Ruby - Bug #4525

Exponential performance when summing Enumerable

03/25/2011 08:09 PM - Confusion (Ivo Wever)

Status:	Closed		
Priority:	Normal		
Assignee:	ko1 (Koichi Sasada)		
Target version:	2.0.0		
ruby -v:	ruby 1.9.2p180 (2011-02-18 revision 30909) [x86_64-linux]	**Backport:**	

Description

=begin
When you sum an Enumerable (using .inject(:+) or in a more verbose fashion), 1.8.7 and 1.9.2 show exponential performance, where JRuby and Rubinius show the expected linear behavior.

```
class A
attr_accessor :foo
def initialize(foo)
@foo = foo
end

def +(other)
A.new(self.foo + other.foo)
end
end

(10..15).each do |factor|
start = Time.now
([A.new(2)] * 2**factor * 100).inject(:+)
puts "#{2**factor * 100} took #{(Time.now - start)} s."
end
```

Outcomes (I'm not interested in the absolute timings: note the factors between each step):
ruby 1.8.7 (2011-02-18 patchlevel 334) [x86_64-linux], MBARI 0x6770, Ruby Enterprise Edition 2011.03

```
102400 took 0.151034 s.
204800 took 0.345033 s.
409600 took 0.920415 s.
819200 took 2.086751 s.
1638400 took 5.727894 s.
3276800 took 18.041329 s.
```

ruby 1.9.2p180 (2011-02-18 revision 30909) [x86_64-linux]

```
102400 took 0.074562689 s.
204800 took 0.164413421 s.
409600 took 0.44690715 s.
819200 took 1.458326897 s.
1638400 took 3.215625728 s.
3276800 took 9.992734203 s.
```

jruby 1.5.6 (ruby 1.8.7 patchlevel 249) (2010-12-03 9cf97c3) (Java HotSpot(TM) 64-Bit Server VM 1.6.0_20) [amd64-java]

```
102400 took 0.705 s.
204800 took 0.206 s.
409600 took 0.222 s.
819200 took 0.37 s.
1638400 took 0.777 s.
3276800 took 1.441 s.
```

rubinius 1.2.4dev (1.8.7 9d6719d4 yyyy-mm-dd JI) [x86_64-unknown-linux-gnu]

```
102400 took 0.132361 s.
204800 took 0.138144 s.
409600 took 0.324924 s.
```

819200 took 0.54765 s.
1638400 took 1.0541179999999999 s.
3276800 took 2.2074 s.

=end

History

#1 - 03/25/2011 08:12 PM - Confusion (Ivo Wever)

=begin
Shoot, misclicked the preview and no ability to edit original submission? A better formatted version:

When you sum an Enumerable (using .inject(:+) or in a more verbose fashion), 1.8.7 and 1.9.2 show exponential performance, where JRuby and Rubinius show the expected linear behavior.

```
class A
attr_accessor :foo
def initialize(foo)
@foo = foo
end

def +(other)
A.new(self.foo + other.foo)
end
end

(10..15).each do |factor|
start = Time.now
([A.new(2)] * 2**factor * 100).inject(:+)**
**puts "#{2**factor * 100} took #{(Time.now - start)} s."
end
```

Outcomes (I'm not interested in the absolute timings: note the factors between each step):

ruby 1.8.7 (2011-02-18 patchlevel 334) [x86_64-linux], MBARI 0x6770, Ruby Enterprise Edition 2011.03

102400 took 0.151034 s.
204800 took 0.345033 s.
409600 took 0.920415 s.
819200 took 2.086751 s.
1638400 took 5.727894 s.
3276800 took 18.041329 s.

ruby 1.9.2p180 (2011-02-18 revision 30909) [x86_64-linux]
102400 took 0.074562689 s.
204800 took 0.164413421 s.
409600 took 0.44690715 s.
819200 took 1.458326897 s.
1638400 took 3.215625728 s.
3276800 took 9.992734203 s.

jruby 1.5.6 (ruby 1.8.7 patchlevel 249) (2010-12-03 9cf97c3) (Java HotSpot(TM) 64-Bit Server VM 1.6.0_20) [amd64-java]
102400 took 0.705 s.
204800 took 0.206 s.
409600 took 0.222 s.
819200 took 0.37 s.
1638400 took 0.777 s.
3276800 took 1.441 s.

rubinius 1.2.4dev (1.8.7 9d6719d4 yyyy-mm-dd JI) [x86_64-unknown-linux-gnu]
102400 took 0.132361 s.
204800 took 0.138144 s.
409600 took 0.324924 s.
819200 took 0.54765 s.
1638400 took 1.0541179999999999 s.
3276800 took 2.2074 s.

=end

#2 - 06/26/2011 06:41 PM - naruse (Yui NARUSE)

- Status changed from Open to Assigned

- Assignee set to ko1 (Koichi Sasada)

- Target version set to 2.0.0

#3 - 03/06/2012 04:19 PM - marcandre (Marc-Andre Lafortune)

- Status changed from Assigned to Closed

Hi,

Ivo Wever wrote:

> When you sum an Enumerable (using .inject(:+) or in a more verbose fashion), 1.8.7 and 1.9.2 show exponential performance, where JRuby and Rubinius show the expected linear behavior.

This is unrelated to inject and won't be the case in the case of inject not creating intermediate objects. Also the progression is not exponential but quadratic.

You results are a direct consequence of the type of garbage collector and memory allocation that CRuby uses. I wrote some code at the end to show this and you can also verify it by adding GC.disable at the beginning of your script.

Indeed, your test is equivalent to:
*) Allocate a big array
*) Time how long it takes to allocate new objects

The bigger array, the slower it takes CRuby to garbage collect as it will have to scan the array. JRuby and rbx have generational garbage collectors and will quickly stop checking this array. Maybe their allocator doesn't try to garbage collect in this case too, I don't know.

I don't think this bug report adds anything new to the known weaknesses of the current garbage collector and there are other issues opened to improve it (e.g. #4990), so I'll close this.

Here's some simplified code to reproduce the performance factors you are seeing and that show how the array allocation at the beginning (with GC enabled) is what is causing it. You'll get factors increasing from 2 to something < 4 in the first test (like you got in your script), but the two other tests will generate numbers close to 2.

```
require 'benchmark'
def time_factors
  timings = (10..15).map do |factor|
    yield n = 2 **factor * 100
    Benchmark.realtime{ n.times{ Object.new } }
  end
  timings.each_cons(2).map{ |a, b| b/a }
end

hold = nil
puts "With variable nb of objects allocated: " + time_factors{|n| hold = Array.new(n)}.to_s

puts "With fixed nb of objects allocated: " + time_factors{}.to_s

hold = nil; GC.start  # start fresh
GC.disable
puts "With variable nb of objects allocated (no GC): " + time_factors{|n| hold = Array.new(n)}.to_s
```