Parallel computations using RecursivelyEnumeratedSet and Map-Reduce

There exists an efficient way to distribute computations when you have a set S of objects defined by RecursivelyEnumeratedSet() (see sage.sets.recursively_enumerated_set for more details) over which you would like to perform the following kind of operations :

- Compute the cardinality of a (very large) set defined recursively (through a call to RecursivelyEnumeratedSet of forest type)
- More generally, compute any kind of generating series over this set
- Test a conjecture : i.e. find an element of S satisfying a specific property; conversely, check that all of them do
- Count/list the elements of S having a specific property
- Apply any map/reduce kind of operation over the elements of S

AUTHORS :

- Florent Hivert – code, documentation (2012-2016)
- Jean Baptiste Priez – prototype, debugging help on MacOSX (2011-June, 2016)
- Nathann Cohen – Some doc (2012)

Contents

How is this different from usual MapReduce ?

This implementation is specific to RecursivelyEnumeratedSet of forest type, and uses its properties to do its job. Not only mapping and reducing is done on different processors but also **generating the elements of** S.

How can I use all that stuff?

First, you need the information necessary to describe a RecursivelyEnumeratedSet of forest type representing your set S (see sage.sets.recursively_enumerated_set). Then, you need to provide a Map function as well as a Reduce function. Here are some examples :

- **Counting the number of elements**: In this situation, the map function can be set to lambda x : 1, and the reduce function just adds the values together, i.e. lambda x,y : x+y.

  Here's the Sage code for binary words of length ≤ 16

```
sage: seeds = [[]]
sage: succ = lambda l: [l+[0], l+[1]] if len(l) <= 15 else []
sage: S = RecursivelyEnumeratedSet(seeds, succ,
....:   structure='forest', enumeration='depth')
sage: map_function = lambda x: 1
sage: reduce_function = lambda x,y: x+y
sage: reduce_init = 0
sage: S.map_reduce(map_function, reduce_function, reduce_init)
131071
```

  One can check that this is indeed the number of binary words of length ≤ 16

```
sage: factor(131071 + 1)
2^17
```

  Note that the function mapped and reduced here are equivalent to the default values of the sage.combinat.backtrack.SearchForest.map_reduce() method so that to compute the number of element you only need to call:

```
sage: S.map_reduce()
131071
```

  You don't need to use RecursivelyEnumeratedSet(), you can use directly RESetMapReduce. This is needed if you want to have fine control over the parallel execution (see Advanced use below):

```
sage: from sage.parallel.map_reduce import RESetMapReduce
```

```
sage: S = RESetMapReduce(
....:  roots = [[]],
....:  children = lambda l: [l+[0], l+[1]] if len(l) <= 15 else [],
....:  map_function = lambda x : 1,
....:  reduce_function = lambda x,y: x+y,
....:  reduce_init = 0 )
sage: S.run()
131071
```

- **Generating series**: In this situation, the map function associates a monomial to each element of S, while the Reduce function is still equal to lambda x,y : x+y.

  Here's the Sage code for binary words of length ≤ 16

```
sage: S = RecursivelyEnumeratedSet(
....:  [[]], lambda l: [l+[0], l+[1]] if len(l) < 16 else [],
....:  structure='forest', enumeration='depth')
sage: sp = S.map_reduce(
....:  map_function = lambda z: x**len(z),
....:  reduce_function = lambda x,y: x+y,
....:  reduce_init = 0 )
sage: sp
65536*x^16 + 32768*x^15 + 16384*x^14 + 8192*x^13 + 4096*x^12 + 2048*x^11 + 1024*x^10 + 512*x^9 + 256*x^8 + 128*x^7 + 64*x^
```

  This is of course $\sum_{i=0}^{i=16} (2x)^i$:

```
sage: bool(sp == sum((2*x)^i for i in range(17)))
True
```

  Here is another example where we count permutations of size ≤ 8 (here we use the default values):

```
sage: S = RecursivelyEnumeratedSet( [[]],
....:  lambda l: ([l[:i] + [len(l)] + l[i:] for i in range(len(l)+1)]
....:            if len(l) < 8 else []),
....:  structure='forest', enumeration='depth')
sage: sp = S.map_reduce(lambda z: x**len(z)); sp
40320*x^8 + 5040*x^7 + 720*x^6 + 120*x^5 + 24*x^4 + 6*x^3 + 2*x^2 + x + 1
```

  This is of course $\sum_{i=0}^{i=8} i! x^i$:

```
sage: bool(sp == sum(factorial(i)*x^i for i in range(9)))
True
```

- **Post Processing**: We now demonstrate the use of post_process. We generate the permutation as previously, but we only perform the map/reduce computation on those of even len. Of course we get the even part of the previous generating series:

```
sage: S = RecursivelyEnumeratedSet( [[]],
....:  lambda l: ([l[:i] + [len(l)+1] + l[i:] for i in range(len(l)+1)]
....:            if len(l) < 8 else []),
....:  post_process = lambda l : l if len(l) % 2 == 0 else None,
....:  structure='forest', enumeration='depth')
sage: sp = S.map_reduce(lambda z: x**len(z)); sp
40320*x^8 + 720*x^6 + 24*x^4 + 2*x^2 + 1
```

  This is also useful for example to call a constructor on the generated elements:

```
sage: S = RecursivelyEnumeratedSet( [[]],
....:  lambda l: ([l[:i] + [len(l)+1] + l[i:] for i in range(len(l)+1)]
....:            if len(l) < 5 else []),
....:  post_process = lambda l : Permutation(l) if len(l) == 5 else None,
....:  structure='forest', enumeration='depth')
sage: sp = S.map_reduce(lambda z: x**(len(z.inversions()))); sp
x^10 + 4*x^9 + 9*x^8 + 15*x^7 + 20*x^6 + 22*x^5 + 20*x^4 + 15*x^3 + 9*x^2 + 4*x + 1
```

  We get here a polynomial called the x-factorial of 5 that is $\prod_{i=1}^{i=5} \frac{1-x^i}{1-x}$:

```
sage: (prod((1-x^i)/(1-x) for i in range(1,6))).simplify_rational()
x^10 + 4*x^9 + 9*x^8 + 15*x^7 + 20*x^6 + 22*x^5 + 20*x^4 + 15*x^3 + 9*x^2 + 4*x + 1
```

- **Listing the objects**: One can also compute the list of objects in a RecursivelyEnumeratedSet of forest type using RESetMapReduce. As an example, we compute the set of numbers between 1 and 63, generated by their binary expansion:

```
sage: S = RecursivelyEnumeratedSet( [1],
....:  lambda l: [(l<<1)|0, (l<<1)|1] if l < 1<<5 else [],
....:  structure='forest', enumeration='depth')
```

Here is the list computed without RESetMapReduce:

```
sage: serial = list(S)
sage: serial
[1, 2, 4, 8, 16, 32, 33, 17, 34, 35, 9, 18, 36, 37, 19, 38, 39, 5, 10, 20, 40, 41, 21, 42, 43, 11, 22, 44, 45, 23, 46, 47, 3, 6, 12, 24, 48, 49, 25, 50, 51, 13,
```

Here is how to perform the parallel computation. The order of the lists depends on the synchronisation of the various computation processes and therefore should be considered as random:

```
sage: parall = S.map_reduce( lambda x: [x], lambda x,y: x+y, [] )
sage: parall  # random
[1, 3, 7, 15, 31, 63, 62, 30, 61, 60, 14, 29, 59, 58, 28, 57, 56, 6, 13, 27, 55, 54, 26, 53, 52, 12, 25, 51, 50, 24, 49, 48, 2, 5, 11, 23, 47, 46, 22, 45, 44, 1
sage: sorted(serial) == sorted(parall)
True
```

Advanced use

Fine control of the execution of a map/reduce computations is obtained by passing parameters to the RESetMapReduce.run() method. One can use the three following parameters:

- max_proc – maximum number of process used. default: number of processor on the machine
- timeout – a timeout on the computation (default: None)
- reduce_locally – whether the workers should reduce locally their work or sends results to the master as soon as possible. See RESetMapReduceWorker for details.

Here is an example or how to deal with timeout:

```
sage: from sage.parallel.map_reduce import RESetMPExample, AbortError
sage: EX = RESetMPExample(maxl = 8)
sage: try:
....:     res = EX.run(timeout=0.01)
....: except AbortError:
....:     print("Computation timeout")
....: else:
....:     print("Computation normally finished")
....:     res
Computation timeout
```

The following should not timeout even on a very slow machine:

```
sage: try:
....:     res = EX.run(timeout=60)
....: except AbortError:
....:     print("Computation Timeout")
....: else:
....:     print("Computation normally finished")
....:     res
Computation normally finished
40320*x^8 + 5040*x^7 + 720*x^6 + 120*x^5 + 24*x^4 + 6*x^3 + 2*x^2 + x + 1
```

As for reduce_locally, one should not see any difference, except for speed during normal usage. Most of the time the user should leave it to True, unless he sets up a mecanism to consume the partial results as soon as they arrive. See RESetParallelIterator and in particular the __iter__ method for a example of consumer use.

Profiling

It is possible the profile a map/reduce computation. First we create a RESetMapReduce object:

```
sage: from sage.parallel.map_reduce import RESetMapReduce
sage: S = RESetMapReduce(
....:     roots = [[]],
....:     children = lambda l: [l+[0], l+[1]] if len(l) <= 15 else [],
....:     map_function = lambda x : 1,
....:     reduce_function = lambda x,y: x+y,
....:     reduce_init = 0 )
```

The profiling is activated by the profile parameter. The value provided should be a prefix (including a possible directory) for the profile dump:

```
sage: prof = tmp_dir('RESetMR_profile')+'profcomp'
sage: res = S.run(profile=prof) # random
[RESetMapReduceWorker-1:58] (20:00:41.444) Profiling in /home/user/.sage/temp/mymachine.mysite/32414/RESetMR_profilewRCRAx/profcomp1 ...
```

```
...
[RESetMapReduceWorker-1:57] (20:00:41.444) Profiling in /home/user/.sage/temp/mymachine.mysite/32414/RESetMR_profilewRCRAx/profcomp0 ...
sage: res
131071
```

In this example, the profile have been dumped in files such as profcomp0. One can then load and print them as follows. See profile.profile for more details:

```
sage: import cProfile, pstats
sage: st = pstats.Stats(prof+'0')
sage: st.strip_dirs().sort_stats('cumulative').print_stats() #random
...
   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1   0.023    0.023    0.432    0.432 map_reduce.py:1211(run_myself)
    11968   0.151    0.000    0.223    0.000 map_reduce.py:1292(walk_branch_locally)
...
<pstats.Stats instance at 0x7fedea40c6c8>
```

See also: The Python Profilers for more detail on profiling in python.

Logging

The computation progress is logged through a logging.Logger in sage.parallel.map_reduce.logger together with logging.StreamHandler and a logging.Formatter. They are currently configured to print warning message on the console.

See also: Logging facility for Python for more detail on logging and log system configuration.

Note: Calls to logger which involve printing the node are commented out in the code, because the printing (to a string) of the node can be very time consuming depending on the node and it happens before the decision whether the logger should record the string or drop it.

How does it work ?

The scheduling algorithm we use here is any adaptation of Wikipedia article Work_stealing:

> In a work stealing scheduler, each processor in a computer system has a queue of work items (computational tasks, threads) to perform. [...]. Each work items are initially put on the queue of the processor executing the work item. When a processor runs out of work, it looks at the queues of other processors and "steals" their work items. In effect, work stealing distributes the scheduling work over idle processors, and as long as all processors have work to do, no scheduling overhead occurs.

For communication we use Python's basic multiprocessing module. We first describe the different actors and communications tools used by the system. The work is done under the coordination of a **master** object (an instance of RESetMapReduce) by a bunch of **worker** objects (instances of RESetMapReduceWorker).

Each running map reduce instance work on a RecursivelyEnumeratedSet of forest type called here C and is coordinated by a RESetMapReduce object called the **master**. The master is in charge of lauching the work, gathering the results and cleaning up at the end of the computation. It doesn't perform any computation associated to the generation of the element C nor the computation of the mapped function. It however occasionally perform a reduce, but most reducing is by default done by the workers. Also thanks to the work-stealing algorithm, the master is only involved in detecting the termination of the computation but all the load balancing is done at the level of the worker.

Workers are instance of RESetMapReduceWorker. They are responsible of doing the actual computations: elements generation, mapping and reducing. They are also responsible of the load balancing thanks to work-stealing.

Here is a description of the attribute of the **master** relevant to the map-reduce protocol:

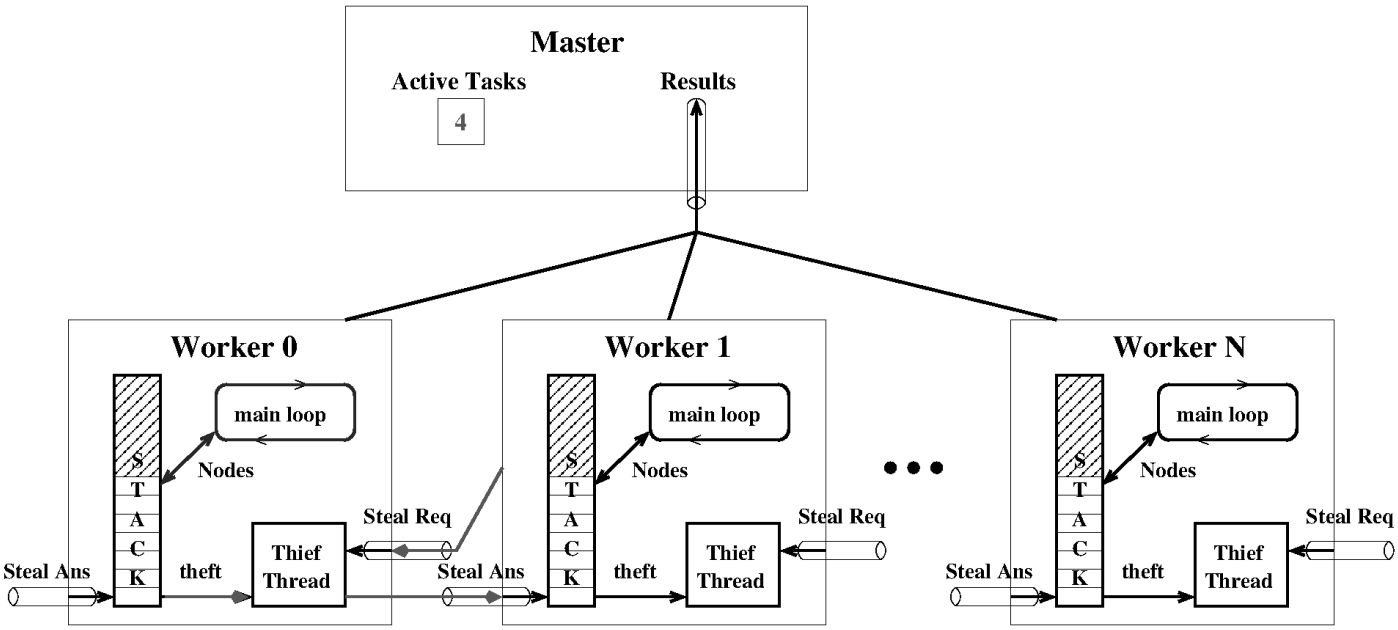
- master._results – a SimpleQueue where the master gathers the results sent by the workers.
- master._active_tasks – a Semaphore recording the number of active task. The work is done when it gets to 0.
- master._done – a Lock which ensures that shutdown is done only once.
- master._abort – a Value() storing a shared ctypes.c_bool which is True if the computation was aborted before all the worker runs out of work.
- master._workers – a list of RESetMapReduceWorker objects. Each worker is identified by its position in this list.

Each worker is a process (RESetMapReduceWorker inherits from Process) which contains:

- worker._iproc – the identifier of the worker that is its position in the master's list of workers

- worker._todo – a collections.deque storing of nodes of the worker. It is used as a stack by the worker. Thiefs steal from the bottom of this queue.
- worker._request – a SimpleQueue storing steal request submitted to worker.
- worker._read_task, worker._write_task – a Pipe used to transfert node during steal.
- worker._thief – a Thread which is in charge of stealing from worker._todo.

Here is a schematic of the architecture:



How thefts are performed

During normal time, that is when all worker are active) a worker W is iterating though a loop inside RESetMapReduceWorker.walk_branch_locally(). Work nodes are taken from and new nodes W._todo are appended to W._todo. When a worker W is running out of work, that is worker._todo is empty, then it tries to steal some work (ie: a node) from another worker. This is performed in the RESetMapReduceWorker.steal() method.

From the point of view of W here is what happens:

- W signals to the master that it is idle master._signal_task_done();
- W chose a victim V at random;
- W sends a request to V : it puts its identifier into V._request;
- W tries to read a node from W._read_task. Then three things may happen:
  - a proper node is read. Then the theft was a success and W starts working locally on the received node.
  - None is received. This means that V was idle. Then W tries another victim.
  - AbortError is received. This means either that the computation was aborted or that it simply succeded and that no more work is required by W. Therefore an AbortError exception is raised leading to W to shutdown.

We now describe the protocol on the victims side. Each worker process contains a Thread which we call T for thief which acts like some kinds of Troyan horse during theft. It is normally blocked waiting for a steal request.

From the point of view of V and T, here is what happens:

- during normal time T is blocked waiting on V._request;
- upon steal request, T wakes up receiving the identification of W;
- T signal to the master that a new task is starting by master._signal_task_start();
- Two things may happen depending if the queue V._todo is empty or not. Remark that due to the GIL, there is no parallel execution between the victim V and its thief tread T.
  - If V._todo is empty, then None is answered on W._write_task. The task is immediately signaled to end the the master through master._signal_task_done().
  - Otherwise, a node is removed from the bottom of V._todo. The node is sent to W on W._write_task. The task will be ended by W, that is when finished working on the subtree rooted at the node, W will call master._signal_task_done().

The end of the computation

To detect when a computation is finished, we keep a synchronized integer which count the number of active task. This is essentially a semaphore but semaphore are broken on Darwin's OSes so we ship two implementations depending on the os (see

ActiveTaskCounter and ActiveTaskCounterDarwin and note below).

When a worker finishes working on a task, it calls master._signal_task_done(). This decrease the task counter master._active_tasks. When it reaches 0, it means that there are no more nodes: the work is done. The worker executes master._shutdown() which sends AbortError on all worker._request() and worker._write_task() Queues. Each worker or thief thread receiving such a message raise the corresponding exception, stoping therefore its work. A lock called master._done ensures that shutdown is only done once.

Finally, it is also possible to interrupt the computation before its ends calling master.abort(). This is done by putting master._active_tasks to 0 and calling master._shutdown().

Warning: The MacOSX Semaphore bug

Darwin's OSes do not correctly implement POSIX's semaphore semantic. Indeed, on this system, acquire may fail and return False not only because the semaphore is equal to zero but also **because someone else is trying to acquire** at the same time. This renders the usage of Semaphore impossible on MacOSX so that on this system we use a synchronized integer.

Are there examples of classes ?

Yes ! Here, there are:

- RESetMPExample – a simple basic example
- RESetParallelIterator – a more advanced example using non standard communication configuration.

Tests

Generating series for sum of strictly decreasing list of integer smaller than 15:

```
sage: y = polygen(ZZ, 'y')
sage: R = RESetMapReduce(
....:     roots = [([], 0, 0)] +[([i], i, i) for i in range(1,15)],
....:     children = lambda list_sum_last:
....:         [(list_sum_last[0] + [i], list_sum_last[1] + i, i) for i in range(1, list_sum_last[2])],
....:     map_function = lambda li_sum_dummy: y**li_sum_dummy[1])
sage: sg = R.run()
sage: bool(sg == expand(prod((1+y^i) for i in range(1,15))))
True
```

Classes and methods

exception sage.parallel.map_reduce.**AbortError**

Bases: exceptions.Exception

Exception for aborting parallel computations

This is used both as exception or as abort message

TESTS:

```
sage: from sage.parallel.map_reduce import AbortError
sage: raise AbortError
Traceback (most recent call last):
...
AbortError
```

sage.parallel.map_reduce.**ActiveTaskCounter**

alias of ActiveTaskCounterPosix

class sage.parallel.map_reduce.**ActiveTaskCounterDarwin**(*task_number*)

Bases: object

Handling the number of Active Tasks

A class for handling the number of active task in distributed computation process. This is essentially a semaphore, but Darwin's OSes do not correctly implement POSIX's semaphore semantic. So we use a shared integer with a lock.

abort()

Set the task counter to 0.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import ActiveTaskCounterDarwin as ATC
sage: c = ATC(4); c
ActiveTaskCounter(value=4)
sage: c.abort()
sage: c
ActiveTaskCounter(value=0)
```

task_done()

Decrement the task counter by one.

OUTPUT:

Calling `task_done()` decrement the counter and returns its value after the decrementation.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import ActiveTaskCounterDarwin as ATC
sage: c = ATC(4); c
ActiveTaskCounter(value=4)
sage: c.task_done()
3
sage: c
ActiveTaskCounter(value=3)

sage: c = ATC(0)
sage: c.task_done()
-1
```

task_start()

Increment the task counter by one.

OUTPUT:

Calling `task_start()` on a zero or negative counter returns 0, otherwise increment the counter and returns its value after the incrementation.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import ActiveTaskCounterDarwin as ATC
sage: c = ATC(4); c
ActiveTaskCounter(value=4)
sage: c.task_start()
5
sage: c
ActiveTaskCounter(value=5)
```

Calling `task_start()` on a zero counter does nothing:

```
sage: c = ATC(0)
sage: c.task_start()
0
sage: c
ActiveTaskCounter(value=0)
```

class sage.parallel.map_reduce.**ActiveTaskCounterPosix**(*task_number*)

Bases: object

Handling the number of Active Tasks

A class for handling the number of active task in distributed computation process. This is the standard implementation on POSIX compliant OSes. We essentially wrap a semaphore.

Note: A legitimate question is whether there is a need in keeping the two implementations. I ran the following experiment on my machine:

```
S = RecursivelyEnumeratedSet( [[]],
      lambda l: ([l[:i] + [len(l)] + l[i:] for i in range(len(l)+1)]
        if len(l) < NNN else []),
    structure='forest', enumeration='depth')
%time sp = S.map_reduce(lambda z: x**len(z)); sp
```

For NNN = 10, averaging a dozen of runs, I got:

- Posix complient implementation : 17.04 s
- Darwin's implementation : 18.26 s

So there is a non negligible overhead. It will probably be worth if we tries to Cythonize the code. So I'm keeping both implementation.

abort()

Set the task counter to 0.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import ActiveTaskCounter as ATC
sage: c = ATC(4); c
ActiveTaskCounter(value=4)
sage: c.abort()
sage: c
ActiveTaskCounter(value=0)
```

task_done()

Decrement the task counter by one.

OUTPUT:

Calling `task_done()` decrement the counter and returns its value after the decrementation.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import ActiveTaskCounter as ATC
sage: c = ATC(4); c
ActiveTaskCounter(value=4)
sage: c.task_done()
3
sage: c
ActiveTaskCounter(value=3)

sage: c = ATC(0)
sage: c.task_done()
-1
```

task_start()

Increment the task counter by one.

OUTPUT:

Calling `task_start()` on a zero or negative counter returns 0, otherwise increment the counter and returns its value after the incrementation.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import ActiveTaskCounterDarwin as ATC
sage: c = ATC(4); c
ActiveTaskCounter(value=4)
sage: c.task_start()
5
sage: c
ActiveTaskCounter(value=5)
```

Calling `task_start()` on a zero counter does nothing:

```
sage: c = ATC(0)
sage: c.task_start()
0
sage: c
ActiveTaskCounter(value=0)
```

class sage.parallel.map_reduce.**RESetMPExample**(*maxl=9*)

Bases: sage.parallel.map_reduce.RESetMapReduce

An example of map reduce class

INPUT:

- maxl – the maximum size of permutations generated (default to 9).

This compute the generating series of permutations counted by their size upto size maxl.

EXAMPLE:

```
sage: from sage.parallel.map_reduce import RESetMPExample
sage: EX = RESetMPExample()
sage: EX.run()
362880*x^9 + 40320*x^8 + 5040*x^7 + 720*x^6 + 120*x^5 + 24*x^4 + 6*x^3 + 2*x^2 + x + 1
```

See also: This is an example of RESetMapReduce

children(*l*)

Return the children of the permutation l.

INPUT:

- l – a list containing a permutation

OUTPUT:

the lists of len(l) inserted at all possible positions into l

EXAMPLE:

```
sage: from sage.parallel.map_reduce import RESetMPExample
sage: RESetMPExample().children([1,0])
[[2, 1, 0], [1, 2, 0], [1, 0, 2]]
```

map_function(*l*)

The monomial associated to the permutation l

INPUT:

- l – a list containing a permutation

OUTPUT:

x^len(l).

EXAMPLE:

```
sage: from sage.parallel.map_reduce import RESetMPExample
sage: RESetMPExample().map_function([1,0])
x^2
```

roots()

Return the empty permutation

EXAMPLE:

```
sage: from sage.parallel.map_reduce import RESetMPExample
sage: RESetMPExample().roots()
[[]]
```

class sage.parallel.map_reduce.**RESetMapReduce**(*roots=None*, *children=None*, *post_process=None*, *map_function=None*, *reduce_function=None*, *reduce_init=None*, *forest=None*)

Bases: object

Map-Reduce on recursively enumerated sets

INPUT:

Description of the set:

- either forest=f – where f is a RecursivelyEnumeratedSet of forest type
- or a triple roots, children, post_process as follows
  - roots=r – The root of the enumeration
  - children=c – a function iterating through children node, given a parent nodes
  - post_process=p – a post processing function

The option post_process allows for customizing the nodes that are actually produced. Furthermore, if post_process(x) returns None, then x won't be output at all.

Decription of the map/reduce operation:

- map_function=f – (default to None)
- reduce_function=red – (default to None)
- reduce_init=init – (default to None)

See also: the Map/Reduce module for details and examples.

abort()

Abort the current parallel computation

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetParallelIterator
sage: S = RESetParallelIterator( [[]],
.....:   lambda l: [l+[0], l+[1]] if len(l) < 17 else [])
sage: it = iter(S)
sage: next(it)
[]
sage: S.abort()
sage: hasattr(S, 'work_queue')
False
```

Cleanups:

```
sage: S.finish()
```

finish()

Destroys the worker and all the communication objects.

Also gathers the communication statistics before destroying the workers.

TESTS:

```
sage: from sage.parallel.map_reduce import RESetMPExample
sage: S = RESetMPExample(maxl=5)
sage: S.setup_workers(2) # indirect doctest
sage: S._workers[0]._todo.append([])
sage: for w in S._workers: w.start()
sage: _ = S.get_results()
sage: S._shutdown()
sage: S.print_communication_statistics()
Traceback (most recent call last):
...
AttributeError: 'RESetMPExample' object has no attribute '_stats'

sage: S.finish()

sage: S.print_communication_statistics()
#proc: ...
...

sage: _ = S.run() # Cleanup
```

See also: print_communication_statistics()

get_results()

Get the results from the queue

OUTPUT:

the reduction of the results of all the workers, that is the result of the map/reduce computation.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMapReduce
sage: S = RESetMapReduce()
sage: S.setup_workers(2)
sage: for v in [1, 2, None, 3, None]: S._results.put(v)
sage: S.get_results()
6
```

Cleanups:

```
sage: del S._results, S._active_tasks, S._done, S._workers
```

map_function(*o*)

Return the function mapped by self

INPUT:

- o – a node

OUTPUT:

By default 1.

Note: This should be overloaded in applications.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMapReduce
sage: S = RESetMapReduce()
sage: S.map_function(7)
1
sage: S = RESetMapReduce(map_function = lambda x: 3*x + 5)
sage: S.map_function(7)
26
```

post_process(*a*)

Return the post-processing function for self

INPUT: a – a node

By default, returns a itself

Note: This should be overloaded in applications.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMapReduce
sage: S = RESetMapReduce()
sage: S.post_process(4)
4
sage: S = RESetMapReduce(post_process=lambda x: x*x)
sage: S.post_process(4)
16
```

print_communication_statistics(*blocksize=16*)

Print the communication statistics in a nice way

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMPExample
sage: S = RESetMPExample(maxl=6)
sage: S.run()
720*x^6 + 120*x^5 + 24*x^4 + 6*x^3 + 2*x^2 + x + 1

sage: S.print_communication_statistics()   # random
#proc:      0   1   2   3   4   5   6   7
reqs sent:   5   2   3  11  21  19   1   0
reqs rcvs:  10  10   9   5   1  11   9   2
- thefs:     1   0   0   0   0   0   0   0
+ thefs:     0   0   1   0   0   0   0   0
```

random_worker()

Returns a random workers

OUTPUT:

A worker for self chosed at random

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMPExample, RESetMapReduceWorker
sage: from threading import Thread
sage: EX = RESetMPExample(maxl=6)
sage: EX.setup_workers(2)
sage: EX.random_worker()
```

```
<RESetMapReduceWorker(RESetMapReduceWorker-..., initial)>
sage: EX.random_worker() in EX._workers
True
```

Cleanups:

```
sage: del EX._results, EX._active_tasks, EX._done, EX._workers
```

reduce_function(*a*, *b*)

Return the reducer function for self

INPUT:

- a, b – two value to be reduced

OUTPUT:

by default the sum of a and b.

Note: This should be overloaded in applications.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMapReduce
sage: S = RESetMapReduce()
sage: S.reduce_function(4, 3)
7
sage: S = RESetMapReduce(reduce_function=lambda x,y: x*y)
sage: S.reduce_function(4, 3)
12
```

reduce_init()

Return the initial element for a reduction

Note: This should be overloaded in applications.

TESTS:

```
sage: from sage.parallel.map_reduce import RESetMapReduce
sage: S = RESetMapReduce()
sage: S.reduce_init()
0
sage: S = RESetMapReduce(reduce_init = 2)
sage: S.reduce_init()
2
```

roots()

Return the roots of self

OUTPUT:

an iterable of nodes

Note: This should be overloaded in applications.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMapReduce
sage: S = RESetMapReduce(42)
sage: S.roots()
42
```

run(*max_proc=None*, *reduce_locally=True*, *timeout=None*, *profile=None*)

Run the computations

INPUT:

- max_proc – maximum number of process used. default: number of processor on the machine
- reduce_locally – See RESetMapReduceWorker (default: True)
- timeout – a timeout on the computation (default: None)
- profile – directory/filename prefix for profiling, or None for no profiling (default: None)

OUTPUT:

the result of the map/reduce computation or an exception AbortError if the computation was interrupted or timeout.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMPExample
sage: EX = RESetMPExample(maxl = 8)
sage: EX.run()
40320*x^8 + 5040*x^7 + 720*x^6 + 120*x^5 + 24*x^4 + 6*x^3 + 2*x^2 + x + 1
```

Here is an example or how to deal with timeout:

```
sage: from sage.parallel.map_reduce import AbortError
sage: try:
....:     res = EX.run(timeout=0.01)
....: except AbortError:
....:     print("Computation timeout")
....: else:
....:     print("Computation normally finished")
....:     res
Computation timeout
```

The following should not timeout even on a very slow machine:

```
sage: from sage.parallel.map_reduce import AbortError
sage: try:
....:     res = EX.run(timeout=60)
....: except AbortError:
....:     print("Computation Timeout")
....: else:
....:     print("Computation normally finished")
....:     res
Computation normally finished
40320*x^8 + 5040*x^7 + 720*x^6 + 120*x^5 + 24*x^4 + 6*x^3 + 2*x^2 + x + 1
```

run_serial()

Serial run of the computation (mostly for tests)

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMPExample
sage: EX = RESetMPExample(maxl = 4)
sage: EX.run_serial()
24*x^4 + 6*x^3 + 2*x^2 + x + 1
```

setup_workers(*max_proc=None*, *reduce_locally=True*)

Setup the communication channels

INPUT:

- mac_proc – an integer: the maximum number of workers
- reduce_locally – whether the workers should reduce locally their work or sends results to the master as soon as possible. See RESetMapReduceWorker for details.

TESTS:

```
sage: from sage.parallel.map_reduce import RESetMapReduce
sage: S = RESetMapReduce()
sage: S.setup_workers(2)
sage: S._results
<multiprocessing.queues.SimpleQueue object at 0x...>
sage: len(S._workers)
2
```

start_workers()

Lauch the workers

The worker should have been created using setup_workers().

TESTS:

```
sage: from sage.parallel.map_reduce import RESetMapReduce
sage: S = RESetMapReduce(roots=[])
sage: S.setup_workers(2)
sage: S.start_workers()
```

```
sage: all(w.is_alive() for w in S._workers)
True

sage: sleep(1)
sage: all(not w.is_alive() for w in S._workers)
True
```

Cleanups:

```
sage: S.finish()
```

class sage.parallel.map_reduce.**RESetMapReduceWorker**(*mapred*, *iproc*, *reduce_locally*)

Bases: multiprocessing.process.Process

Worker for generate-map-reduce

This shouldn't be called directly, but instead created by RESetMapReduce.setup_workers().

INPUT:

- mapred – the instance of RESetMapReduce for which this process is working.
- iproc – the id of this worker.
- reduce_locally – when reducing the results. Three possible values are supported:
  - True – means the reducing work is done all locally, the result is only sent back at the end of the work. This ensure the lowest level of communication.
  - False – results are sent back after each finished branches, when the process is asking for more work.

run()

The main function executed by the worker

Calls run_myself() after possibly setting up parallel profiling.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMPExample, RESetMapReduceWorker
sage: EX = RESetMPExample(maxl=6)
sage: EX.setup_workers(1)

sage: w = EX._workers[0]
sage: w._todo.append(EX.roots()[0])

sage: w.run()
sage: sleep(1)
sage: w._todo.append(None)

sage: EX.get_results()
720*x^6 + 120*x^5 + 24*x^4 + 6*x^3 + 2*x^2 + x + 1
```

Cleanups:

```
sage: del EX._results, EX._active_tasks, EX._done, EX._workers
```

run_myself()

The main function executed by the worker

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMPExample, RESetMapReduceWorker
sage: EX = RESetMPExample(maxl=6)
sage: EX.setup_workers(1)

sage: w = EX._workers[0]
sage: w._todo.append(EX.roots()[0])
sage: w.run_myself()

sage: sleep(1)
sage: w._todo.append(None)

sage: EX.get_results()
720*x^6 + 120*x^5 + 24*x^4 + 6*x^3 + 2*x^2 + x + 1
```

Cleanups:

```
sage: del EX._results, EX._active_tasks, EX._done, EX._workers
```

send_partial_result()

Send results to the MapReduce process

Send the result stored in self._res to the master an reinitialize it to master.reduce_init.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMPExample, RESetMapReduceWorker
sage: EX = RESetMPExample(maxl=4)
sage: EX.setup_workers(1)
sage: w = EX._workers[0]
sage: w._res = 4
sage: w.send_partial_result()
sage: w._res
0
sage: EX._results.get()
4
```

steal()

Steal some node from another worker

OUTPUT:

a node stolen from another worker choosed at random

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMPExample, RESetMapReduceWorker
sage: from threading import Thread
sage: EX = RESetMPExample(maxl=6)
sage: EX.setup_workers(2)

sage: w0, w1 = EX._workers
sage: w0._todo.append(42)
sage: thief0 = Thread(target = w0._thief, name="Thief")
sage: thief0.start()

sage: w1.steal()
42
```

walk_branch_locally(*node*)

Work locally

Performs the map/reduce computation on the subtrees rooted at node.

INPUT:

- node – the root of the subtree explored.

OUTPUT:

nothing, the result are stored in self._res

This is where the actual work is performed.

EXAMPLES:

```
sage: from sage.parallel.map_reduce import RESetMPExample, RESetMapReduceWorker
sage: EX = RESetMPExample(maxl=4)
sage: w = RESetMapReduceWorker(EX, 0, True)
sage: def sync(): pass
sage: w.synchronize = sync
sage: w._res = 0

sage: w.walk_branch_locally([])
sage: w._res
x^4 + x^3 + x^2 + x + 1

sage: w.walk_branch_locally(w._todo.pop())
sage: w._res
2*x^4 + x^3 + x^2 + x + 1

sage: while True: w.walk_branch_locally(w._todo.pop())
Traceback (most recent call last):
...
IndexError: pop from an empty deque
sage: w._res
24*x^4 + 6*x^3 + 2*x^2 + x + 1
```

class sage.parallel.map_reduce.**RESetParallelIterator**(*roots=None*, *children=None*, *post_process=None*, *map_function=None*, *reduce_function=None*, *reduce_init=None*, *forest=None*)

Bases: sage.parallel.map_reduce.RESetMapReduce

A parallel iterator for recursively enumerated sets

This demonstrate how to use RESetMapReduce to get an iterator on a recursively enumerated sets for which the computations are done in parallel.

EXAMPLE:

```
sage: from sage.parallel.map_reduce import RESetParallelIterator
sage: S = RESetParallelIterator( [[]],
....:   lambda l: [l+[0], l+[1]] if len(l) < 15 else [])
sage: sum(1 for _ in S)
65535
```

map_function(*z*)

Return a singleton tuple

INPUT: z – a node

OUTPUT: (z,)

EXAMPLE:

```
sage: from sage.parallel.map_reduce import RESetParallelIterator
sage: S = RESetParallelIterator( [[]],
....:   lambda l: [l+[0], l+[1]] if len(l) < 15 else [])
sage: S.map_function([1, 0])
([1, 0],)
```

sage.parallel.map_reduce.**proc_number**(*max_proc=None*)

Computing the number of process used

INPUT:

- max_proc – the maximum number of process used

EXAMPLE:

```
sage: from sage.parallel.map_reduce import proc_number
sage: proc_number() # random
8
sage: proc_number(max_proc=1)
1
sage: proc_number(max_proc=2) in (1, 2)
True
```