Co-array Python:
A Parallel Extension to the Python Language

Craig E. Rasmussen[1], Matthew J. Sottile[1], Jarek Nieplocha[2],
Robert W. Numrich[3], and Eric Jones[4]

[1] Los Alamos National Laboratory, Los Alamos, NM 87545, USA
[2] Pacific Northwest National Laboratory
[3] Minnesota Supercomputing Institute, University of Minnesota,
Minneapolis, MN 55455
[4] Enthought, Inc.

Abstract. A parallel extension to the Python language is introduced
that is modeled after the Co-Array Fortran extensions to Fortran 95. A
new Python module, `CoArray`, has been developed to provide co-array
syntax that allows a Python programmer to address co-array data on
a remote processor. An example of Jacobi iteration using the `CoArray`
module is shown and corresponding performance results are presented.

1 Introduction

There have been several attempts at extending programming languages for use in
a parallel processing environment. These language extensions include, but are not
limited to, C* [1], Unified Parallel C (UPC) [2], Titanium (a Java extension) [3],
F^{--} (a Fortran 77 extension) [4], High Performance Fortran (HPF) [5], and Co-
Array Fortran (CAF) [6]. The existence of these parallel extensions indicate the
popularity of the idea, but as yet, none have had the nearly universal acceptance
of MPI [7] as a parallel programming model.

The Python language [8] provides a rapid prototyping environment and sev-
eral extension modules (mostly serial) have been created for the scientific com-
munity [9]. A notable Python feature is that many elements of the language can
be modified and extended, allowing tool developers to use existing syntactic ele-
ments to express new, more sophisticated operations. Furthermore, Python can
be extended with "plugin" modules implementing these new linguistic features,
rather than requiring a new or modified compiler. Portability is dictated by the
portable implementation of these specific modules, unlike the significantly larger
compiler suites for earlier parallel languages and extensions.

In this research note, we present a parallel Python extension module,
`CoArray`, as a way of providing co-array notation and functionality to Python
programmers. A co-array is a local data structure existing on all processors ex-
ecuting in a Single Program Multiple Data (SPMD) fashion. Data elements on
non-local processors can be referenced via an extra-array dimension, called the
co-dimension.

In this implementation, the Aggregate Remote Memory Copy (ARMCI) library [10] is used to facilitate memory transfer between co-array elements existing in memory on remote processors. ARMCI has also been used to implement the Global Array library, GPSHMEM – a portable version of Cray SHMEM library, and the portable Co-Array Fortran compiler from Rice University [11]. ARMCI provides simpler progress rules and a less synchronous model of RMA than MPI-2.

2   Co-array Python Implementation

The `CoArray` Python module implements co-arrays as an extension to Numeric Python [9], though other Python numerical array libraries could be used. The Numeric third-party module provides higher performance than do standard Python arrays. `CoArray` extends the Python array syntax by adding a set of parentheses (the co-dimension) to the standard array notation. For example, consider a two dimensional array A and a load operation from A into the scalar, $c = A[i, j]$ from row i and column j. If T is a co-array, then a similar load operation would be expressed as $c = T(k)[i, j]$, except in this case, the scalar is loaded with data from row i and column j on *processor* k, not necessarily the local processor. Note that the co-dimension k appears within parentheses to distinguish it from the normal Python array indices appearing within square brackets.

While this example illustrates load and store operations for a scalar, the truly powerful features are slicing operations. Using Python slices, one can transfer entire regions of a co-array in a single Python statement. For example, the statement $T(0)[-1, :] = T[0, :]$ puts the entire first row from T to the last row (index of -1) of T on processor 0. Assuming that the local processor index is 1, this could also have been written as $T(0)[-1, :] = T(1)[0, :]$.

The `CoArray` module uses the ARMCI library to transfer data (although nothing prohibits it from being implemented on top of other data-transport layers). ARMCI provides general-purpose, efficient, and widely portable remote memory access (RMA) operations (one-sided communication). ARMCI operations are optimized for contiguous and noncontiguous (strided, scatter/gather, I/O vector) data transfers. It also exploits native network communication interfaces and system resources such as shared memory [12]. Because very little processing occurs in Python, and because no extra copies of data are made (memory for the Numeric arrays are actually allocated by ARMCI), memory transfer operations using the `CoArray` module are roughly comparable to a C implementation, as will be shown in the example in next section.

It should also be noted that the local portion of a co-array can easily be shared with C as pointers to local co-array data are readily available. In addition, using the Chasm array descriptor library [13], one can assign local co-array data to an assumed shape, Fortran 90 array pointer.

2.1   Implementation Details

The key to the parallel `CoArray` module implementation is the Python `__call__`, `__setitem__` and `__getitem__` methods. The `__call__` method is invoked by the interpreter when the co-dimension is selected (using parenthesis notation) and returns the local Numeric array if the co-dimension index is local or a proxy to the remote co-array otherwise. The `__setitem__` method is invoked when the normal array dimensions are selected (using square bracket notation) on the left side of an assignment statement. When `__setitem__` is called on a remote proxy, the ARMCI library is used to put data to the remote co-array. Likewise, when `__getitem__` is called on a remote proxy, the ARMCI library is used to get data from the remote array. Otherwise, the `__getitem__` and `__setitem__` methods are forwarded to the local Numeric array.

3   Co-array Python Example

To illustrate the simplicity and expressive power of Co-Array Python syntax, we consider the two-dimensional Laplace equation on a square of size (`M` × `M`). We cut the square into horizontal strips by dividing the first dimension by the number of processors `N=M/nProcs`. Each processor allocates its own co-array,

```
T = coarray((N+2,M+2), Numeric.Float)
```

which contains a local strip with a halo of width one. The halo values on the boundary of the global square enforce Derichlet boundary conditions, but the halo values on interior boundaries, row boundaries in our case, must be updated with data from neighboring strips after each iteration.

The following variables are defined:

```
nProcs = mpi.size
me = mpi.rank

up = me - 1
dn = me + 1
if me == 0: up = None
if me == nProcs - 1: dn = None
```

where `me` is the local processor index and `up` and `dn` are the neighboring processor indices on which the logically up and down array strips are allocated. As can be seen by the use of the pyMPI [14] module elements, `mpi.size` and `mpi.rank`, the `CoArray` module has been designed to be used with MPI.

The inner, iterative Python loop, executed in SPMD fashion by each processor, is:

```
1 """ update interior values """
2 T[1:-1,1:-1] = ( T[0:-2,1:-1] + T[2:,1:-1] +
3                  T[1:-1,0:-2] + T[1:-1,2:] ) / 4.0
```

```
4 """ exchange boundary conditions """
5 mpi.barrier()
6 if up != None: T(up)[-1,:] = T[ 1,:]
7 if dn != None: T(dn)[ 0,:] = T[-2,:]
8 mpi.barrier()
```

Each processor replaces each value in its local strip by the average of the four surrounding values. Standard Python syntax allows us to represent this averaging in very compact notation (lines 2-3, above). The first two terms on the right side represent values up and down and the last two terms represent values left and right. This code corresponds to a true Jacobi iteration because the Numeric Python array module computes the entire result on the right side of the statement before storing the result to the left side. A Jacobi iteration written in C requires two arrays to avoid polluting the new solution with partially updated values. Two arrays are not needed in Python because temporary arrays are created as binary expressions on the right side of the equation are evaluated.

No communication between processors takes place during the averaging. It is all local computation. To update the halos after averaging, we need two barriers. The first barrier guarantees that all processors have finished computing their average using the old values before any processor updates the halos with new values. The second barrier guarantees that all processors have finished updating halos before any processor performs the next average using the new values.

We also coded the example in Python, using the pyMPI module, and in C with MPI to compare with Co-Array Python. We ran all three versions on a dual processor Macintosh G4 (1 GHz, 1.5 Gbyte) using Mac OS X version 10.3.2 and Python version 2.3. We used LAM/MPI version 7.0.4 for the C and Python code and ARMCI version 1.1 for the Co-Array Python code.

Table 1 shows timing results. Note that communication times for Co-Array Python are much shorter than those for pyMPI due to the need for Python to serialize (pickle) every message before sending it. This requires a heap allocation, a copy, and additional processing [14]. Co-Array Python is able to send data with no extra memory copies and communication times are roughly two times those of the C version.

Code complexity is an important metric for evaluating programming models. The Co-Array Python code requires less than half the number of statements,

Table 1. Timing data (seconds) for the Co-Array Python (CoP), MPI Python (PyMPI), and C (C) versions. Data are an average of 5 runs, each for 40 iterations. The Python MPI version failed to complete the 2048x2048 run.

Size	CoP_{comm}	CoP_{total}	$PyMPI_{comm}$	$PyMPI_{total}$	C_{comm}	C_{total}
128x128	0.017	0.33	0.07	0.38	0.013	0.05
256x256	0.023	1.28	0.13	1.41	0.015	0.14
512x512	0.041	6.28	0.28	6.47	0.020	0.55
1024x1024	0.068	28.4	0.52	28.78	0.032	2.49
2048x2048	0.089	113.5			0.047	10.13

six versus thirteen, to represent data transfer compared with the pyMPI code. The additional code in the MPI version was required to avoid synchronization deadlock.

Although not encountered in this simple example, the C MPI version would require an additional level of complexity, if data were partitioned across processors by blocks, rather than by strips. This would require the transfer of noncontiguous halo data along columns by the use of an `MPI_Type_vector` to specify strides. The Co-Array Python module transfers data stored in noncontiguous memory transparently, as the module provides the necessary stride information to the ARMCI library, rather than placing the burden on the user. This is also true of the pyMPI version, although it would suffer the same performance penalties discussed above.

4   Future Work

This research note describes an implementation of co-arrays in Python. The co-array syntax provides a concise mechanism for implementing parallel applications while hiding the underlying communication details, making it ideal for rapid prototyping. While computation was dominant in our simple example, other algorithms exist where communication is more of a concern. Therefore, we will pursue performance optimizations in the implementation of the `CoArray` module to take advantage of asynchronous transfers to provide overlap with computation.

Performance may be improved by changing when a data transfer occurs relative to when it is posted. An eager evaluation approach, which is currently implemented, forces immediate transfer. Varying degrees of laziness in this evaluation could defer the actual transfer until a later time, possibly eliminating them altogether if the data are not used. A lazy implementation of this extension could take advantage of data locality of several posted communication requests to aggregate multiple small communication requests into larger, coarser grained messages that perform well on modern communication hardware.

Finally, we would like to implement the `CoArray` module in C for performance and on top of other communications libraries to broaden the number of users who can take advantage of this library, with particular attention to compatibility with extensions and tools included as part of the Scientific Python (SciPy) distribution [9].

Acknowledgments

References

1. Thinking Machines Corporation, Cambridge, Massachusetts: C* Language Reference Manual (1991).
2. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, 17100 Science Drive, Bowie, MD 20715 (1999) http://www.super.org/upc/
3. Yelick, K., Semenzato, L., Pike, G., on Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance Java dialect. Concurrency: Practice and Experience **10** (1998) 825–836.
4. Numrich, R.W.: A parallel extension to Cray Fortran. Scientific Programming **6** (1997) 275–284.
5. Koebel, C.H., Loveman, D.B., Schrieber, R.S., Steele, G.L., Zosel, M.E.: The High Performance Fortran Handbook. The MIT Press, Cambridge, Massachusetts (1994).
6. Numrich, R.W., Reid, J.K.: Co-Array Fortran for parallel programming. ACM Fortran Forum **17** (1998) 1–31 http://www.co-array.org/
7. Gropp, W., Lusk, E., Skjellum, A.: Using MPI, portable parallel programming with the Message-Passing Interface. The MIT Press, Cambridge, Massachusetts (1994).
8. van Rossum, G., Drake, F. L., Jr. (ed): Python Reference Manual. (2003) http://www.python.org/
9. Scientific Python web site (2004) http://www.scipy.org/
10. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In: Proceedings of RTSPP/IPPS99 (1999).
11. Coarfa, C., Dotsenko, Y., Eckhardt, J., Mellor-Crummey, J.: Co-array Fortran performance and potential: An NPB experimental study. In: Proceedings of LCPC 2003 (2003).
12. Nieplocha, J., Ju, J., Straatsma, T.P.: A multiprotocol communication support for the global address space programming model on the IBM SP. In: Proceedings of EuroPar 2000 (2000).
13. Chasm language interoperability web site (2004) http://chasm-interop.sf.net/
14. Miller, P.: pyMPI – An introduction to parallel Python using MPI (2002) http://pympi.sourceforge.net/