Parallel, Distributed Scripting with Python

P. Miller
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Abstract[1]

Interpreted languages are ideal both for building control and administration tools as well as for providing a framework for non-time critical components of a larger application (e.g. GUI and file handling). On parallel architectures such as large SMP's and clusters, a *parallel* scripting language is needed to provide the same functionality. We present `pyMPI`, a set of parallel enhancements to the Python language that enables programmers to write truly distributed, parallel scripts. It allows developers to write parallel extension modules.

[1]This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes. This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

1 Introduction

Parallel computers used to be, for the most part, one-of-a-kind systems which were extremely difficult to program portably. With SMP architectures, the advent of the POSIX thread API and OpenMP gave developers ways to portably exploit on-the-box shared memory parallelism. Since these architectures didn't scale cost-effectively, distributed memory clusters were developed. The associated MPI message passing libraries gave these systems a portable paradigm too. Having programmers effectively use this paradigm is a somewhat different question. Distributed data has to be explicitly transported via the messaging system in order for it to be useful.

In high level languages, the MPI library gives access to data distribution routines in C, C++, and FORTRAN. But we need more than that. Many reasonable and common tasks are best done in (or as extensions to) scripting languages. Consider sysadm tools such as password crackers, file purgers, etc... These are simple to write in a scripting language such as Python (an open source, portable, and freely available interpreter). But these tasks beg to be done in parallel. Consider the a password checker that checks an encrypted password against a 25,000 word dictionary. This can take around 10 seconds in Python (6 seconds in C). It is trivial to parallelize **if** you can distribute the information and co-ordinate the work.

2 pyMPI

Here we present `pyMPI`, a distributed implementation of Python extended with an MPI interface. The tool makes it easy to write parallel Python scripts for system administration, data exploration, file post-processing, and even for writing full blown scientific simulations. Parallel Python also allows developers to prototype the data distribution for parallel algorithms in a easy, interactive, and intuitive manner without having to compile code, build specialized MPI types, and build serialization mechanisms. `pyMPI` supports most of the MPI API. It allows access to sends, receives, barriers, asynchronous messaging, communicators, requests, and status. In short, it provides a fully functional parallel environment coupled with a powerful scripting engine. The combination simplifies the generation of large scale, distributed tools for clusters.

Scripting languages are often used as glue – to mix and match previously unrelated tools and modules to solve a problem. Parallel scripting languages can combine serial components to build parallel applications. In some cases, the parallelization is trivial and one simply needs a simple framework in which to execute – `pyMPI` provides that. For example, one could write a parallel Python script to undertake a parameter study of a serial application or component. It is a distributed loop, but it is easy to write in parallel Python. See Figure 2.

It is important that the parallel scripting framework have a complete and robust implementation of MPI. This is particularly true when tasks need to be co-ordinated. Consider pre-processing graphics files and combining into a

```
import mpi
...
for x in mpi.scatter(parameters):
   os.system('sample_application -parameter=\%s'%x)
...
```

Figure 1: A distributed loop in pyMPI

```
import mpi
...
for file in mpi.scatter(files):
   os.system('postprocess %s'%file)
mpi.barrier()
if mpi.rank == 0:
   os.system('buildmaster %s'%string.join(files))
...
```

Figure 2: Using a barrier

master file. A simple shell script can only co-ordinate through the file system, but pyMPI can simply throw a barrier as in Figure 2

One can also co-ordinate a parallel application written as a Python extension. In one application, we have written a sophisticated, parallel, multi-level time-step control in pure Python (enhanced with MPI). Similarly, one can use the capabilities of a parallel enhanced Python to bring distributed data back together so that serial tools can work on the combined data as in Figure 2

Extra nodes can be utilized to perform clean-up, post-processing, and archival tasks that need to be tightly co-ordinated with a parallel task. In the example below, dump files are moved off to the archive after every step. Since the move is prompted by a MPI message from the application, the files will never be picked up in an intermediate state (e.g. the file exists, but has not been finalized). See Figure 2

In a similar manner, one can construct simulations of large grain components. Consider a physically divided computation such as a turbo-machinery simulation. The application follows air flows through compressor turbines through a combustor and then through exhaust vanes. Consider the major components of the application: a turbine simulator, a combustor simulator, and an interface

```
import mpi
...
# All processors do this...
rho = simulation.computeDensity()

global_position = mpi.gather(x)
global_ho = mpi.gather(rho)
if mpi.rank == 0:
  # Write in style suitable for gnuplot
  for i in range(len(global_position)):
    print global_position[i],global_rho[i]
...
```

Figure 3: Using gather to build a global field

```
import mpi
# Build a sub communicator for the ‘‘real’’ work
comm = mpi.dup(world[:-1])
if comm:
   worker = something(comm) # Use the sub-communicator
   for i in range(steps):
worker.do_work()
comm.barrier()   # Wait just on workers
mpi.send("archive",mpi.size-1,tag=0) # on world comm
   mpi.send("done",mpi.size-1,tag=1) # on world comm
else:
   # A single cleanup process
   while 1:
      msg, status = mpi.recv()
      if msg == "archive":
          <copy files to archive>
      elif msg == "done":
          break
```

Figure 4: Setting aside a single process for cleanup

control. Parallel Python can be used to set up the communicators for each, handle details of data manipulation, control outputs, handling load balancing, etc...

3 Buying into scripting

There are three basic premises upon which pyMPI is built.

1. Scripting is more than just *cool*. There are a number of applications for which scripting is the appropriate way to solve or at least prototype the solution.

2. MPI is the **right** way to go massively parallel (maybe not a good way, but reasonable anyhow).

3. An order of magnitude slowdown in runtime is acceptable because of the human programmer savings

Scripting is more than just a fun way to program. For short one-off runs, the time to solution is driven by programmer time. Interpreted scripting languages give power and flexibility to the programmer and can avoid a slow compile-link-run-debug cycle. Beyond simple hacking, scripting can be a great way to prototype solutions – working out details and structures before committing them to a compiled asset. Advanced scripting languages are also designed to build large systems. Languages like Python[vR98] and Ruby[TH01] provide modular, object-oriented structures for building large, complex systems[RMO+00]. The idea is to do work where it is most appropriate – interpreted scripts or in compiled extensions. Scripting languages also allow previously unrelated modules to be combined in new and interesting ways. This allows a simulation developer to leverage existing graphics and plotting packages, for instances [JO02].

Parallel programming is hard, and doing it portably is harder still. The MPI library makes the problem at least tractable. From a robust reference implementation[GL96], programmers can reliably count on compilation and link portability across a wide range of SMP, distributed, and cluster architectures.

Nothing comes for free. Scripting languages tend to be one to two orders of magnitude slower than similar solutions crafted in a purely compiled language. For number crunching, it can be as much as three orders of magnitude! For many applications, this speed is acceptable. When the response time of an application 10 milliseconds instead of 1 microsecond, it may not matter. For applications that are compute bound in a library or where I/O is the limiting factor, scripting languages can be appropriate. The popular *Myst* family of PC games were originally crafted in Apple's HyperCard interpreter[DeV00]. Parts of Kalisto USA's SpongeBob Squarepants game for PS2 and GameCube are written in Python [Asb02] with a small Python interpreter adapted to each game system.

4 MPI enabled extensions to Python

Serial Python has an established and well documented C API for building extension modules[vR02]. If, however, the extension uses MPI parallelism, the serial API offers little help to the programmer. A variety of approaches are applicable to the problem. One can do make Python minimally compliant with MPI enabled applications. One can expose a bare minimum of the MPI library to enable some control of MPI extensions. Or lastly, one can port large segments of the MPI specification to Python to allow native parallel code to be written.

4.1 Minimally compliant MPI Python

The MPI library, depending on it's implementation[2], can be quite sensitive to initialization time[For95] In practice, this means that `MPI_Init()` should be called as soon as practical after entering the main() routine. Since Python *does* change the state of the the process (file opens and dynamic loads) before handing control over the user, it is imperative to make some kind of patch to the Python source that calls `MPI_Init()`.

The CAMP (Center for Atomic-scale Materials Physics in Denmark) built exactly such a patch[Sch]. It essentially calls `MPI_Init()` at the right time and then hopes the user routines do something reasonable. The individual Python interpreters that are spawned can do nothing to coordinate their actions that isn't preprogrammed into the parallel extension modules.

4.2 Adding coordination primitives

Beyond just getting a parallel program to simply operate in a Python environment, a programmer needs more control. At a minimum, a programmer needs some kind of synchronization primitive and some kind of messaging primitive. On massively parallel, distributed machines, it also nice to have some kind of control over console I/O lest your output be inundated with 1000 "here I am" messages!

David Beazley of SWIG fame developed a version of parallel Python for MPI with just `MPI_Barrier()`, `MPI_Allreduce()` (for double precision only), and a way to turn console and I/O on and off. It was written for Python 1.3 and ported by Python 1.4 by Brian Yang of Lawrence Livermore National Laboratory[BL97]. The patches to make this work required many changes to the Python core implementation which made the whole project unmaintainable. Development on the project has since been abandoned[Bea01]. It was, until recently however, the core parallel framework for a major simulation code at Lawrence Livermore National Laboratory[RMO+00] before being replaced with `pyMPI` [MCC+01].

[2] [GL96] In the `MPI_Init()` description, the MPICH reference implementation state "The MPI standard does not say what a program can do before an MPI_INIT or after an MPI_FINALIZE. In the MPICH implementation, you should do as little as possible. In particular, avoid anything that changes the external state of the program, such as opening files, reading standard input or writing to standard output."

The `pypar` project of the Australian National University[Nie] has a more powerful set of primitives defined (barriers and blocking send/receive). It was used as a teaching language to introduce students to message passing concepts.

4.3 Other more advanced models

Konrad Hinsen's `Scientific.MPI` module adds more primitive operations, including communicators[Hin00], but his bindings only loosely follow the MPI naming scheme. Additionally, his messaging style is array based which is an unnatural fit to Python objects. Much of his parallel implementation is designed to support the BSP (Bulk Synchronous Parallel) style of parallel programming rather than a traditional arbitrary message model.

5 pyMPI

The `pyMPI` project is designed to expose as much of the MPI library as *makes sense* to the Python programmer. It can either control MPI enabled extensions or be used to write pure Python parallel programs. It supports multiple communicators and (near) arbitrary Python types. It maps most MPI functions onto communicator member functions, but then uses the power of Python's dynamic objects to rebind those names to functions. For instance a call to `mpi.send` is really a call to `MPI_Send()` on the world communicator. `pyMPI` is open source code available from SourceForge[Mil].

`pyMPI` builds an alternate startup executable for Python, but parasitic-ally uses the installed base of Python code modules. That is to say, if you build `pyMPI` on top of, say, `/usr/local/bin/python2.2`, then `pyMPI` will use the same string, regular expression, etc... modules as `python2.2`.

`pyMPI` by default runs in a sort of SPMD mode by instantiating multiple Python interpreters each with access to (several) world communicators and each with a unique rank corresponding to a process ID. The program initializes MPI on startup (the builder can specify an alternate startup method), sets up interfaces to MPI_COMM_WORLD and its clone PYTHON_COMM_WORLD and initializes the parallel console I/O facility. `pyMPI` clones the WORLD communicator so that Python messages are not inadvertently confused with user extension messages that use COMM_WORLD.

5.1 Parallel styles

By default, `pyMPI` starts up in SPMD (Same Program Multiple Data) mode. That is, all processes are running the same script. The execution is asynchronous (e.g. not in lockstep) unless synchronizing operations are used. Each process has a unique rank on the WORLD communicator, so execution can start to diverge. For instance, each process can read files based on its rank:

```
>>> input = open('foo%d.data'%mpi.rank)
```

```
import mpi
n = mpi.size / 2  # Use half for each partition
turbine_comm = mpi.comm_create(mpi.WORLD[:n])
combustor_comm = mpi.comm_create(mpi.WORLD[n:])
if turbine is not None:
  import turbine
  run_turbine(turbine_comm)
if combustor_comm is not None:
  import combustor
  run_combustor(combustor_comm)
```

Figure 5: Code harness for MIMD style computation

As execution evolves, diverging execution paths may be taken depending on the data encountered.

While `pyMPI` does not yet directly support MIMD mode on the command line (i.e. specifying multiple scripts to run), it is quite easy to mimic such behavior by building the appropriate sub-communicators. For instance:

```
>>> sub = mpi.WORLD.comm_create(mpi.WORLD[:n])
```

will create a sub communicator on the first **n** processors of the WORLD communicator and return the special value `None` on the others. Figure 5.1 illustrates how to set up a system wherein part of a simulation can run a turbine module while the rest runs a combustor module.

5.2 Console I/O controls

With the potential to use hundreds to thousands of processors, it is important to be able to control console output. The simplest control is to simply ignore output to a particular process. `pyMPI` provides a simple mechanism to throttle output on `stdout` and `stderr`.

```
>>> mpi.synchronizeQueuedOutput('/dev/null')
```

The `synchronizeQueuedOutput()` command is synchronous across a communicator (mpi implies the PYTHON_COMM_WORLD communicator) and it flushes any previously queued output (we'll see that below) and, with the `/dev/null` argument, redirects standard output to the null file.

Of course, there are situations when simply tossing all output is inappropriate. For example, when the interpreter is throwing exceptions on one processor, the user doesn't want to lose any output. Yet, it can be difficult to discern which processor output a particular line particularly since parallel console output can be asynchronously interspersed. `pyMPI` handles this case by redirecting output to a family of files:

```
>>> mpi.synchronizeQueuedOutput('foo','goo')
```

This command creates a series of files, foo.out.1, foo.out.2, ... and goo.err.1, goo.err.2, ... that hold the output and errput streams for the rank 1, 2, ... processors respectively.

Different ranks can choose different styles so long as all synchronously call the function. Use:

```
>>> mpi.synchronizeQueuedOutput(None)
```

to restore the original default output stream. Note that the rank 0 process always writes to the original output or errput stream.

5.3 Making it look like MPI and Python

In the philosophy of `pyMPI`, the semantics it provides need to mimic standard MPI. For a system to provide a good prototyping platform, it should be trivial to transliterate back and forth. In `pyMPI` this is provided by hoisting functions and constants into the `mpi` module. Where in C, one would write

```
status = MPI_Allreduce(&value, ..., MPI_SUM, ...);}
```

in Python we would write

```
value = mpi.allreduce(value,mpi.SUM)
```

In essence, the MPI_ is replaced with mpi.

At the same time, Python has a style and semantic that should be followed as well. Objects are simply convenient references. The statement:

```
>>> C += 1
```

does **not** mean to increment and update the value `C`, but rather it means "take the value of C, increment that value as appropriate for the type, and rebind the new value to the name C." A particular types implementation may take some care to reuse the storage, but it is not part of the language semantic. Similarly, Python provides no API mechanism for updating a value in place. So in the function call

```
>>> x = 3; f(x)
```

The value of `x` is still three after the function call. Python does not have an & operator like C to force update in place (indeed, since Python uses an interned constant model, updating the value of 0 would redefine the value of zero throughout the execution!).

`pyMPI` uses a value return model to get around that issue. MPI calls like recv, sendrecv, reduce, allreduce, and gather return the messages they receive (and in the case of recv, a status object describing the sender and tag). This allows `pyMPI` to message all objects the same way without having to resort to explicitly packing them into buffers. Note that the normal MPI status return is not used. `pyMPI` rigorously error checks MPI calls and converts MPI errors into Python exceptions. Like MPI, however, `pyMPI` cannot guarantee that execution

can continue after certain MPI errors occur. The system does insure that all such errors are detected and that the programmer is given the opportunity to catch the error and end execution cleanly.

5.4 The "two message" model

Since the users are simply messaging arbitrary Python objects, buffer preallocations are impractical and are semantically difficult to provide and enforce. A Python container like a list, dictionary, or array is treated as a single object; the serialization/de-serialization handle the details. This means, however, that `pyMPI` cannot be completely sure how large a method might be until it actually gets it. The MPI standard is deliberately hazy on error recovery. Even when error handlers are set to return messages (MPICH generally defaults to fatal errors), you cannot in practice recover from a receive into an undersized buffer.

`pyMPI` solves this problem by sending either one or two messages. The first message, in a fixed size buffer, either fits or indicates the size of the remaining buffer that comes in a *second* message that follows. The current small message size is set at 64 bytes, but in practice should be the eager limit for sending small messages. This has an impact on performance.

5.5 Building on Python's big shoulders

The Python framework included some important features that made implementing `pyMPI` relatively easy. First and foremost was the well developed C API for building extensions. It was relatively easy to produce the interface tables and implementation functions required to add the MPI extension module. Secondly, the object serialization was handled by the Python `pickle` module. This simplified the task of messaging arbitrary objects. The `pickle` module, however, is accessed through the Python function call API which is rather heavy weight. The `pyMPI` could, but does not, shortcut directly to the C implementation. It also used to, but does not currently, special case common messages like long integers and double precision floats. Lastly, Python provided a simple to use hook to replace the readline call. This allows `pyMPI` to simply read console input on the master node and broadcast it (`MPI_Bcast()`) to the slave nodes. This allows user interactively even with hundreds of processors In so doing, it even supports GNU readline style histories across all the processors. The input routine simply replicates the same deletions, insertions, and control characters across all processors.

The largest problem that was encountered was getting the reference counting right. Python's uses a pure reference count model to know when to reclaim memory (with a garbage collector to break cycles in object chains). Reference counting mistakes appear as either memory leaks (a beta version of `pyMPI` leaked a megabyte array on a gather) or worse as early reclamation of an object leading to a core dump or corrupted memory when a stale pointer is accessed. In a language like Ruby which uses mark-and-sweep garbage collection or in a language like Java with no explicit pointers this could have been avoided.

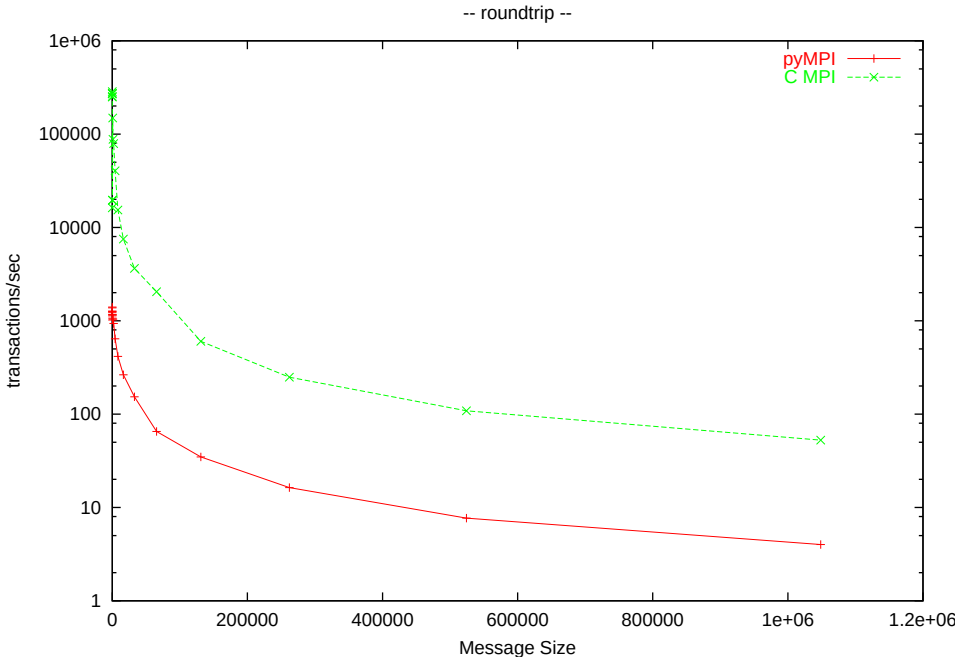

Figure 6: Ping-Pong test in pyMPI and C

5.6 Performance

The Python versions of most benchmarks inherently include more work than their C counterparts, so some drop off in performance is expected. In particular, most C benchmarks consist of messaging primitives that send or receive from raw, fully processed byte buffers (e.g. an array of characters). In Python, sends must run the object serializer to convert an arbitrary Python object or array into a character string. At a minimum, this requires allocating space from the heap and making a copy to actually send. After the send completes (either following a blocking send or after a successful wait on an isend), the data must be deallocated as well. On a receive, the reverse with a commensurate amount of work (allocate, copy, deallocate) is done. The round trip (ping pong) test in Figure 5.6 shows a roughly two orders of magnitude drop off in performance using `pyMPI` for a Linux SMP running LAM MPI. Similar drop offs are occur for latency, bandwidth, and bidirectional bandwidth computations. The Python benchmarks are based on the `LLCbench.mpbench` MPI benchmark suite from the University of Tennessee [Muc00]. The C benchmarks were converted to Python extensions and run by `pyMPI`. Figure 5.6 shows the Python code used to generate Figure 5.6.

```
def roundtrip(cnt,bytes):
    if mpi.rank == 0:
        TIMER_START()
        for i in range(cnt):
            mpi.send(message[:bytes],slave)
            msg,status = mpi.recv(slave)
        TIMER_STOP()

        total = TIMER_ELAPSED()
        return cnt / ( total *1e-6 ),"transactions/sec"

    elif mpi.rank == slave:
        for i in range(cnt):
            msg,status = mpi.recv(master)
            mpi.send(message[:bytes],master)
        return 0.0,"transactions/sec"

    else:
        return 0.0,"transactions/sec"
```

Figure 7: Ping-Pong test in pyMPI and C

5.7 Example

The first example code in [GLS94] implements a simple parallel program to compute π. The original code was written in FORTRAN with MPI library calls. That code can be easily transliterated into `pyMPI`. Table 5.7 shows the code.

5.8 Future work

Near term deliveries of `pyMPI` will include support for several missing MPI-1 functions, new MPI-2 functionality including MPI-IO support for both console objects and general parallel files. Additionally, the SIMD support will be enabled for a kind of `*Python` array processing language and MIMD support will be enhanced. Finally, better native MPI methods will be included to support messaging from Python components to C/FORTRAN components.

6 Conclusion

`pyMPI` couples the distributed parallelism of MPI with the scripting power of Python. MPI unleashes the power of large clusters, while Python brings access to hundreds of easy to use, freely available modules. Together, they can simplify both sysadm and scientific simulation tasks on high-end clustered machines.

```
import mpi

def f(self,a):
  "The function to integrate"
  return 4.0/(1.0 + a*a)

def integrate(self, rectangles, function):
  # All processes share the count
  n = mpi.bcast(rectangles)

  h = 1.0/n
  sum = 0.0
  # Expression "stripes" the rectangles
  for i in range(mpi.rank+1,n+1,mpi.procs):
      x = h * (i-0.5)
      sum = sum + function(x)

  myAnswer = h * sum
  answer = mpi.allreduce(myAnswer,mpi.SUM)
  return answer

print integrate(2000,f)
```

Table 1: Computing π in Python

Table 2: MPI components implemented in `pyMPI`

mpi.allgather, mpi.allreduce, mpi.barrier, mpi.bcast, mpi.cancel, mpi.cart_create, mpi.comm_create, mpi.comm_dup, mpi.comm_rank, mpi.comm_size, mpi.communicator, mpi.deltaT, mpi.finalize, mpi.finalized, mpi.free, mpi.gather, mpi.initialized, mpi.irecv, mpi.isend, mpi.map, mpi.mapserver, mpi.mapstats, mpi.name, mpi.procs, mpi.rank, mpi.recv, mpi.reduce, mpi.scan, mpi.scatter, mpi.send, mpi.sendrecv, mpi.size, mpi.synchronizeQueuedOutput, mpi.synchronizedWrite, mpi.test, mpi.test_cancelled, mpi.tick, mpi.version, mpi.wait, mpi.waitall, mpi.waitany, mpi.wtick, mpi.wtime, mpi.WORLD, mpi.COMM_WORLD, mpi.COMM_NULL, mpi.BAND, etc...

Implementation	URL
`pyMPI`	http://pympi.sourceforge.net
`pythonmpi`	http://www.fysik.dtu.dk/ schiotz/comp/
`ScientificPython.MPI`	http://starship.python.net/ hinsen/ScientificPython
`pypar`	http://datamining.anu.edu.au/ ole/pypar

Figure 8: MPI enhanced Python implementations

References

[Asb02] Jason Asbahr. Developing game logic: Python on the Sony Playstation 2 and Nintendo GameCube. In *Proceedings, OSCON 2002*, July 2002. 5

[Bea01] David Beazley. Personal communication. 2001. 6

[BL97] David Beazley and Peter Lomdahl. Feeding a large-scale physics application to python. Proceedings of the 6th International Python Conference, 1997. 6

[DeV00] Jeanne DeVoto. http://www.jaedworks.com/hypercard/, January 2000. 5

[For95] Message Passing Interface Forum. MPI: A message-passing interface standard. http://www.mcs.anl.gov/mpi/mpi-standard /mpi-report-1.1/mpi-report.html, 1995. 6

[GL96] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6. 5, 6

[GLS94] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994. 12

[Hin00] Konrad Hinsen. The molecular modeling toolkit: A new approach to molecular simulations. *J. Comp. Chem*, 21, 2000. 7

[JO02] Eric Jones and Travis Oliphant. Scientific computing with Python. In *The Tenth International Python Conference*, 2002. 5

[MCC+01] Kathleen McCandless, Martin Casado, Charlie Crabb, Susan Hazlett, Jeff Johnson, Patrick Miller, and Chuzo Okuda. Taming kull the conqueror: Innovative computer science development in a 3d, parallel, multi-physics application. Ucrl-jc-146343-abs, LLNL, 2001. 6

[Mil] Patrick Miller. pympi. http://pympi.sourceforge.net. 7

[Muc00] Phillip Mucci. http://www.cs.utk.edu/ mucci/DOD/mpbench.ps, 2000. 11

[Nie] Ole Moller Nielsen. pypar. http://datamining.anu.edu.au / ole/pypar/. 7

[RMO+00] J A Rathkopf, D S Miller, J M Owen, L M Stuart, M R Zika, P G Eltgroth, N K Madsen, K P McCandless, P F Nowak, M K Nemanic, N A Gentile, N D Keen, and T S Palmer. Kull: Llnl's asci inertial confinement fusion simulation code. Ucrl-jc-137053, LLNL, 2000. 5, 6

[Sch] Jakob Schiotz. pythonmpi. http://www.fysik.dtu.dk/ schiotz/comp /PythonAndSwig/pythonMPI.html. 6

[TH01] Dave Thomas and Andy Hunt. Programming in ruby. *Dr. Dobb's Journal*, January 2001. 5

[vR98] Guido van Rossum. Glue it all together with Python. OMG-DARPA-MCC Workshop on Compositional Software Architecture, January 1998. 5

[vR02] Guido van Rossum. Python/C API reference manual. http://www.python.org/doc/current/api/api.html, 2002. 6