Computing and Informatics, Vol. 27, 2008, 249–259

A SCALABLE INTERACTIVE PARALLEL COMPUTING ENVIRONMENT FOR PYTHON

Sudarshan Raghunathan

Interactive Supercomputing, Inc.
135 Beaver Street
Waltham, MA 02452, USA
e-mail: `rsudarshan@interactivesupercomputing.com`

Abstract. Modern open source high-level languages such as R and Python are increasingly playing an important role in increasing programmer productivity when programming high-performance computers. In this article, we describe Python Star-P, a high-level interactive parallel programming environment in Python. We discuss the architecture of the environment and the programming model along with a number of examples. We also describe the performance of the examples on a cluster of multi-core machines. Finally, we compare our environment with that of other existing parallel computing tools for Python and describe the advantages of our model over others.

1 INTRODUCTION

Star-P is a parallel computing environment that bridges high-level "desktop-oriented" languages such as MATLAB®, Python and R to high-performance computers. While many of these environments already have bindings to communication libraries such as Twisted or MPI, one of the key advantages of Star-P is that a user can parallelize existing serial applications written in very high-level languages without having to worry about thorny issues in traditional parallel programming such as data distributions, task allocation, load balancing, message passing and synchronization.

The services for parallel programming in Star-P are achieved via three mechanisms: a partitioned global address space containing primitives for indexing and operating on distributed objects, interfaces to a number of existing high-performance libraries for handling dense and sparse linear algebra, signal processing and parallel I/O operations, and finally a mechanism for users to plug in legacy serial and parallel libraries written in languages such as C, C++ and Fortran.

1.1 Outline

We will first provide a brief overview of the Star-P architecture followed by a description of the parallel programming model (such as the model for creating and operating on distributed arrays and porting existing serial code to run in parallel) under Python Star-P. Then, we describe the implementation of a few applications in both serial and parallel, and present performance results on running the benchmarks on clusters of multi-core machines. Finally, we describe a few other parallel programming environments in Python and describe how Star-P resembles and differs from these.

2 THE STAR-P ARCHITECTURE

As illustrated in Figure 1, Star-P is architected as a client-server system: the client interfaces with the user's high-level desktop computing environment and the server interfaces with the high-performance computer and provides language-neutral services for managing and operating on global objects. The user can continue to use the existing desktop environment to prototype his or her application on small-sized problems and use Star-P to transparently migrate the application to the supercomputer. At this point, the Star-P client intercepts function calls in the serial application and dispatches them to the server for parallel execution. The Star-P server in turn provides a rich set of optimized primitive operations on large data sets and supports multiple parallel modes of execution such as data and task parallelism.

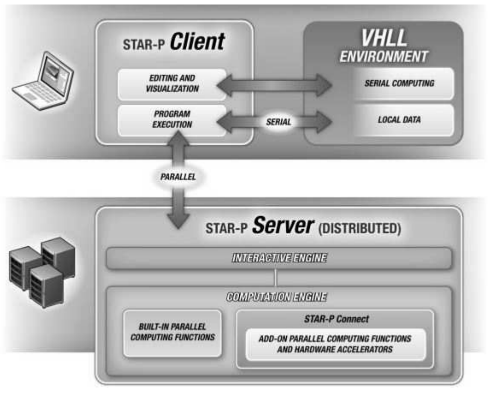

STAR-P Client
VHLL ENVIRONMENT
EDITING AND VISUALIZATION
SERIAL COMPUTING
PROGRAM EXECUTION
SERIAL
LOCAL DATA
PARALLEL
STAR-P Server (DISTRIBUTED)
INTERACTIVE ENGINE
COMPUTATION ENGINE
STAR-P Connect
BUILT-IN PARALLEL COMPUTING FUNCTIONS
ADD-ON PARALLEL COMPUTING FUNCTIONS AND HARDWARE ACCELERATORS

Fig. 1. Client-Server architecture of the Star-P platform

At present, there exist Star-P client interfaces to three high-level languages: MATLAB® (or M), Python and R. An important point to note is that even though Star-P supports multiple languages, it does *not* impose a common style of programming across the languages and instead naturally adapts to whatever client environment it is plugged into. For example, the syntax and semantics for operations on distributed matrices in M is inherently different from the corresponding syntax and semantics under Python and R.

2.1 Python and Python Star-P

Python [1] is a high-level language created by Guido Van Russom. Natively, Python does not have data types and containers such as matrices and lacks linear algebra and signal processing functions. Instead, these are added to the language through Python extension modules. Currently, the *de facto* Python module for numerical computing is NumPy [2] authored by Travis Oliphant and others.

The Star-P package in Python is an extension module that can be imported into an existing Python installation just like any of the modules in the Python standard library. The syntax and semantics in Star-P Python closely model those in NumPy. In the next few sections, we briefly describe the parallel programming model (distributions and how they propagate for data parallel operations as well as a mechanism for executing task parallel loops).

2.1.1 Parallel Programming Model in Python Star-P

The basic premise of the model is to maintain compatibility in syntax with serial codes written using the NumPy module. In most cases, the user must not be burdened with having to think "in parallel", keep track of distributions or worry about which portions of the code runs in serial and which in parallel. This allows users with a large existing serial application to port it to run in parallel with the least amount of effort.

A user can start using the Star-P Python simply by importing the Star-P module (using `import starp`). Most of the commonly used matrix constructors and operations available in NumPy are then available to the user. For example, to create a 100 element one dimension vector in NumPy, one would use:

```
import numpy
x = numpy.random.rand(100,)
```

whereas to create a one-dimensional distributed vector in Star-P, one would use:

```
import starp
X = starp.numpy.random.rand(100,)
```

or more succinctly as

```
import starp.numpy
X = numpy.random.rand(100,)
```

Many of the sub-packages in NumPy (such as `linalg` for linear algebra operations and `fft` for signal processing functions) and operators (element-wise unary and binary operators, reduction operators and indexing) are also available in Star-P and work in the same manner as their serial counterparts.

For example, in the previous example, to compute the maximum element, one would use `m = numpy.max(x)` in NumPy vs. `M = starp.numpy.max(X)` in Star-P Python.

In fact, for many simple applications, parallelizing is a matter of replacing `import numpy` with `import starp, starp.numpy as numpy` and adding a statement to connect to the server.

Although the syntax and semantics of Star-P Python closely model NumPy, the Star-P module provides addition functionality not available in NumPy. Chief among these are the facilities for transferring data back and forth from the client to the server, parallelizing loops with no loop-carried dependencies (the Star-P `PPEVAL` construct) and commands for parallel I/O which are discussed in more detail later.

2.1.2 Creation and Propagation of Parallelism

One of the key concepts in Star-P is the notion that parallelism is "infectious": once a distributed object is created, all subsequent operations involving it run in parallel and produce other distributed objects; the only exceptions to this rule are when the result is too small to operate efficiently in parallel and when the user explicitly destroys parallelism by converting the distributed object into a non-distributed one.

In Star-P Python, distributed inputs are created using the constructors modeled after their NumPy equivalents. By default, a distributed matrix is divvied up evenly across the cores on the HPC server along its last non-singleton dimension, but this can be explicitly over-ridden by the user. As mentioned before, operations on distributed matrices (even when some of the operands are non-distributed) run in parallel and produce other distributed matrices.

2.1.3 Performing Embarrassingly Parallel Computations

Star-P provides a convenient higher order function `PPEVAL` to apply a function in parallel to slices of a distributed multi-dimensional array. This mechanism is most effectively used to parallelize loops with no dependencies between the iterations.

To see how loops without any carried dependencies can be parallelized, consider the following code segment that computes the singular values of K $M \times N$ matrices in NumPy:

```
x = numpy.random.rand(M, N, K)
y = numpy.zeros(numpy.min((M, N)), K)

for i in xrange(K):
   y[...,i] = numpy.linalg.svd(x[...,i])[1]
```

The loop iterations are completely independent and can be executed in parallel using the `PPEVAL` construct as follows:

```
Y = starp.ppeval(numpy.linalg.svd, x)[1]
```

The result array `Y` is of size $\min(M, N) \times K$, each column of which contains the singular values of the corresponding $M \times N$ slice of the array x.

By default, each of the arguments to the function being evaluated is split among the iterations along the last non-singleton dimension (for example, two-dimensional arrays are divided by columns, three-dimensional arrays are divided along the third dimension, etc.). If the iterations need to be performed along a different dimension, the dimension can be explicitly specified using the `PPSPLIT` command. Moreover, read-only objects that must be available in their entirety to each iteration in the loop can be tagged using the `PPBCAST` directive.

The outputs from the invocation of a `PPEVAL` call are handled as follows: If each iteration of the function being evaluated produces outputs of the same shape then the outputs from all iterations are concatenated along the last non-singleton dimension. However, if the outputs are of different sizes then the resulting object is an opaque heterogeneous container. In keeping with the convention that distributed objects on the server are never transferred to the client without explicit user input, the outputs in either case are always distributed.

The main limitation of the `PPEVAL` construct is that when parallelizing loops, the body must currently be hoisted into its own function; we are currently exploring the possibility of automatically (or semi-automatically) transforming a loop with no carried dependencies to run in parallel. Another limitation is that currently the input arguments being split between the iterations are restricted to being dense multi-dimensional arrays of double precision and complex double precision values.

3 APPLICATION EXAMPLES

In this section, we describe a few examples of how Star-P Python can be used to implement or parallelize existing serial applications. In particular, we consider the implementation of the HPC Challenge Class II benchmarks [6] using our environment and demonstrate the performance on a large cluster of multicore machines.

3.1 Serial vs. Parallel Implementations

The benchmarks implemented using the Star-P environment can be run in both serial and parallel as follows: The main method of the implementation accepts a parameter for the number of processors to use on the server. If the number of processes is set to 0, the NumPy module is imported and the entire benchmark runs locally on the client; when the number of processes is non-zero, the Star-P Python module is imported instead and the entire benchmark runs on the HPC server.

3.2 The Execution Framework

The main measurement routine is the following higher-function that runs a function to be timed multiple times and returns the total time for the runs:

```
def iterate_func(nr, func, *args):
  """
  Execute a function, func, nr times and return the result along with
  the total time taken for all the iterations
  """
  t0 = time.time();
  for i in xrange(0, nr):
    out = func(*args)
  return (out, time.time() - t0)
```

3.3 High-Performance LINPACK

The High-Performance LINPACK benchmark measures the performance of solving a dense linear system using Gauss Elimination with partial pivoting. Our implementation of the benchmark creates an $N \times N$ matrix, a length N vector and calls the framework with the underlying linear solve function (using LAPACK in the case of serial computations and custom parallel solver when running under Star-P).

```
def run_hpl(n, nr, tol=16):
  """
  Run the High-performance LINPACK test on a matrix of size n x n, nr
  number of times and ensures that the the maximum of the three
  residuals is strictly less than the prescribed tolerance (defaults
  to 16).

  This function returns the performance in GFlops/Sec.
  """
  a = numpy.random.rand(n, n);
  b = numpy.random.rand(n, 1);
  x,t = iterate_func(nr, numpy.linalg.solve, a, b)

  r  = numpy.dot(a, x)-b
  r0 = numpy.linalg.norm(r, inf)
  r1 = r0/(eps * numpy.linalg.norm(a, 1) * n)
  r2 = r0/(eps * numpy.linalg.norm(a, inf) *
    numpy.linalg.norm(x, inf) * n)

  performance  = (1e-9 * (2.0/3.0 * n * n * n + 3.0/2.0 * n * n) *
    nr/t)
  verified     = numpy.max((r0, r1, r2)) < 16

  if not verified:
    raise RuntimeError, "Solution did not meet the prescribed
```

```
      tolerance %d"%tol

  return performance
```

3.4 Fast Fourier Transform

The FFT benchmark creates a one-dimensional vector A of length N and calls the underlying FFT function in NumPy when running in serial (which in turn calls FFTPACK) or Star-P (which calls optimized parallel FFT routines). The driver also verifies that the result of calling the inverse FFT returns back the original vector within machine precision. The function then returns the performance of the FFT function in Star-P in GFlops.

```
def run_fft(n, nr, tol=16):
  """
  Run the one-dimensional FFT benchmark on a vector of size n, nr
  number of times and verifies that the inverse transforms recreates
  the original vector upto a tolerance, tol (defaults to 16).

  This function returns the performance in GFlops/sec.
  """
  a = numpy.random.rand(n,1)
  b, t = iterate_func(nr, fft.fft, a)

  log2n = math.log(n)/math.log(2)
  performance = 1e-9 * 5.0 * n * log2n * nr/t
  verified    = numpy.linalg.norm(a - (numpy.fft.ifft(b))) /
    (eps * log2n) < tol

  if not verified:
    raise RuntimeError, "Solution did not meet the tolerance %d"%tol

  return performance
```

3.5 Embarrasingly Parallel Stream Addition

The stream benchmark creates a scalar s and two random vectors, A and B of length N and times the computation of the SAXPY expression, $C = s \times A + B$. Our implementation first creates the two vectors and the scalar. An anonymous function representing the SAXPY computation along with the input vector is then passed to the framework function to measure the performance over multiple runs.

```
def run_epstream(n, nr):
  """
```

```
Run the embarrasingly parallel stream benchmark on vectors of size
n, nr number of times.

This function returns the performance of the benchmark in
GFlops/second.
"""
s = numpy.random.rand(1);
a = numpy.random.rand(n);
b = numpy.random.rand(n);
c,t = iterate_func(nr, lambda s, a, b: s*a+b, s, a, b)

performance = (1e-9) * 24.0 * nr * n / t

return performance
```

Note that operations on Star-P distributed arrays work in a manner identical to NumPy; for a distributed multi-dimensional array, `X`, `a * X` for a being `a` scalar scales each element of `X` by `a` and `X + Y` for two Star-P distributed arrays returns an element-wise sum of the arrays.

3.6 Performance

We timed the performance of our implementations on a 32-node cluster at San Diego Supercomputing Center composed of quad core Intel Xeon 5 140 processors, each with 8 GB of main memory and connected via an Infiniband network. The results on running the benchmarks are summarized below:

Number of Processes	LINPACK (Gflops/sec)	FFT (Gflops/sec)	Stream (Gflops/sec)
16	50.7	35.6	17.9
32	98.9	74.0	35.6
64	166.0	152.6	70.8
96	254.2	232.5	106.5
128		300.0	139.5

In each case, the size of the problem was fixed in accordance to the guidelines in the HPC Challenge specification [6] (for example, for HPL, the total memory of the problem was chosen as half the available system memory).

As seen in the above table, the benchmarks scale close to linearly with increasing number of processes for all the implemented cases, even though they were implemented using only a few lines of code in each case.

3.6.1 Other Applications

Apart from the small benchmarks implemented in this article, Star-P can be used to implement real world applications in financial engineering [7], life sciences [8], defense and intelligence and [9] other areas. In most of these cases, users not previously familiar with low-level parallel computing principles have been able to easily and effectively parallelize critical applications that could not have run on a desktop computer and would have taken a substantial amount of effort to parallelize using tradition techniques such as message passing. More examples of user successes for non-benchmark applications can be found in [10].

4 RELATED WORK

There are a number of extension modules that provide facilities for parallel programming in Python. Many of them such as PyMPI [11], MPI4Py [12] and MyMPI [13] wrap MPI libraries on top of a standard Python interpreter. While this approach has the advantage that users with existing knowledge of MPI can prototype parallel programs inside Python, the main disadvantage is that this model is still significantly hard to use for programmers with little or no experience with message passing. Moreover, all the classic issues associated with classical parallel programming (such as deadlocks and race conditions) still exist using these approaches. Another disadvantage of wrapping MPI libraries in Python is that it is relatively hard to interface with existing parallel libraries written in C or Fortran.

A different approach to interactive parallel programming in Python is IPython1 [14] that is based on a client-server architecture similar to Star-P. IPython1 allows a user to asynchronously execute commands from a single controller (akin to the Star-P client) to multiple engines (akin to the processes in the Star-P server). The programming model in IPython1 allows a user to perform embarrassingly parallel computations in a manner similar to the `PPEVAL` construct in Star-P, but in order to perform computations that require communication between the engines, the user must explicitly use message passing like the other solutions discussed above. In contrast, with Star-P, the embarrassingly parallel computations can be performed using `PPEVAL` and computations that require communication can be constructed out of the dozens of parallelized primitive operations on distributed arrays without requiring any knowledge of message passing. Moreover, the Star-P client is fully integrated and can be used in conjunction with the IPython interpreter.

5 CONCLUSIONS AND FURTHER WORK

In this article, we described an interactive parallel computing environment for Python. The programming model for Star-P Python provides a good balance between productivity and performance and makes it very convenient to parallelize existing numerical serial applications written using the NumPy package without requiring any knowledge of low-level parallel programming primitives such as message passing . We also described the implementation of a sample set of benchmarks along with representative performance and scaling results on a cluster of contemporary multicore machines.

In future versions of Star-P Python, we plan to support for additional distributed data types in addition to multi-dimensional matrices and add more features in the `PPEVAL` construct such as facilities for load balancing.

Acknowledgements

We would like to thank D. J. Choi and Nancy Wilkins-Diehr of San Diego Supercomputing Center for providing us access to their cluster for running our benchmarks.

REFERENCES

[1] van Rossum, G.—Drake, F. L.: Python Reference Manual, Release 2.5.1. `http://docs.python.org/ref/ref.html`.

[2] Oliphant, T. E.: Guide to NumPy. `http://www.tramy.us/`.

[3] Interactive Supercomputing, Getting Started with Star-P for Python. `http://www.interactivesupercomputing.com/support/content/pdf/PyGettingStartedGuide.pdf`.

[4] Interactive Supercomputing, Star-P for Python User Guide. `http://www.interactivesupercomputing.com/doc/2.5.1/pdf/ISC_Starp_Python_UG_R251.pdf`.

[5] Interactive Supercomputing, Software Development Kit Guide. `http://www.interactivesupercomputing.com/doc/2.5.1/pdf/ISC_SDK_Tutorial_R251.pdf`.

[6] Dongarra, J.—Luszczek, P.: Introduction to the HPCChallenge Benchmark Suite. ICL Technical Report, ICL-UT-05-01, Innovative Computing Laboratory, University of Texas at Knoxville, 2005.

[7] Interactive Supercomputing, Industry Solutions: Financial Engineering. `http://www.interactivesupercomputing.com/industrysolutions/is_finance.php`.

[8] Interactive Supercomputing, Industry Solutions: Life Sciences. `http://www.interactivesupercomputing.com/industrysolutions/is_lifesciences.php`.

[9] Interactive Supercomputing, Industry Solutions: Defense and Intelligence. `http://www.interactivesupercomputing.com/industrysolutions/is_defense.php`.

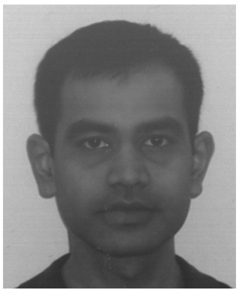
[10] Interactive Supercomputing, Customer Video Page. `http://www.youtube.com/user/ParallelLounge`.

[11] Miller, P. et al.: PyMPI project. `http://pympi.sourceforge.net/`.

[12] Dancin, L. et al.: MPI4Py Project. `http://mpi4py.scipy.org/`.

[13] Kaiser, T.: MyMPI Project. `http://peloton.sdsc.edu/~tkaiser/mympi`.

[14] Granger, B.—Perez, F. et al.: IPython1: Interactive Parallel Computing. `http://ipython.scipy.org/moin/Parallel_Computing`.



Sudarshan Raghunathan is a Member Technical Staff at Interactive Supercomputing, Inc. since 2005 where he primarily works on the core Star-P server runtime. He received his M. Sc. and Ph. D. in computational engineering from Massachusetts Institute of Technology in 2002 and 2005, respectively. Apart from parallel scientific computing, his research interests include multiresolution signal processing techniques such as wavelets, partial differential equations, preconditioning methods and mesh adaptation techniques.