmethod`, treating `ndarray`s as if they were immutable objects. Recall from 2.1.2 that `__new__()` is Python's hook for subclassing its built-in objects since otherwise the `__init__()` method of the built-in's subclass would never be called. The creation of `ndarray`s is complicated by the need to return views of `ndarray`s that are also `ndarray`s. However, due to the use of `__new__()`, subclasses of `ndarray` would never get the chance to modify their attributes during construction. `__array_finalize__()` is called instead of `__init__()` for `ndarray` subclasses to avoid this limitation.

2.2.2 Slicing

Unlike slicing with built-in Python sequences, slicing in NumPy is performed per axis. Each sliced axis is separated by commas within the usual bracket notation. Further, slicing in NumPy produces "views" rather than copies of the original `ndarray`. If a copy of the result is truly desired, it can be explicitly requested. This allows operations on subarrays without unnecessary copying of data. To the programmer, `ndarray`s behave the same

whether they are the result of a slicing operation. Views have an additional attribute, `base`, assigned that points to the `ndarray` that owns the data. The original `ndarray`'s `base` is `None` (effectively a null pointer.) When an `ndarray` is sliced, the resulting `ndarray` may have different `shape`, `strides`, and `ndim` attributes appropriately. There is no restriction on taking slices of already sliced `ndarray`s, either.

For example, the following array A is a 3x4x5 array of integers. Zeros were prepended to smaller numbers for clarity.

000	001	002	003	004	100	101	102	103	104	200	201	202	203	204
010	011	012	013	014	110	111	112	113	114	210	211	212	213	214
020	021	022	023	024	120	121	122	123	124	220	221	222	223	224
030	031	032	033	034	130	131	132	133	134	230	231	232	233	234

Below is a sample of slicing notations and their results on A.

$A[0, 2, 4] = 024$.

$$A[1, 0:2, 1:5:2] = \begin{matrix} 101 & 103 \\ 111 & 113 \end{matrix}$$

$$A[:, ::-1, 1:5:2] = \begin{matrix} 031 & 033 & 131 & 133 & 231 & 233 \\ 021 & 023 & 121 & 123 & 221 & 223 \\ 011 & 013 & 111 & 113 & 211 & 213 \\ 001 & 003 & 101 & 103 & 201 & 203 \end{matrix}$$

There are a few special cases of slicing implemented by NumPy. The form described previously is standard slicing. NumPy also supports a form of slicing called *fancy slicing*. Fancy slicing allows other `ndarray`s to be used as the indices into other arrays so long as they consist of integers or booleans. The result of indexing with an integer `ndarray` causes the resulting array to have the same shape as the index array with its integer elements

replaced by their corresponding values in the array being indexed. Indexing with a boolean array causes the corresponding `True` elements to be selected.

2.2.3 Universal Functions

The element-wise operators in NumPy are known as *Universal Functions*, or *ufuncs*. Many of the methods of the `ndarray` simply invoke the corresponding ufunc. For example, the operator `+` calls `ndarray.__add__()` which invokes the ufunc `add`. Ufuncs are either unary or binary, taking either one or two arrays as input, respectively. Ufuncs always return the result of the operation as an array. Optionally, an additional array may be specified to receive the results of the operation. Specifying this output array to the ufunc avoids the sometimes unnecessary creation of a new array.

Ufuncs are more than just callable functions. They also have some special methods such as `reduce` and `accumulate`. `reduce` is similar to Python's built-in function of the same name that repeatedly applies a callable object to its last result and the next item of the sequence. This effectively reduces a sequence to a single value. When applied to arrays the reduction occurs along the first axis by default, but other axes may be specified. Each ufunc defines the function that is used for the reduction. For example, `add` will sum the values along an axis while `multiply` will generate the running product. `accumulate` is similar to `reduce`, but it returns the intermediate results of the reduction.

Ufuncs can operate on objects that are not `ndarrays`. In order for subclasses of the `ndarray` or `ndarray`-like objects to utilize the ufuncs, they may define three methods and one attribute which are `__array_finalize__()`, `__array_wrap__()`, `__array__()`, and `__array_priority__`, respectively. `__array_wrap__` takes an `ndarray` as its only argument and expects a subclass of `ndarray` to be returned. In the case of binary ufuncs, the input arrays may be different subclasses of the `ndarray`. Since ufuncs return the base `ndarray` as a result of their execution, `__array_wrap__`

is used to return a subclass of `ndarray` for the input array with the highest `__array_priority__`. If an object is specified to receive the results of the ufunc and implements `__array__()`, the results from the ufunc will be written to the array returned by that method.

2.2.4 Broadcasting

NumPy introduces the powerful feature of allowing otherwise incompatible arrays to be used as operands in element-wise operations. If the number of dimensions do not match for two arrays, 1's are repeatedly prepended to the shape of the array with the least number of dimensions until their `ndim`s match. Arrays are then broadcast-compatible (also *broadcastable*) if for each of their dimensions their shapes either match or one of them is equal to 1. For example, the shapes $(3, 4, 5)$ and $(2, 3, 4, 1)$ are broadcastable. In this way, scalars can be used as operands in element-wise array operations since they will be broadcast to match any other array. Broadcasting relies on the `strides` attribute of the `ndarray`. A stride of 0 effectively causes the data for that dimension to repeat, which is precisely what happens when broadcasting occurs in element-wise array operations.

2.3 Parallel Programming Paradigms

Parallel applications can be classified into a few well defined programming paradigms. Each paradigm is a class of algorithms that have the same control structure. The literature differs in how these paradigms are classified and the boundaries between paradigms can sometimes be fuzzy or intentionally blended into hybrid models[7]. The Master/Slave and SPMD paradigms are discussed further.

2.3.1 Master/Slave

The master/slave paradigm, also known as task-farming, is where a single master process farms out tasks to multiple slave processes. The control is always maintained by the master,

dispatching commands to the slaves. Usually, the communication takes place only between the master and slaves. This model may either use static or dynamic load-balancing. The former involves the allocation of tasks to happen when the computation begins whereas the latter allows the application to adjust to changing conditions within the computation. Dynamic load-balancing may involve recovering after the failure of a subset of slave processes or handling the case where the number of tasks is not known at the start of the application.

2.3.2 Single-Program Multiple-Data (SPMD)

With SPMD, each process executes essentially the same code but on a different part of the data. The communication pattern is highly structured and predictable. Occasionally, a global synchronization may be needed. The efficiency of these types of programs depends on the decomposition of the data and the degree to which the data is independent of its neighbors. These programs are also highly susceptible to process failure. If any single process fails, generally it causes deadlock since global synchronizations thereafter would fail.

2.4 Message Passing Interface (MPI)

Message passing is one form of inter-process communication. Each process is considered to have access only to its local memory. Data is transferred between processes by the sending and receiving of messages which usually requires the cooperation of participating processes. Communication can take the form of one-to-one, one-to-many, many-to-one, or many-to-many.

Message passing libraries allow efficient parallel programs to be written for distributed memory systems. MPI[19], also known as MPI-1, is a library specification for message-passing that was standardized in May 1994 by the MPI Forum. It is designed for high performance on both massively parallel machines and on workstation clusters. An MPI implementation exists on nearly all modern parallel systems and there are a number of

freely available, portable implementations for those systems that do not[7]. As such, MPI is the de facto standard for writing massively parallel application codes in either FORTRAN, C, or C++.

MPI programs are typically started with either `mpirun` or `mpiexec`, specifying the number of processes to invoke. If the MPI program is run without the use of those, then it is run as if only one process was specified. Not all MPI implementations support running without the use of the `mpirun` or `mpiexec` programs. MPI programs can query their environment to determine how many processes were specified. Further, each process can query to determine which process they are out of the total number specified.

MPI programs are typically conform to the SPMD paradigm[7]. The `mpiexec` programs by default launch programs for this type of parallelism. A single program is specified on the command line which gets replicated to all participating processes. This same program is then executed within its own address space on each process, such that any process knows only its own data until it communicates with other processes, passing messages (data) around. A “hello world” program executed in this fashion would print ”hello world” once per process.

2.4.1 MPI-2

The MPI-2 standard[20] was first completed in 1997 and added a number of important additions to MPI including, but not limited to, process creation and management, one-sided communication, parallel file I/O, and the C++ language binding. With MPI-2, any single MPI process or group of processes can invoke additional MPI processes. This is useful when the total number of processes required for the problem at hand cannot be known a priori.

Before MPI-2, all communication required explicit handshaking between the sender and receiver via `MPI_Send()` and `MPI_Recv()` in addition to non-blocking variants.

MPI-2's one-sided communication model allows reads, writes, and accumulates of remote memory without the explicit cooperation of the process owning the memory. If synchronization is required at a later time, it can be requested via `MPI_Barrier()`. Otherwise, there is no strict guarantee that a one-sided operation will complete before the data segment it accessed is used by another process.

Parallel I/O in MPI-2, sometimes referred to as MPI-IO, allows for single, collective files to be output by an MPI process. Before MPI-IO, one such I/O model for SPMD programs was to have each process write to its own file. Having each process write to its own file may be fast, however in most cases it requires substantial post-processing in order to stitch those files back together into a coherent, single-file representation thus diminishing the benefit of parallel computation. Other forms of parallel I/O before MPI-IO was introduced included having all other processes send their data to a single process for output. However, any computational speed-ups from the parallelism are reduced by having to communicate all data back to a single node. MPI-IO hides the I/O model behind calls to the API, allowing efficient I/O routines to be developed independently of the calling MPI programs. One such popular implementation of MPI-IO is ROMIO[41].

2.4.2 mpi4py

mpi4py is a Python wrapper around MPI written to mimic the C++ language bindings. It supports point-to-point communication as well as the collective communication models. Typical communication of arbitrary objects in the FORTRAN or C bindings of MPI require the programmer to define new MPI datatypes. These datatypes describe the number and order of the bytes to be communicated. On the other hand, strings could be sent without defining a new datatype so long as the length of the string was understood by the recipient. mpi4py is able to communicate any `pickle`able Python object since `pickle`d objects are just byte streams. mpi4py also has special enhancements to efficiently communicate

any object implementing Python's buffer protocol, such as NumPy arrays. It also supports dynamic process management and parallel I/O. [10, 9, 11]

2.5 Global Arrays (GA)

The GA toolkit[30, 29, 17] is a software system from Battelle Pacific Northwest National Laboratory that enables an efficient, portable, and parallel shared-memory programming interface to manipulate physically distributed dense multidimensional arrays, without the need for explicit cooperation by other processes. GA compliments the message-passing programming model and is compatible with MPI so that the programmer can use both in the same program. The GA library handles the distribution of arrays across processes and recognizes that accessing local memory is faster than accessing remote memory. However, the library allows access mechanisms for any part of the entire distributed array regardless of where its data is located. Local memory is acquired via `NGA_Access()` returning a pointer while remote memory is retrieved via `NGA_Get()` filling an already allocated array buffer. GA has been leveraged in several large computational chemistry codes and has been shown to scale well.

CHAPTER THREE

PREVIOUS WORK

GAiN is similar in many ways to other parallel computation software packages. It attempts to leverage the best ideas for transparent, parallel processing found in current systems. The following packages provided insight into how GAiN was to be developed.

3.1 MITMatlab

MITMatlab[24, 33] provides a client-server model for interactive, large-scale scientific computation. It does so by providing a transparently parallel front-end through the popular MATLAB[25] numerical package and sends the parallel computations to its Parallel Problem Server workhorse. Separating the interactive, serial nature of MATLAB from the parallel computation server allows the user to leverage both of their strengths. This also allows much larger arrays to be operated over than is allowed by a single compute node. MITMatlab does not allow parallel programs to run without the bottleneck imposed by the client-server model because MATLAB is still run serially. Only the server is ever run in parallel.

3.2 PyTrilinos

Trilinos[23] consists of a suite of related solvers built on established libraries such as PETSc[4, 3, 5], Aztec[44], BLAS[12], LAPACK[1], and others and strives to make them more capable, robust and user friendly. PyTrilinos[39] takes this a step further by wrapping selected Trilinos packages in Python. This adds the convenience and capabilities of Python to many of the Trilinos features, such as its parallelism. PyTrilinos supplements the capabilities of SciPy[26] and NumPy[32] by providing a high degree of compatibility with those packages.

3.3 GAMMA: Global Arrays Meets MATLAB

GAMMA[34] provides a MATLAB binding to the Global Arrays toolkit, thus allowing for larger problem sizes and parallel computation. MATLAB provides an interactive interpreter, however to fully utilize GAMMA one must run within a parallel environment such as provided by MPI and a cluster of compute nodes. GAMMA was shown to scale well even within an interpreted environment like MATLAB.

3.4 IPython and distarray

IPython[35] provides an enhanced interactive Python shell as well as an architecture for interactive parallel computing. IPython supports practically all models of parallelism but more importantly in an interactive way. For instance, a single interactive Python shell could be controlling a parallel program running on a super computer. This is done by having a Python engine running on a remote machine which is able to receive Python commands.

distarray[18] is an experimental package for the IPython project. distarray uses IPython's architecture as well as MPI extensively in order to look and feel like NumPy's `ndarray`. Only the SPMD model of parallel computation is supported, unlike other parallel models supported directly by IPython. Further, the status of distarray is that of a proof of concept and not production ready.

3.5 GpuPy

A Graphics Process Unit (GPU) is a powerful parallel processor that is capable of more floating point calculations per second than a traditional CPU. However, GPUs are more difficult to program and require other special considerations such as copying data from main memory to the GPU's on-board memory in order for it to be processed, then copying the results back. The GpuPy[14, 13] Python extension package was developed to lessen these burdens by providing a NumPy-like interface for the GPU. Preliminary results demonstrate

considerable speedups for certain single-precision floating point operations.

3.6 pyGA

The Global Arrays toolkit was wrapped in Python for the 3.x series of GA by Robert Harrison[21]. It was written as a C extension to Python and only wrapped a subset of the complete GA functionality. It illustrated some important concepts such as the benefits of integration with NumPy and the difficulty of compiling GA on certain systems.

In pyGA, the local or remote portions of the global arrays were retrieved as NumPy arrays at which point they could be used as inputs to NumPy functions like the ufuncs. However, the burden was still on the programmer to understand the SPMD nature of the program. For example, when accessing the global array as an `ndarray`, the array shape and dimensions would match that of the local array maintained by the process calling the access function. Such an implementation is entirely correct, however there was no attempt to handle slicing at the global level as it is implemented in NumPy. In short, pyGA recognized the benefit of returning portions of the global array wrapped in a NumPy array, but it did not treat the global arrays as if they were themselves a subclass of the `ndarray`.

CHAPTER FOUR

DESIGN

There comes a point at which a single compute node does not have the resources necessary for executing a given problem. The need for parallel programming and running these programs on parallel architectures is obvious, however, efficiently programming for a parallel environment can be a daunting task. One area of research is to automatically parallelize otherwise serial programs and to do so with the least amount of user intervention.[7] GAiN attempts to do this for certain Python programs utilizing the NumPy module. It will be shown that some NumPy program can be parallelized in a nearly transparent way with GAiN and its multidimensional distributed array object `gainarray`.

4.1 Supporting Two User Communities

Both NumPy[32, 31] and Global Arrays[30, 29, 17] are well established in their respective communities. However, as stated in 1, NumPy is inherently serial. Also, the size of its arrays are limited by the resources of a single compute node. NumPy's computational capabilities may be efficient, however parallelizing them using the SPMD paradigm will allow for larger problem sizes and may also see performance gains. This design attempts to leverage the substantial work that is Global Arrays in support of large parallel array computation within the NumPy framework.

Python is known for among other things its ease of use, elegant syntax, and its interactive interpreter. Python users would expect these capabilities to remain intact for any extension written for it. As discussed in Section 3.4, the IPython project is a good example of supporting the interactive interpreter and parallel computation simultaneously[35]. Users familiar with NumPy would expect its syntax and semantics to remain intact if large parallel array computation were added to its feature set.

High performance computing users are familiar with writing codes that optimize every last bit of performance out of the system where they are are being run. Although message-passing is a useful and widely adopted computation model, Global Arrays users have come to appreciate the abstraction of a shared-memory interface to a distributed memory array. In either case, users are familiar with the challenges involved in maintaining scalability as problem sizes increase or as additional hardware is added. Maintaining these codes may be difficult if they are muddled with FORTRAN and/or C and various message-passing API calls. If one of these users were to switch to NumPy in order to leverage its strengths, they would hope to not sacrifice the performance and scalability they once may have enjoyed.

Not all NumPy programs are suitable for parallel execution. If the Python interpreter is to be used interactively, or if the NumPy program has made assumptions about a single process environment, there must be a separation between Python running as a single process and a parallel back-end where the parallel computation is performed. For example, the program might open a connection to a database. Doing so with multiple processes and in addition having each process make multiple updates to that database would be disastrous. Explicit knowledge on the user's part of the parallel nature of the execution environment would be needed to mitigate database access.

Mitigating those NumPy programs unsuitable for parallel execution can be handled by utilizing master/slave parallelism. The master NumPy program could run as usual until an expression was reached involving a `gainarray`. This expression would then be sent to the slave processes for execution. This strategy cleanly separates the serial from the parallel.

4.2 Global Arrays in Python

Robert Harrison's work to wrap a subset of Global Arrays functionality in the Python language was successful[21]. However, it was our goal to make GAiN a pure Python extension

module. Further, there may have been API incompatibilities between the 3.x release of GA and the current 4.x release series or the version of the NumPy C-API used. Rather than leverage the existing pyGA code, it was decided to leverage its best ideas and use `ctypes` to wrap what was needed from the Global Arrays toolkit.

Even though a compiled extension module generally performs faster than its pure Python counterpart, there were a number of benefits to writing at least the first version of GAiN as pure Python. First and foremost it would require very little to be installed by the end user. Second, it leverages Python's strengths such as its readability, maintainability, and the elimination of the need to compile machine code. It is beyond the scope of this thesis to compare the performance of `ctypes` to a compiled extension, however if the need arose it would only be a matter of time to write GAiN as a compiled C extension.

4.3 Whether to Subclass ndarray

Both NumPy and GA provide multidimensional arrays and implement element-wise or matrix operations, as well as linear algebra routines. Although they have a number of differences, the primary one is that NumPy programs run within a single address space. When manipulating `ndarray`s, the `ndarray`s in their entirety are being manipulated in a serial fashion. With Global Arrays, each process gets a subarray of the distributed array.

When translating from NumPy to Global Arrays, each process must translate NumPy calls into calls with respect to their local array portions. The first thought would be to simply have the `gainarray` subclass the `ndarray` and implement the appropriate methods by which the subclass could integrate with NumPy. However, there are a few cases where this would result in unwanted behavior.

One must be careful whether NumPy mechanisms may attempt to allocate the entire `ndarray` *per process*. There are a few ways in which `ndarray`s are created, either invoking the `ndarray` constructor directly or one of the many array creation functions.

It would be simple enough to replace all of the array creation functions and override the `__new__` operator within the `ndarray` subclass to appropriately handle the creation of our `gainarray`. However, the ufuncs present a challenge.

The `ndarray` methods used by the ufuncs are insufficient to handle the duality required by our `gainarray`. An instance of our subclass both represents the entire array as well as its local portion, if any. The problem arises in the case of inputs to binary ufuncs. Take, for example, a binary ufunc such as `add`. If a gainarray is input and an `ndarray` is input, how should this be handled? For maximum computational efficiency, only the subarray represented by the `gainarray` should be operated over per process. The shape of the `gainarray` is likely broadcastable with the shape of the `ndarray`, however, the shape of the subarray of the `gainarray` is certainly not so, not to mention broadcasting a subarray to be compatible with the other input is illogical. The other input must then be sliced to match that of the subarray.

The slicing of the other `ndarray` input to match that of the `gainarray`'s subarray must happen as part of the call to the binary ufunc. NumPy has no mechanism to manipulate the inputs to ufuncs. `__array__()` and `__array_wrap__()` only operate on the output array, if specified. Therefore, the ufuncs must be wrapped with functionality that handles making any `ndarray`s compatible with the subarrays of any `gainarray`s passed as arguments to ufuncs. Similarly, if two `gainarray`s are used as inputs to a binary ufunc, their subarrays relevant to the calculation at hand must be made to match. This is just one example where NumPy must be made to understand the distributed nature of the `gainarray`. Subclassing `ndarray` in this case would not provide sufficient means to handle the distributed `gainarray`.

The last case against subclassing `ndarray` is when the NumPy program is to be run in the master/slave configuration. In this configuration, when a `gainarray` is created, it must communicate to the slaves to also create the `gainarray` within their memory

space. If the gainarray subclassed the ndarray then it would override __new__(). To initialize the data members that are defined by the ndarray.__new__(), that class method would also need to be called. That call would attempt to allocate the memory for the ndarray. This would cause the distributed gainarray to be allocated both within the slaves as well as entirely on the master. There is no way to initialize the ndarray members without allocating the array's memory. Subclassing the ndarray would make it impossible to run GAiN in the master/slave configuration.

Rather than subclass the ndarray, then, we must create an ndarray work-alike replacement. Besides being impractical, such as in the master/slave configuration, there is no benefit to subclassing the ndarray since the majority of ndarray attributes would need to be replaced with GAiN-specific, distributed functionality. Further, not subclassing the ndarray ensures no memory will ever be allocated by any NumPy mechanisms inadvertently.

4.3.1 Utilizing both NumPy and Global Arrays

Regardless of whether gainarray subclasses ndarray, memory will be allocated by Global Arrays. This memory must be made available to NumPy as ndarrays. There are a few ways of doing this, either from within NumPy's C-API or from within Python. ndarrays hold a pointer to a memory location representing the beginning of the underlying C array. Using a pointer obtained from either NGA_Access() or NGA_Get() is relatively straightforward. Recall from 3.6 that pyGA used NumPy's C-API in order to create ndarrays in this way. However, NumPy does not expose those same array creation functions from within Python. Instead, a Python buffer object must be created first and then used as input to a special ndarray creation function. However, creating buffer objects from C pointers is only possible from Python's C-API. Thankfully, Python's ctypes module exposes Python's C-API, so we can use it to create our buffer object which can

then be passed to NumPy[22]. See the code sample in Appendix A.2 for details.

There is a fair bit of overlap between NumPy and Global Arrays functionality including certain element-wise operations and linear algebra support. In the cases where overlap occurs, we need to choose which library should implement the functionality. Using Global Arrays directly might be faster since there would be no overhead in creating the `ndarray`s from the Global Arrays data pointers. However, it was noticed in a set of test programs that the same program written using GA directly versus GAiN produced slightly different results due to different round-off error between the two implementations. Even though there are differences in the results, it might still be beneficial to use the Global Arrays methods if there is a significant speedup, such as with the matrix multiplication algorithm used by GA[27].

4.3.2 Slicing

GAiN would be incomplete if it didn't handle the slicing of `gainarray`s. NumPy extends the functionality of slicing in a number of ways, as noted earlier, to include single-element access, subarray slices, and indexing using other arrays. Each are valuable in their own right. All types of slicing operations utilize the same overloaded operator `__getitem__` and so are dispatched to appropriate handlers based on the arguments to this function.

Single-element access is the easiest to implement. Given a single index in each dimension, the value can simply be retrieved using GA's `NGA_Get()` without regard to where the actual data resides. This value can be directly returned.

Simple Slicing

Subarray slices, also known as "simple slicing" in NumPy, represent the more traditional form of slicing. The slice itself can be a single Python int, slice, Ellipsis, or None object or a sequence thereof. NumPy's C-API provides a convenient collection of routines for calculating the number of dimensions, shape, strides, and memory offset of a new `ndarray`

given an existing `ndarray` and the slicing argument. These functions are not exposed to Python, but their functionality is vital for maintaining the global view of the distributed arrays. This and other functionality found in NumPy's C-API but not in NumPy's Python interface must be ported to Python. Perhaps in the future these functions will be exposed to Python directly without having to create and slice an `ndarray` to get the same results.

Recall that simple slicing in NumPy produces new `ndarray`s that are views of their sliced originals. However, these views are still typed as `ndarray`s. Slicing in GAiN produces similar results. Slicing a `gainarray` will produce a new `gainarray` that shares its memory with its original. The resulting `gainarray` will have its own `ndim`, `shape`, and `strides` arguments and its `base` attribute will refer to the original `gainarray`.

Global Arrays maintains information on both the global properties of an array as well as its distribution across processes. Therefore, there will be one `gainarray` instance per process containing both information about the global array as well as that process's local distribution. The `gainarray` must behave as if it were an `ndarray`, so it's attributes `shape`, `strides`, `ndim`, etc. will refer to the global representation of the array. When a slicing operation occurs, the local distribution information does not change. It is only when the `gainarray` is to be used in an expression that the `ndarray` wrapper is produced. At that time, each process determines whether it holds a local portion of the sliced `gainarray`, returning immediately if they don't.

Slice Arithmetic

As slices are taken from the `gainarray` the underlying memory remains unchanged but the view onto that memory changes. If a view of a `gainarray` must be converted into an `ndarray`, we must be able to reconstruct an `ndarray` with the correct shape and data elements. Therefore, as slices are taken, the slicing operand is cached within the `gainarray`. If a slice of a view is taken, the cached operand is further sliced to reflect

the new view which we call *slice arithmetic*.

4.4 Master/Slave Parallelism

It was mentioned in the opening of this chapter that master/slave parallelism was needed for those NumPy programs unsuitable for parallel execution as well as for supporting interactive parallel programs via the interactive Python interpreter. The original program assumes the role of the master while the parallel portions of the program are sent to separate, parallel slaves. Unless the data being operated on already exists on the slaves, both data and the function to operate on the data must be transmitted. To accomplish this parallelism, we need to use the process management features of the MPI-2 specification as well as a custom `pickle` subclass.

4.4.1 Using MPI-2 and Spawn

MPI-2 added dynamic process management to the MPI specification. The function we will use is `MPI_Spawn` which allows one group of processes to create another group of processes. In our case the first group of processes will consist of the single master process. The number of slaves to spawn will be determined either as a parameter from the master's command-line invocation or from an external configuration file. `mpi4py` has an implementation of the spawn function.

4.4.2 Serializing Python Functions

Telling the slaves what actions to perform could be accomplished by establishing a command language. The commands would be sent from the master and then interpreted by the slaves. The language would only consist of a certain set of commands, likely pertaining to either NumPy or GA operations. This sort of command language might be needed for languages like C or FORTRAN that are not interpreted, however, since Python is already an interpreted language it is itself the command language that we need. All we need, then,

is a means of sending arbitrary objects and functions as our commands.

`mpi4py` makes this communication simpler since any `pickle`able object can be communicated. The `pickle` operation is usually performed automatically by `mpi4py`, if needed. Unfortunately, as stated earlier, a function's byte code is not pickled but rather its name only. If we assumed that pickling functions by name was acceptable, requiring the slaves to run the same versions of the Python libraries (and the Python interpreter itself,) then we might be okay. However, the NumPy array creation function `fromfunction` accepts any user-defined function as an argument. That user-defined function would need to be communicated to the slaves or else `fromfunction` would need to create the array on the master before sending it to the slaves. Further, that user-defined function might come from an interactive Python session as opposed to being written as part of a module that could be imported by a slave.

`pickle` may not be sufficient for our function pickling needs, however the `marshal` module is able to serialize Python byte code. Unfortunately, `marshal` does not share other desirable features of `pickle` such as tracking objects that have already been serialized so they won't be serialized again and support of more than just the built-in Python types.

Our solution was to subclass the `pickle` machinery to add function marshalling. Functions are like any other object in Python whose attributes can be inspected at runtime. The important attributes of Python's `FunctionType` include `func_globals`, `func_code`, and `func_defaults` which represent the objects within the function's scope, the byte code of the function, and the default arguments to the function, respectively. All three of those attributes are necessary and sufficient to reconstruct a function.

4.4.3 Communication Protocol

The communication between the master and the slaves consists of two messages. The first message is a one-to-many message from the master to the slaves consisting of the function

to execute and the functions arguments. The second message is a many-to-one message from the slaves to the master and consists of the function's returned values. If the slave encounters an exception (most likely some sort of evaluation error), then the message will contain the exception instance generated by the slave. Slaves may fail independently, so there may be one or more exceptions within the message. To be correct, the entire message must be scanned for exceptions.

Pickling a function for the master/slave configuration is not perfect, however. If the function being serialized is not a pure function, meaning its arguments are passed by reference and manipulated within the function but not returned, then the slaves will not communicate those arguments back to the master. Only the function's returned arguments will be communicated back to the master.

4.5 Implicit versus Explicit Parallelism

GAiN will ultimately operate as a stand-alone SPMD program (i.e. explicitly parallel) or as a master controlling an SPMD slave program (i.e. implicitly parallel.) The factor that decides which mode GAiN will be run under is whether GAiN is imported within a single process. The MPI environment can be queried to determine how many processes were requested for the current process. If only one process was requested, GAiN will run in the master/slave configuration. The user need not alter how GAiN is imported within their programs since GAiN will determine at run-time which mode is desired.

Whether GAiN is running explicitly or implicitly parallel, both modes will share a significant amount of code. For most of GAiN's functionality, the master simply needs to communicate one of its own functions to the slaves. Rather than implement two versions of GAiN that are practically identical, GAiN will be implemented as if it were only explicitly an SPMD program. Using Python's function decorators, GAiN's functionality will be altered at import-time with proxied versions of its functions.

GAiN	NumPy
gainint16	int16
gainint32	int32
gainint64	int64
gainfloat32	float32
gainfloat64	float64

Table 4.1: GAiN Data Types. These are the data types supported by GAiN and their NumPy equivalents.

4.6 Using GAiN

Some of the goals of GAiN are to improve performance over NumPy and to require little change to existing NumPy programs in order to use GAiN. For some NumPy programs, the only change necessary is to change `from numpy` to `from gain` or to change `import numpy` to `import gain as numpy`. Some programs may require specifying the `gain` data types instead of the NumPy ones to functions taking the `dtype` attribute. Table 4.1 shows the `gain` types and their NumPy equivalent.

CHAPTER FIVE

IMPLEMENTATION

The design discussed previously and its evaluation programs were implemented in approximately 10000 lines of purely Python code. The C versions of the evaluation programs are the only exceptions to the pure Python target. The majority of the design was implemented without undue effort. The exceptional cases are listed in more detail in the following sections.

5.1 Accessing Global Arrays in Python

pyGA[21] recognized that wrapping memory returned from Global Arrays within a NumPy `ndarray` was the appropriate course of action. The pyGA distribution includes some sample code demonstrating calling NumPy ufuncs on `ndarray`-wrapped GA memory. This is important functionality, however, rather than approach the problem from the perspective of having Global Arrays use NumPy, GAiN synergizes Global Arrays and NumPy.

Rather than continue or adapt pyGA to the needs of GAiN, it was this author's goal to make GAiN a pure Python extension module. Even though a compiled extension module should perform faster than its pure Python counterpart, there were a number of benefits to writing at least the first version of GAiN in pure Python. First and foremost it would require very little to be installed by the end user. Pure Python modules are by far the simplest to install. Second, it leverages Python's strengths such as its readability, maintainability, and the elimination of the need to compile machine code. It is beyond the scope of this thesis to compare the performance of `ctypes` to a compiled extension, however if the need arose it would only be a matter of time to write GAiN as a compiled C extension.

As with pyGA, only a subset of the complete GA API was ported. The subset of GA functionality wrapped using `ctypes` and bundled with GAiN could be used independently

MA_init	GA_Elem_minimum	NGA_Release
NGA_Access	GA_Fill	NGA_Release_update
GA_Add_constant	NGA_Get	GA_Scale
GA_Compare_distr	GA_Initialize	NGA_Scatter
GA_Copy	NGA_Inquire	NGA_Select_elem
NGA_Create	NGA_Locate_region	NGA_Strided_get
GA_Destroy	GA_Nnodes	NGA_Strided_put
NGA_Distribution	GA_Nodeid	GA_Sync
GA_Duplicate	NGA_Put	GA_Terminate

Table 5.1: Subset of Global Arrays C-API wrapped with Python's ctypes, in alphabetical order.

of GAiN and may serve as the beginnings of a complete GA implementation for Python. Functions were only wrapped on an as-needed basis until the subset in Table 5.1 was established. Historically, GA only supported two-dimensional arrays. The two different function prefixes `GA_` and `NGA_` are a result of maintaining backwards compatibility with the original API when the move was made to supporting arbitrarily-dimensioned arrays. Since GAiN's wrapper interface for GA need not be backwards compatible, the prefixes were stripped from the function names and the remaining part of the function name was made lowercase. This follows Python naming conventions. All wrapped GA functionality is now found in the module `ga`.

Although `ctypes` makes it possible to call C functions directly from within Python, it is not as simple as passing Python objects to C functions. The only native Python objects that are directly supported are `str` and `int`. Most GA function calls require one or more integer arrays. The GA functions that move data also require pointers to data arrays. Rather than force the caller of the GA functions to use the `ctypes` C data types for constructing function arguments, Python sequences are converted internally to the appropriate types. In an effort to minimize the overhead of Python-to-C type coercion, in functions returning values the `ctypes` types are returned rather than converted back into Python sequences.

5.2 gainarray.flat

The Performance Python[38] test program to be discussed in 6.2 makes heavy use of the `flat` attribute on the `gainarray`. For an `ndarray`, this attribute returns a 1-dimensional view of the given array as an `iterator` without copying data. This is no problem for the `ndarray` where all data for the array lives in a single address space. In that case, only a pointer to the beginning of the original array must be maintained as well as the strides necessary to move between adjacent array elements. The flat iterator may be passed to NumPy functions just like any other `ndarray`.

For `gainarray`s, the problem is much more difficult. Though it would be possible to create an `iterator` for a `gainarray`, passing one to one of GAiN's ufuncs would be problematic. GAiN's ufuncs expect to operate on their local piece of the input array. An `iterator` argument would need to be split appropriately among the many processes. Unfortunately, the memory that each process maintains is not 1-dimensionally contiguous. Instead, the low and high points defined by a subarray may span across dimensions discontiguously.

It was attempted to `flatten()` the `gainarray` rather than use the `iterator` approach, with just as poor results. Recall from the design that reshaping an array requires a copy of the data since no such operation is implemented by Global Arrays to redistribute or reshape an array. Each process must then `scatter()`s its portion of the original array into the flattened copy. This approach caused egregious communications overhead.

Recognizing that `flat` was used such that the `dot` function would operate element-wise on two 2-dimensional inputs, the test program was changed to instead call the appropriate `multiply` and `sum` functions. To be fair, this change was also made to the original laplace.py test program.

5.3 Implicit versus Explicit Parallelism

When running explicitly parallel, all N processes run the same program and therefore function calls behave as for a serial program. When running implicitly parallel, the functions are sent to the slaves but the result of executing the function must be reduced down to as single result to the master. If the functions that the implicitly parallel `gainarray` proxies return large objects, such as an `ndarray`, then it would be wasteful to send N identical results back to the master only to have $N - 1$ discarded. As a solution, the primary slave is the only slave that communicates actual results back to the master. The remaining slaves only communicate Python's `None` or an exception object, if one were to be raised by the function. The protocol initially designed had all slaves communicating their results back to the master, which was unnecessary.

CHAPTER SIX

EVALUATION

The success of GAiN hinges on its ability to enable distributed array processing in NumPy, to transparently enable this processing, and most importantly to efficiently accomplish those goals. This chapter describes two benchmark codes developed to test GAiN and reports on their performance. The programs were run on a homogeneous cluster of dual 3.2GHz P4 processors, 1GB main memory, using 1GBit Ethernet, TCP/IP socket communication, running Ubuntu 8.10. GAiN utilized 8 nodes of the cluster while NumPy and others ran serially on a single node (as they must.)

6.1 distmap

The idea for the distmap program originated from Ben Eitzen's work on transparently using GPUs within NumPy[13]. The program randomly distributes points on an $N \times N$ grid, then calculates the distance from any grid cell to its nearest point. When the results map is written as a gray image, it produces a pattern similar to soap bubbles. This algorithm is used, for example, as part of the level set method[40] and it is useful for demonstrating relative performance of the algorithm when implemented in various ways. In GpuPy, two versions of the distmap program were written, one using GpuPy and the other using NumPy. Similarly for GAiN, there is a NumPy and a GAiN version of the distmap program.

For the first test, an $N \times N$ grid size was used with N varying from 1000 to 5000 at intervals of 1000. The results appear in Figure 6.1 below.

GAiN was not only able to scale better than NumPy, it also was able to run a larger problem size than NumPy. At $N = 8000$, NumPy was unable to complete its task due to memory swap thrashing. Even if it weren't for the lack of memory, GAiN performed much faster.

6.2 Performance Python

Performance Python[38] "perfpy" was conceived to demonstrate the ways Python can be used for high performance computing. It evaluates NumPy and the relative performance of various Python extensions to NumPy including SciPy's weave (blitz and inline)[43], Pyrex[16], and f2py[36]. It represents an important benchmark by which any additional high performance numerical Python module should be measured. The original program `laplace.py` was modified by importing `gain` instead of `numpy` and then stripped of the additional test codes so that only `gain` remained. The latter modification makes no impact on the timing results since all tests are run independently but was necessary because `gain` is run on multiple processes while the original test suite is serial. Recall from Chapter 5, the original code was also modified to use `multiply` and `sum` instead of the `flat` attribute

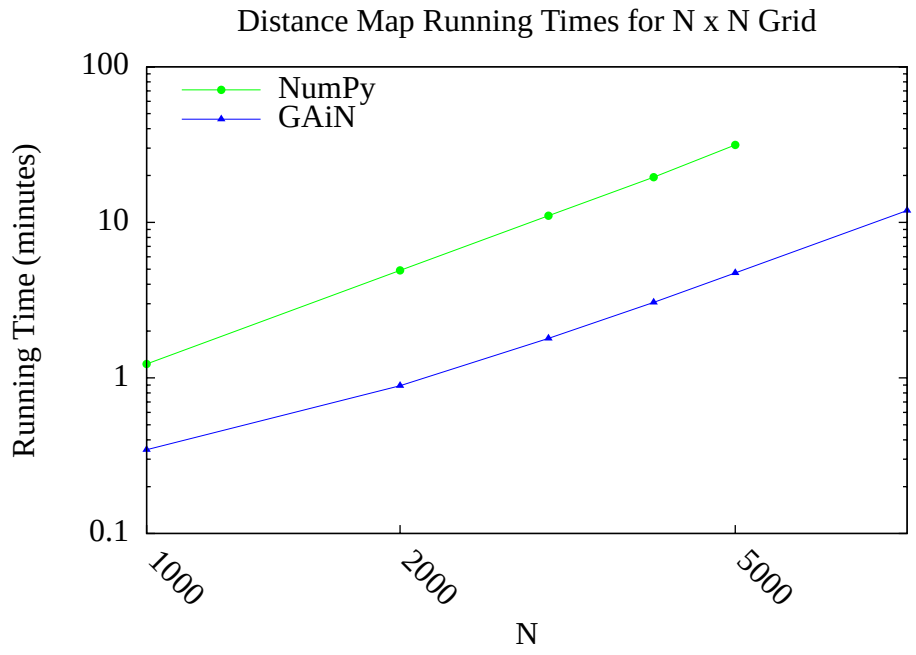

Figure 6.1: Distance Map Running Times for $N \times N$ Grid. This plot compares the running times of the GAiN and NumPy implementations of the distance map test. GAiN both scales better than NumPy and exceeds beyond the limits of NumPy's largets problem size.

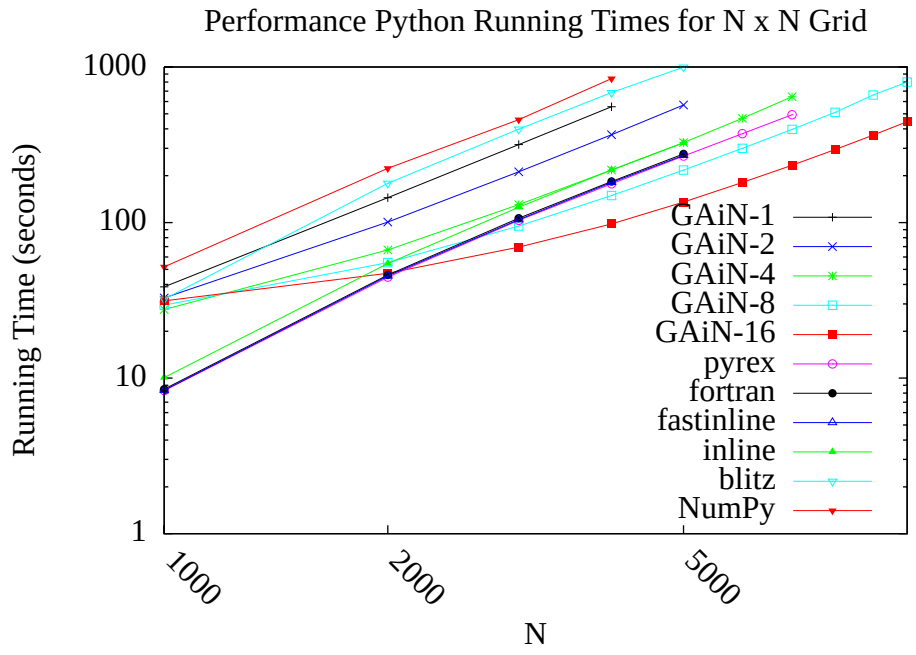

Figure 6.2: Performance Python Running Times for $N \times N$ Grid. This plot compares the running times of GAiN and various NumPy or Python extensions. GAiN was run using 1, 2, 4, 8, and 16 nodes. GAiN scales only as well as the best compiled implementation but does extend beyond the system resource limitations.

and the `dot` function. The results of running with the grid length N varying from 1000 to 5000 at intervals of 1000 appear in Figure 6.2. The data for Figure 6.2 can be found in Table 6.1.

Using 8 or more nodes, GAiN scaled better than NumPy and its related just-in-time compiled or pre-compiled codes. All codes appeared to scale uniformly with each other. GAiN was able to run much larger sizes of N while the other tests thrashed due to a lack of memory. The perfpy code represents in general a more realistic use case for GAiN whereas the distmap program is idealized with very little communication.

Length	GAiN-1	GAiN-2	GAiN-4	GAiN-8	GAiN-16	pyrex	fortran	fastinline	inline	blitz	NumPy
1000	38	32	27	29	31	8	8	8	10	32	51
2000	144	100	66	55	47	44	45	45	54	178	223
3000	317	211	130	94	69	103	106	104	125	398	459
4000	555	366	217	149	98	177	183	180	217	684	841
5000	?	570	326	217	135	267	276	270	325	997	?
6000	?	?	468	300	181	372	?	?	?	?	?
7000	?	?	644	397	233	493	?	?	?	?	?
8000	?	?	?	511	295	?	?	?	?	?	?
9000	?	?	?	660	364	?	?	?	?	?	?
10000	?	?	?	800	445	?	?	?	?	?	?

Table 6.1: Performance Python Results. GAiN was run using 1, 2, 4, 8, and 16 nodes. Times are in seconds. Missing values are represented by a “?” and indicate thrashing during the test.

6.3 Implicit versus Explicit Parallelism

Using 8 processors, in the master/slave configuration the distance map program took approximately 15 seconds for a small test case while the SPMD configuration took approximately 5 seconds. As expected, GAiN runs slower when implicitly parallel in the master/slave configuration than if it is run explicitly in SPMD mode. Profiling results show that the majority of the time is spent communicating between the master and the slaves. Even though an effort was made to reduce the number and size of messages communicated, it was not enough. Other communication protocols between the master and slaves must be devised or the current protocol must be optimized.

CHAPTER SEVEN

CONCLUSIONS

GAiN succeeds in its ability to grow problem sizes beyond a single compute node, however its performance in all cases does not scale as anticipated. In the case of the distance map test, scalability across nodes was achieved. The performance of the perfpy code leaves room for improvement. As described in 4.6 GAiN allows certain classes of existing NumPy programs to run using GAiN with sometimes as little effort as changing the import statement, immediately taking advantage of the ability to run in a cluster environment. Further, GAiN seamlessly allows parallel codes to be developed interactively. Once a smaller-sized program has been developed and tested on a desktop computer, it can then be run on a cluster with very little effort. GAiN provides the groundwork for large distributed multidimensional arrays within NumPy.

CHAPTER EIGHT

FUTURE WORK

GAiN is not a complete implementation of the NumPy API nor does it represent the only way in which distributed arrays can be achieved for NumPy. The following sections describe some of NumPy's important missing features within GAiN as well as discusses alternative implementations for GAiN that would be worth exploring further.

8.1 Missing Functionality

GAiN successfully implements all of the ufuncs, simple slicing, simple slice assignment, and many of the array creation functions. This is only a subset of the NumPy API and really a small fraction of NumPy's capabilities, many of which could benefit from similar auto-parallelizing treatment. Notable missing features include fancy slicing, binary ufunc special methods such as reduce and accumulate, and linear algebra. GAiN is certainly not complete. Unlike GpuPy, missing functionality cannot default to a built-in NumPy routine. This is due to the `gainarray` not subclassing the `ndarray`. NumPy simply would not know how to handle this type. Unless NumPy's ability to integrate with classes that do not subclass the `ndarray` is extended to support distributed arrays they will continue to need to exist as work-alike replacements to NumPy.

8.2 Linearization of the Underlying Multidimensional Global Array

The Global Arrays that make up the internals of the `gainarray` are multidimensional. This differs from NumPy, where the arrays are always represented in memory as contiguous memory segments. It was noted in a comparison of CoArray FORTRAN (CAF) and Unified Parallel C (UPC) [8] that UPC suffered from performance problems due to its linearization of multidimensional arrays, its synchronization model, and its communication

efficiency for strided data, among other issues. Considering the performance penalties associated with linearizing multidimensional arrays within UPC, it is still worth exploring whether GAiN would benefit from this one-dimensional approach since NumPy has used this model successfully since its inception. Global Arrays already exposes a 64-bit interface to allow for the large one-dimensional arrays that would be required if this design were attempted. Unfortunately, it would require a rewrite of much of GAiN since GAiN assumes the multidimensional nature of the underlying Global Arrays data.

8.3 C Implementation

Python is inherently slower than its C equivalent due to its interpreted nature. Writing the first version of GAiN in pure Python was easy for both development and accessibility for end-users, however it is likely a cause of some of the less significant performance problems. Since GAiN is intended for use in high performance computing situations, it would be possible to implement it as a C extension to further increase its performance.

BIBLIOGRAPHY

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. LAPACK Working Note 20, Department of Computer Science, University of Tennessee, Knoxville, Knoxville, TN 37996, USA, May 1990.

[2] David Ascher, Paul F. Dubois, Konrad Hinsen, Jim Hugunin, and Travis Oliphant. Numerical python. `http://numpy.scipy.org/numpydoc/numdoc.htm`, September 2001.

[3] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[4] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2001. http://www.mcs.anl.gov/petsc.

[5] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[6] Duncan Booth. Integrating python, c and c++. `http://www.suttoncourtenay.org.uk/duncan/accu/integratingpython.html`, 2003.

[7] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems, Vol. 1*. Prentice Hall PTR, 1 edition, May 1999.

[8] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, New York, NY, USA, 2005. ACM.

[9] Lisandro Dalcin, Rodrigo Paz, and Mario Storti. Mpi for python. *Journal of Parallel and Distributed Computing*, 2005.

[10] Lisandro Dalcin, Rodrigo Paz, and Mario Storti. Mpi for python documentation. `svncohttp://mpi4py.scipy.org/svn/mpi4py/mpi4py/trunkmpi4py`, 2007.

[11] Lisandro Dalcin, Rodrigo Paz, Mario Storti, and Jorge D'Elia. Mpi for python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, 2007.

[12] J. Dongarra. Basic linear algebra subprograms technical forum standard. *International Journal of High Performance Applications and Supercomputing*, 16(2):1–111.

[13] Benjamin Eitzen. Gpupy: Efficiently using a gpu with python. Master's thesis, Washington State University Tricities, Richland, WA, August 2007.

[14] Benjamin Eitzen and Robert R. Lewis. Gpupy: Transparently and efficiently using a gpu for numerical computation in python. *in preparation*, 2009.

[15] Dave Abrahams et al. Boost.python. `http://www.boost.org/doc/libs/1_38_0/libs/python/doc/index.html`.

[16] G. Ewing. Pyrex, a language for writing python extension modules. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex`, 2008.

[17] Global arrays webpage. `http://www.emsl.pnl.gov/docs/global/`.

[18] Brian Granger. Ipython distributed arrays. `https://code.launchpad.net/~ipython-dev/ipython/ipythondistarray`, July 2008.

[19] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface second edition*. MIT Press, November 1999.

[20] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[21] Robert J Harrison. Global arrays python interface 1.0. `http://www.emsl.pnl.gov/docs/global/pyGA/pyGA.html`.

[22] Mark Helsep and Travis Oliphant. Numpy array from ctypes pointer object? `http://projects.scipy.org/pipermail/numpy-discussion/2006-July/009438.html`, July 2006.

[23] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[24] Parry Husbands and Charles Lee Isbell Jr. The parallel problems server: A client-server model for interactive large scale scientific computation. In Palma et al. [33], pages 156–169.

[25] William Palm III. *A Concise Introduction to Matlab*. McGraw-Hill, 1st edition, October 2007.

[26] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[27] Manojkumar Krishnan and Jarek Nieplocha. A matrix multiplication algorithm suitable for clusters and scalable shared memory systems. Proceedings of IEEE International Conference on Parallel and Distributed Processing Symposium, 2004.

[28] Fredrik Lundh. *Python Standard Library*. O'Reilly Media, Inc., May 2001.

[29] Jarek Nieplocha, Manojkumar Krishnan, Bruce Palmer, Vinod Tipparaju, and Jialin Ju. The global arrays user's manual, 2006.

[30] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edo Apra. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[31] Travis E. Oliphant. Guide to numpy. `http://www.numpy.org`, December 2006.

[32] Travis E. Oliphant. Python for scientific computing. *Computing in Science and Engg.*, 9(3):10–20, 2007.

[33] José M. Laginha M. Palma, Jack Dongarra, and Vicente Hernández, editors. *Vector and Parallel Processing - VECPAR '98, Third International Conference, Porto, Portugal, June 21-23, 1998, Selected Papers and Invited Talks*, volume 1573 of *Lecture Notes in Computer Science*. Springer, 1999.

[34] Rajkiran Panuganti, Muthu Manikandan Baskaran, David E. Hudak, Ashok Krishnamurthy, Jarek Nieplocha, Atanas Rountev, and P. Sadayappan. Gamma: Global arrays meets matlab. `ftp://ftp.cse.ohio-state.edu/pub/tech-report/2006/TR15.pdf`.

[35] Fernando Perez and Brian E. Granger. Ipython: A system for interactive scientific computing. *Computing in Science and Engg.*, 9(3):21–29, 2007.

[36] Pearu Peterson. F2py users guide and reference manual. `http://cens.ioc.ee/projects/f2py2e/usersguide/f2py_usersguide.pdf`, January 2005.

[37] David Mertz PhD. Charming python: Decorators make magic easy. `http://www.ibm.com/developerworks/linux/library/l-cpdecor.html`, December 2006.

[38] Prabhu Ramachandran. Performance python. `http://www.scipy.org/PerformancePython`, May 2008.

[39] Marzio Sala, W. F. Spotz, and M. A. Heroux. Pytrilinos: High-performance distributed-memory solvers for python. *ACM Trans. Math. Softw.*, 34(2):1–33, 2008.

[40] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science (Cambridge ... on Applied and Computational Mathematics)*. Cambridge University Press, June 1999.

[41] Rajeev Thakur, Ewing Lusk, and William Gropp. Users guide for romio: A high-performance, portable mpi-io implementation, May 2004.

[42] TODO. Todo. TODO, TODO TODO.

[43] Mike Trumpis. Weave. `http://www.scipy.org/Weave`, 2008.

[44] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid. Official aztec user's guide: Version 2.1, December 1999.

[45] Guido van Rossum. Extending and embedding the python interpreter. `http://docs.python.org/ext/ext.html`, September 2006.

[46] Guido van Rossum. Python 2.6.1 documentation. `http://docs.python.org`, March 2009.

APPENDIX

APPENDIX ONE
SOURCE CODE

A.1 Python Function Decorators

```
def enhance(func):
    def new(*args, **kwargs):
        print "I am enhanced"
        return func(*args, **kwargs)
    return new

def old_style_decoration(a,b,c):
    return a,b,c
old_style_decoration = enhance(old_style_decoration)

@enhance
def new_style_decoration(a,b,c):
    return a,b,c
```

A.2 Wrapping Global Array Pointers with an ndarray

```
#!/usr/bin/env python

from ctypes import *
import gain.ga as ga
import numpy as numpy

shape = [2,2]

# create our global array and get its distribution
# single processes will hold the entire distribution
# e.g. here lo=[0,0] hi=[1,1]
g_a = ga.create(ga.C_FLOAT, shape)
lo,hi = ga.distribution(g_a)
ptr,ld = ga.access(g_a, lo, hi)

# calculate the size of the local portion of the distribution,
# if we have multiple processes
diffs = map(lambda x,y: y-x+1, lo, hi)
def safe_product(x,y):
    if x == 0: x = 1
    if y == 0: y = 1
    return x*y
nelements = reduce(safe_product, diffs)

# set up the python function that creates buffers,
# from ctypes.pythonapi
func = pythonapi.PyBuffer_FromReadWriteMemory
func.restype = py_object

# ptr is returned as a c_types pointer to the actual type
# of the global array. This differs from the GA C API which
# normally returns a void pointer
buffer = func(ptr, nelements * sizeof(ptr))

# create the numpy array, wrapping our global array pointer
# note that without specifying a shape, we get a 1-dimensional array
a = numpy.frombuffer(buffer, numpy.float32)

# now return control of the global array memory
# should probably get rid of temporary numpy array so that
```

```
# we don’t try to use it later
del a
ga.release_update(g_a, lo, hi)
```