Sage Reference Manual: Parallel Computing

Release 7.0

The Sage Development Team

January 20, 2016

CONTENTS

CHAPTER

ONE

DECORATE INTERFACE FOR PARALLEL COMPUTATION

class sage.parallel.decorate.**Fork**(*timeout=0*, *verbose=False*)
A fork decorator class.

class sage.parallel.decorate.**Parallel**(*p_iter='fork'*, *ncpus=None*, ***kwds*)
Create a parallel-decorated function. This is the object created by parallel().

class sage.parallel.decorate.**ParallelFunction**(*parallel*, *func*)
Bases: object

Class which parallelizes a function or class method. This is typically accessed indirectly through Parallel.__call__().

sage.parallel.decorate.**fork**(*f=None*, *timeout=0*, *verbose=False*)
Decorate a function so that when called it runs in a forked subprocess. This means that it won't have any in-memory side effects on the parent Sage process. The pexpect interfaces are all reset.

INPUT:

- f – a function
- timeout – (default: 0) if positive, kill the subprocess after this many seconds (wall time)
- verbose – (default: False) whether to print anything about what the decorator does (e.g., killing the subprocess)

Warning: The forked subprocess will not have access to data created in pexpect interfaces. This behavior with respect to pexpect interfaces is very important to keep in mind when setting up certain computations. It's the one big limitation of this decorator.

EXAMPLES:

We create a function and run it with the fork decorator. Note that it does not have a side effect. Despite trying to change the global variable a below in g, the variable a does not get changed:

```
sage: a = 5
sage: @fork
... def g(n, m):
...         global a
...         a = 10
...         return factorial(n).ndigits() + m
sage: g(5, m=5)
8
sage: a
5
```

We use fork to make sure that the function terminates after one second, no matter what:

```
sage: @fork(timeout=1, verbose=True)
... def g(n, m): return factorial(n).ndigits() + m
sage: g(5, m=5)
8
sage: g(10^7, m=5)
Killing subprocess ... with input ((10000000,), {'m': 5}) which took too long
'NO DATA (timed out)'
```

We illustrate that the state of the pexpect interface is not altered by forked functions (they get their own new pexpect interfaces!):

```
sage: gp.eval('a = 5')
'5'
sage: @fork()
... def g():
...       gp.eval('a = 10')
...       return gp.eval('a')
sage: g()
'10'
sage: gp.eval('a')
'5'
```

We illustrate that the forked function has its own pexpect interface:

```
sage: gp.eval('a = 15')
'15'
sage: @fork()
... def g(): return gp.eval('a')
sage: g()
'a'
```

We illustrate that segfaulting subprocesses are no trouble at all:

```
sage: cython('def f(): print <char*>0')
sage: @fork
... def g(): f()
sage: print "this works"; g()
this works...

--------------------------------------------------------------------------
Unhandled SIG...
--------------------------------------------------------------------------
'NO DATA'
```

sage.parallel.decorate.**normalize_input**(*a*)

Convert `a` to a pair `(args, kwds)` using some rules:

- if already of that form, leave that way.
- if `a` is a tuple make `(a,{})`
- if `a` is a dict make `(tuple([]),a)`
- otherwise make `((a,),{})`

INPUT:

- `a` – object

OUTPUT:

- `args` – tuple

- `kwds` – dictionary

EXAMPLES:

```
sage: sage.parallel.decorate.normalize_input( (2, {3:4}) )
((2, {3: 4}), {})
sage: sage.parallel.decorate.normalize_input( (2,3) )
((2, 3), {})
sage: sage.parallel.decorate.normalize_input( {3:4} )
((), {3: 4})
sage: sage.parallel.decorate.normalize_input( 5 )
((5,), {})
```

`sage.parallel.decorate.`**`parallel`**(*p_iter='fork'*, *ncpus=None*, ***kwds*)

This is a decorator that gives a function a parallel interface, allowing it to be called with a list of inputs, whose values will be computed in parallel.

Warning: The parallel subprocesses will not have access to data created in pexpect interfaces. This behavior with respect to pexpect interfaces is very important to keep in mind when setting up certain computations. It's the one big limitation of this decorator.

INPUT:

- **`p_iter` – parallel iterator function or string:**
  - `'fork'` – (default) use a new forked subprocess for each input
  - `'multiprocessing'` – use multiprocessing library
  - `'reference'` – use a fake serial reference implementation
- `ncpus` – integer, maximal number of subprocesses to use at the same time
- `timeout` – number of seconds until each subprocess is killed (only supported by 'fork'; zero means not at all)

Warning: If you use anything but `'fork'` above, then a whole new subprocess is spawned, so none of your local state (variables, certain functions, etc.) is available.

EXAMPLES:

We create a simple decoration for a simple function. The number of cpus (or cores, or hardware threads) is automatically detected:

```
sage: @parallel
... def f(n): return n*n
sage: f(10)
100
sage: sorted(list(f([1,2,3])))
[(((1,), {}), 1), (((2,), {}), 4), (((3,), {}), 9)]
```

We use exactly two cpus:

```
sage: @parallel(2)
... def f(n): return n*n
```

We create a decorator that uses three subprocesses, and times out individual processes after 10 seconds:

```
sage: @parallel(ncpus=3, timeout=10)
... def fac(n): return factor(2^n-1)
sage: for X, Y in sorted(list(fac([101,119,151,197,209]))): print X,Y
```

```
((101,), {}) 7432339208719 * 341117531003194129
((119,), {}) 127 * 239 * 20231 * 131071 * 62983048367 * 131105292137
((151,), {}) 18121 * 55871 * 165799 * 2332951 * 7289088383388253664437433
((197,), {}) 7487 * 26828803997912886929710867041891989490486893845712448833
((209,), {}) 23 * 89 * 524287 * 94803416684681 * 1512348937147247 * 5346950541323960232319657
```

```
sage: @parallel('multiprocessing')
... def f(N): return N^2
sage: v = list(f([1,2,4])); v.sort(); v
[(((1,), {}), 1), (((2,), {}), 4), (((4,), {}), 16)]
sage: @parallel('reference')
... def f(N): return N^2
sage: v = list(f([1,2,4])); v.sort(); v
[(((1,), {}), 1), (((2,), {}), 4), (((4,), {}), 16)]
```

For functions that take multiple arguments, enclose the arguments in tuples when calling the parallel function:

```
sage: @parallel
... def f(a,b): return a*b
sage: for X, Y in sorted(list(f([(2,3),(3,5),(5,7)]))): print X, Y
((2, 3), {}) 6
((3, 5), {}) 15
((5, 7), {}) 35
```

For functions that take a single tuple as an argument, enclose it in an additional tuple at call time, to distinguish it as the first argument, as opposed to a tuple of arguments:

```
sage: @parallel
... def firstEntry(aTuple): return aTuple[0]
sage: for X, Y in sorted(list(firstEntry([((1,2,3,4),),((5,6,7,8),)]))): print X, Y
(((1, 2, 3, 4),), {}) 1
(((5, 6, 7, 8),), {}) 5
```

The parallel decorator also works with methods, classmethods, and staticmethods. Be sure to apply the parallel decorator after ("above") either the `classmethod` or `staticmethod` decorators:

```
sage: class Foo(object):
...       @parallel(2)
...       def square(self, n):
...           return n*n
...       @parallel(2)
...       @classmethod
...       def square_classmethod(cls, n):
...           return n*n
sage: a = Foo()
sage: a.square(3)
9
sage: sorted(a.square([2,3]))
[(((2,), {}), 4), (((3,), {}), 9)]
sage: Foo.square_classmethod(3)
9
sage: sorted(Foo.square_classmethod([2,3]))
[(((2,), {}), 4), (((3,), {}), 9)]
sage: Foo.square_classmethod(3)
9
```

Warning: Currently, parallel methods do not work with the multiprocessing implementation.

CHAPTER

TWO

REFERENCE PARALLEL PRIMITIVES

These are reference implementations of basic parallel primitives. These are not actually parallel, but work the same way. They are good for testing.

sage.parallel.reference.**parallel_iter**(*f*, *inputs*)

Reference parallel iterator implementation.

INPUT:

- `f` – a Python function that can be pickled using the pickle_function command.
- `inputs` – a list of pickleable pairs (args, kwds), where args is a tuple and kwds is a dictionary.

OUTPUT:

- iterator over 2-tuples `(inputs[i], f(inputs[i]))`, where the order may be completely random

EXAMPLES:

```
sage: def f(N,M=10): return N*M
sage: inputs = [((2,3),{}),  (tuple([]), {'N':3,'M':5}), ((2,),{})]
sage: set_random_seed(0)
sage: for a, val in sage.parallel.reference.parallel_iter(f, inputs):
...       print a, val
((2,), {}) 20
((), {'M': 5, 'N': 3}) 15
((2, 3), {}) 6
sage: for a, val in sage.parallel.reference.parallel_iter(f, inputs):
...       print a, val
((), {'M': 5, 'N': 3}) 15
((2,), {}) 20
((2, 3), {}) 6
```

CHAPTER

THREE

PARALLEL ITERATOR BUILT USING THE `FORK()` SYSTEM CALL

class sage.parallel.use_fork.**p_iter_fork**(*ncpus*, *timeout=0*, *verbose=False*, *reset_interfaces=True*)

A parallel iterator implemented using `fork()`.

EXAMPLES:

```
sage: X = sage.parallel.use_fork.p_iter_fork(2,3, False); X
<sage.parallel.use_fork.p_iter_fork instance at ...>
sage: X.ncpus
2
sage: X.timeout
3.0
sage: X.verbose
False
```

CHAPTER

FOUR

PARALLEL ITERATOR BUILT USING PYTHON'S MULTIPROCESSING MODULE

sage.parallel.multiprocessing_sage.**parallel_iter**(*processes*, *f*, *inputs*)

Return a parallel iterator.

INPUT:

- processes – integer
- f – function
- inputs – an iterable of pairs (args, kwds)

OUTPUT:

- iterator over values of f at args,kwds in some random order.

EXAMPLES:

```
sage: def f(x): return x+x
sage: import sage.parallel.multiprocessing_sage
sage: v = list(sage.parallel.multiprocessing_sage.parallel_iter(2, f, [((2,), {}), ((3,),{})]))
sage: v.sort(); v
[(((2,), {}), 4), (((3,), {}), 6)]
```

sage.parallel.multiprocessing_sage.**pyprocessing**(*processes=0*)

Return a parallel iterator using a given number of processes implemented using pyprocessing.

INPUT:

- processes – integer (default: 0); if 0, set to the number of processors on the computer.

OUTPUT:

- a (partially evaluated) function

EXAMPLES:

```
sage: from sage.parallel.multiprocessing_sage import pyprocessing
sage: p_iter = pyprocessing(4)
sage: P = parallel(p_iter=p_iter)
sage: def f(x): return x+x
sage: v = list(P(f)(range(10))); v.sort(); v
[(((0,), {}), 0), (((1,), {}), 2), (((2,), {}), 4), (((3,), {}), 6), (((4,), {}), 8), (((5,), {}
```

CHAPTER

FIVE

PARALLELIZATION CONTROL

This module defines the singleton class `Parallelism` to govern the parallelization of computations in some specific topics. It allows the user to set the number of processes to be used for parallelization.

Some examples of use are provided in the documentation of `sage.tensor.modules.comp.Components.contract()`.

AUTHORS:

- Marco Mancini, Eric Gourgoulhon, Michal Bejger (2015): initial version

class sage.parallel.parallelism.**Parallelism**

Bases: sage.misc.fast_methods.Singleton, sage.structure.sage_object.SageObject

Singleton class for managing the number of processes used in parallel computations involved in various fields.

EXAMPLES:

The number of processes is initialized to 1 (no parallelization) for each field (only tensor computations are implemented at the moment):

```
sage: Parallelism()
Number of processes for parallelization:
 - tensor computations: 1
```

Using 4 processes to parallelize tensor computations:

```
sage: Parallelism().set('tensor', nproc=4)
sage: Parallelism()
Number of processes for parallelization:
 - tensor computations: 4
sage: Parallelism().get('tensor')
4
```

Using 6 processes to parallelize all types of computations:

```
sage: Parallelism().set(nproc=6)
sage: Parallelism()
Number of processes for parallelization:
 - tensor computations: 6
```

Using all the cores available on the computer to parallelize tensor computations:

```
sage: Parallelism().set('tensor')
sage: Parallelism()  # random (depends on the computer)
Number of processes for parallelization:
 - tensor computations: 8
```

Using all the cores available on the computer to parallelize all types of computations:

```
sage: Parallelism().set()
sage: Parallelism()  # random (depends on the computer)
Number of processes for parallelization:
 - tensor computations: 8
```

Switching off all parallelizations:

```
sage: Parallelism().set(nproc=1)
```

get (*field*)

Return the number of processes which will be used in parallel computations regarding some specific field.

INPUT:

- `field` – string specifying the part of Sage involved in parallel computations

OUTPUT:

- number of processes used in parallelization of computations pertaining to `field`

EXAMPLES:

The default is a single process (no parallelization):

```
sage: Parallelism().reset()
sage: Parallelism().get('tensor')
1
```

Asking for parallelization on 4 cores:

```
sage: Parallelism().set('tensor', nproc=4)
sage: Parallelism().get('tensor')
4
```

get_all()

Return the number of processes which will be used in parallel computations in all fields

OUTPUT:

- dictionary of the number of processes, with the computational fields as keys

EXAMPLES:

```
sage: Parallelism().reset()
sage: Parallelism().get_all()
{'tensor': 1}
```

Asking for parallelization on 4 cores:

```
sage: Parallelism().set(nproc=4)
sage: Parallelism().get_all()
{'tensor': 4}
```

get_default()

Return the default number of processes to be launched in parallel computations.

EXAMPLES:

A priori, the default number of process for parallelization is the total number of cores found on the computer:

```
sage: Parallelism().reset()
sage: Parallelism().get_default()  # random (depends on the computer)
8
```

It can be changed via `set_default()`:

```
sage: Parallelism().set_default(nproc=4)
sage: Parallelism().get_default()
4
```

reset()

Put the singleton object `Parallelism()` in the same state as immediately after its creation.

EXAMPLE:

State of `Parallelism()` just after its creation:

```
sage: Parallelism()
Number of processes for parallelization:
 - tensor computations: 1
sage: Parallelism().get_default()  # random (depends on the computer)
8
```

Changing some values:

```
sage: Parallelism().set_default(6)
sage: Parallelism().set()
sage: Parallelism()
Number of processes for parallelization:
 - tensor computations: 6
sage: Parallelism().get_default()
6
```

Back to the initial state:

```
sage: Parallelism().reset()
sage: Parallelism()
Number of processes for parallelization:
 - tensor computations: 1
sage: Parallelism().get_default()  # random (depends on the computer)
8
```

set(*field=None*, *nproc=None*)

Set the number of processes to be launched for parallel computations regarding some specific field.

INPUT:

- `field` – (default: `None`) string specifying the computational field for which the number of parallel processes is to be set; if `None`, all fields are considered
- `nproc` – (default: `None`) number of processes to be used for parallelization; if `None`, the number of processes will be set to the default value, which, unless redefined by `set_default()`, is the total number of cores found on the computer.

EXAMPLES:

The default is a single processor (no parallelization):

```
sage: Parallelism()
Number of processes for parallelization:
 - tensor computations: 1
```

Asking for parallelization on 4 cores in tensor algebra:

```
sage: Parallelism().set('tensor', nproc=4)
sage: Parallelism()
Number of processes for parallelization:
 - tensor computations: 4
```

Using all the cores available on the computer:

```
sage: Parallelism().set('tensor')
sage: Parallelism()  # random (depends on the computer)
Number of processes for parallelization:
 - tensor computations: 8
```

Using 6 cores in all parallelizations:

```
sage: Parallelism().set(nproc=6)
sage: Parallelism()
Number of processes for parallelization:
 - tensor computations: 6
```

Using all the cores available on the computer in all parallelizations:

```
sage: Parallelism().set()
sage: Parallelism()  # random (depends on the computer)
Number of processes for parallelization:
 - tensor computations: 8
```

Switching off the parallelization:

```
sage: Parallelism().set(nproc=1)
sage: Parallelism()
Number of processes for parallelization:
 - tensor computations: 1
```

set_default (*nproc=None*)

Set the default number of processes to be launched in parallel computations.

INPUT:

- `nproc` – (default: `None`) default number of processes; if `None`, the number of processes will be set to the total number of cores found on the computer.

EXAMPLES:

A priori the default number of process for parallelization is the total number of cores found on the computer:

```
sage: Parallelism().get_default()  # random (depends on the computer)
8
```

Changing it thanks to `set_default`:

```
sage: Parallelism().set_default(nproc=4)
sage: Parallelism().get_default()
4
```

Setting it back to the total number of cores available on the computer:

```
sage: Parallelism().set_default()
sage: Parallelism().get_default()  # random (depends on the computer)
8
```

CHAPTER

SIX

CPU DETECTION

sage.parallel.ncpus.**ncpus**()

Detects the number of effective CPUs in the system.

EXAMPLES:

```
sage: sage.parallel.ncpus.ncpus()  # random output -- depends on machine.
2
```

See also:

- *Parallel Interface to the Sage interpreter*

CHAPTER

SEVEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

p

F

G

N

P

R

S