Using Pyrex to Speed up SAGE and to Interface C/C++ Libraries

Martin Albrecht (malb@informatik.uni-bremen.de)

January 1, 2007

Outline

1 What is Pyrex

2 Performance and Pyrex

3 Pyrex and C++

4 ToDo

Basic Facts about Pyrex

Pyrex lets you write code that mixes Python and C data types any way you want, and compiles it into a C extension for Python.

(http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/)

- Written by Greg Ewing of New Zealand.
- `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`
- Python-like code converted to C code that is compiled by a C compiler. All non-C memory management done automatically.
- Easy way to implement C extension modules for Python and to interface Python to C and C++ libraries.

Pyrex and SAGE I

Time-critical SAGE code gets implemented in Pyrex, which **is (as fast as) C** *code, but easier to read (e.g., since all variables and scopes are explicit).*

(http://modular.math.washington.edu/talks/2006-07-09-cnta/2006-07-09-cnta.pdf)

"is (as fast as) C"

This is not necessarily true, you need to write almost C for this

Lots of code in SAGE like library interfaces and basic arithmetic types already implemented in Pyrex:

```
$: cat */*.pyx */*/*.pyx */*/*.pyx | wc -l
63706
```

Pyrex and SAGE II

The version of Pyrex shipped with SAGE is patched:

- two patches to allow `cimports` across directories by William Stein and me.
  - probably will never be accepted upstream as Greg Ewing doesn't like them.
  - He doesn't consider the bug we reported a bug.
- Several patches so that Pyrex works with Python 2.5

Getting Started with Pyrex in SAGE

You may start writing Pyrex code by

- writing an `.spyx` file and loading/attaching it,
- put `%pyrex` on top of a notebook cell, it will get compiled and executed, or
- write a `.pyx` file and add it to `setup.py`.

Now write your almost Python code, besides some exceptions:

Pyrex and Python Differences I

- No list comprehension:

```
sage: [f(i) for i in range(xyz)] # no valid Pyrex code!
sage: map(f,range(xyz))
sage: [i for i in range(xyz) if f(i)] #no valid Pyrex code!
sage: filter(f,range(xyz))
```

- No $i+=1$ etc., use $i = i + 1$
- No `__le__`, `__eq__`, `__ne__`, etc. but `__cmp__` and `__richcmp__`
- In `Class.__add__(left,right)` `left` doesn't need to be of type `Class`; no `__radd__` etc.
- Pickling (saving and loading objects) doesn't "just works", implement `__reduce__`
- no `yield`: Write an iterator class and implement `__next__` there.

Pyrex and Python Differences II

- cdef you class to allow access from C but that invalids **AttributeError** programming like this:

```
try:
  return self.__cached_result
except AttributeError
 self.__cached_result = self._calculate_result() #won't work
 return self.__cached_result
```

- Instead all members must be known at compile time:

```
cdef class MyClass
  cdef object __cached_result
   ...
  def calculate_result(MyClass self):
    if self.__cached_result != None:
      return self.__cached_result
    else:
      self.__cached_result = self._calculate_result()
      return self.__cached_result
```

Outline

1 What is Pyrex

2 Performance and Pyrex

3 Pyrex and C++

4 ToDo

"Premature Pyrexification is the Root of all Evil"

Before you port your class to Pyrex profile and test it!

- Profiling/debugging Python code is much more convenient than profiling Pyrex code to spot algorithmic bottle-necks.
  - the iPython profiler frontend

```
sage: R.<a,b> = PolynomialRing(GF(2),2)
sage: %prun for i in range(10000): _ = a+b
```

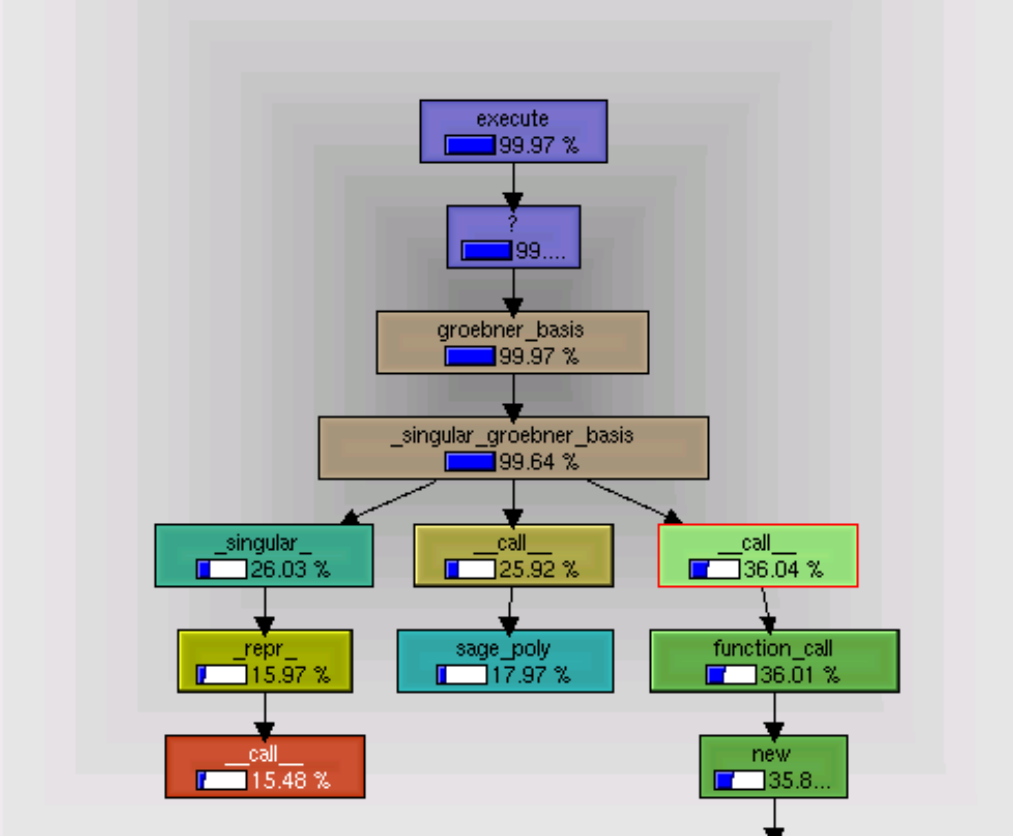
  - hotshot

```
sage: R.<s,a,g,e> = PolynomialRing(GF(2),4)
sage: I = sage.rings.ideal.Cyclic(R)
sage: import hotshot
sage: filename = "pythongrind.prof"
sage: prof = hotshot.Profile(filename, lineevents=1)
sage: prof.run('I.groebner_basis()')
sage: prof.close()
```

- Python profilers don't really pick up extension code.

Hotshot

You may convert the output of hotshot using `hotshot2calltree` and view the result in `kcachegrind`.



Profiling and Debugging Pyrex

Use the tools you would use to profile C/C++ applications. You profile the Python application then.

gdb and all it's frontends like DDD

valgrind Excellent memory debugger (`--leak-check=full`) and profiler (`--tool=callgrind`).

gprof Standard GNU profiler, needs recompilation of C/C++ code, haven't tested it.

Tips to Gain Speed I

- Pyrex tries to make things easy for you which may interfear with speed.
- `cdef` all integers as `int` if possible
- Use `int` **for**-loops:

```
cdef int i #this is important!
for i from 0 <= i < n:
  # do something
```

- Pyrex knows `cdef f()` functions/methods and `def f()` functions/methods. The later are callable from Python but calling them is much more expensive than calling a `cdef` function/method.
- Avoid Python! If you basically call heaps of Python code things won't be faster

Tips to Gain Speed II

- `isinstance` is expensive (discovery due to David Harvey), use `PyObject_TypeCheck`
- Pyrex plays safe when it comes to list, tuple, dict access:

```
def test():
    t = tuple([1,2])
    t[0]
```

`t[0]` gets translated to:

```
__pyx_1 = PyInt_FromLong(0);

if (!__pyx_1) {
  __pyx_filename = __pyx_f[0];
  __pyx_lineno = 10;
  goto __pyx_L1;
}
__pyx_3 = PyObject_GetItem(__pyx_v_t, __pyx_1);

if (!__pyx_3) {
  __pyx_filename = __pyx_f[0];
  __pyx_lineno = 10;
  goto __pyx_L1;
}
Py_DECREF(__pyx_1); __pyx_1 = 0;
```

Tips to Gain Speed III

```
Py_DECREF(__pyx_3); __pyx_3 = 0;
```

This is faster:

```
cdef extern from "Python.h":
    void* PyTuple_GET_ITEM(object p, int pos)

def test2():
    cdef object w
    t = tuple([1,2])
    w = <object> PyTuple_GET_ITEM(t,0)
    return 0
```

As it gets translated to:

```
_4 = (PyObject *)PyTuple_GET_ITEM(t,0);
Py_INCREF(_4);
Py_DECREF(w);
w = _4;
_4 = 0;
```

- So use Python C API directly, but be carefull with refcounting

Outline

1 What is Pyrex

2 Performance and Pyrex

3 Pyrex and C++

4 ToDo

C++ I

Pyrex knows no classes but it knows structs and function pointers. Those "look" like methods in classes when feed to a C++ compiler.

```
cdef extern from "linbox/field/givaro-gfq.h":

    ctypedef struct GivaroGfq "LinBox::GivaroGfq":
        #attributes
        int one
        int zero

        # methods
        int (* mul)(int r, int a, int b)
        ...
        unsigned int (* characteristic)()
        ....
    GivaroGfq *gfq_factorypk "new LinBox::GivaroGfq" (int p, int k)
    GivaroGfq *gfq_factorypkp "new LinBox::GivaroGfq" (int p, int k, intvec poly)
    GivaroGfq  gfq_deref "*"(GivaroGfq *orig)
    void delete "delete "(void *o)
    int gfq_element_factory "LinBox::GivaroGfq::Element"()
```

C++ II

This class may now be used like this:

```
def some_function():
    cdef GivaroGfq *k
    cdef int e
    k = gfq_factorypk(2,8)
    e = k.mul(e,k.one,k.zero)
    delete(k)
```

To ensure that the resulting C++ code is feed to a C++ compiler specify `language='c++'` in setup.py:

```
linbox_gfq = Extension('sage.libs.linbox.finite_field_givaro',
                       sources = ["sage/libs/linbox/finite_field_givaro.pyx"],
                       libraries = ['gmp', 'gmpxx', 'm', 'stdc++', 'givaro', 'linbox'],
                       language='c++'
                       )
```

C++ III

- Templates are not supported but "C name specifiers" allow to deal with templates:

```
cdef extern from "linbox/integer.h":
    ctypedef struct intvec "std::vector<LinBox::integer>":
        void (* push_back)(int elem)

    intvec intvec_factory "std::vector<LinBox::integer>"(int len)
```

- Overloading of functions/methods is not supported. Create a C alias for every combination.
- If everything else fails: You can always wrap the C++ code in a C function and call this from Pyrex. However this introduces a function call as overhead.
- `pyrexembed` (shipped with SAGE) is a nice tool to do this: You write the C wrapper functions and the Pyrex code in one file and `pyrexembed` splits them up for you. (Slightly annoying when debugging etc.)

Outline

1 What is Pyrex

2 Performance and Pyrex

3 Pyrex and C++

4 ToDo

Inclusion of a C Data Structure Library

I propose **libcprops**

- `http://cprops.sourceforge.net/`
- pro: ANSI-C (which both Pyrex and I understand much better than C++)
- pro: data structures: **linked_list**, heap, priority_list, **hashtable**, hashlist, **avltree**. red-black tree . . .
- pro: thread safe
- pro: easy to read, I could adapt it
- con: recursive implementation which is supposed to be less performant than a iterative implementation but that is probably negligible

. . . but I haven't really evaluated it.

Other ToDos

- Make this work:

```
sage: o = SomePyrexClass()
sage: o.__add__??
<source code of SomePyrexClass.__add__>
```

- Make `inspect` work with extension modules as they are easily debugable and profilable.
- Incredibly useful documentation: William is writing a Pyrex chapter for the reference manual, David Harvey started a Wiki page for speed wisdom.
- What else?

Questions?

Thank You!