Ruby - Bug #17497

Ractor performance issue

12/31/2020 09:48 PM - marcandre (Marc-Andre Lafortune)

Status:	Closed		
Priority:	Normal		
Assignee:	ko1 (Koichi Sasada)		
Target version:			
ruby -v:	ruby 3.0.0p0 (2020-12-25 revision 95aff21468) [x86_64-darwin18]	**Backport:**	2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN, 3.0: DONE

Description

There's a strange performance issue with Ractor (at least on MacOS, didn't run on other OS).

I ran a benchmark doing 3 different types of work:

- "fib": method calls (naive fibonacci calculation)
- "cpu": (0...1000).inject(:+)
- "sleep": call sleep

I get the kind of results I was excepting for the fib and for sleeping, but the results for the "cpu" workload show a problem.

It is so slow that my pure Ruby backport (using Threads) is 65x faster on my Mac Pro (despite having 6 cores). Expected results would be 6x slower, so in that case Ractor is 400x slower than it should

On my MacBook (2 cores) the results are not as bad, the cpu workload is 3x faster with my pure-Ruby backport (only) instead of ~2x slower, so the factor is 6x too slow.

```
$ gem install backports
Successfully installed backports-3.20.0
1 gem installed
$ ruby ractor_test.rb
<internal:ractor>:267: warning: Ractor is experimental, and the behavior may change in future vers
ions of Ruby! Also there are many implementation issues.
fib: 110 ms  |  cpu: 22900 ms  |  sleep: 206 ms
$ B=t ruby ractor_test.rb
Using pure Ruby implementation
fib: 652 ms  |  cpu: 337 ms  |  sleep: 209 ms
```

Notice the sleep run takes similar time, which is good, and fib is ~6x faster on my 6-core CPU (and ~2x faster on my 2-core MacBook), again that's good as the pure ruby version uses Threads and thus runs with a single GVL.

The cpu version is the problem.

Script is here: https://gist.github.com/marcandre/bfed626e538a3d0fc7cad38dc026cf0e

Associated revisions

Revision 1ecda213668644d656eb0d60654737482447dd92 - 01/29/2021 07:22 AM - ko1 (Koichi Sasada)

global call-cache cache table for rb_funcall*

rb_funcall* (rb_funcall(), rb_funcallv(), ...) functions invokes
Ruby's method with given receiver. Ruby 2.7 introduced inline method
cache with static memory area. However, Ruby 3.0 reimplemented the
method cache data structures and the inline cache was removed.

Without inline cache, rb_funcall* searched methods everytime.
Most of cases per-Class Method Cache (pCMC) will be helped but
pCMC requires VM-wide locking and it hurts performance on
multi-Ractor execution, especially all Ractors calls methods
with rb_funcall*.

This patch introduced Global Call-Cache Cache Table (gccct) for
rb_funcall*. Call-Cache was introduced from Ruby 3.0 to manage
method cache entry atomically and gccct enables method-caching

without VM-wide locking. This table solves the performance issue
on multi-ractor execution.
[Bug #17497]

Ruby-level method invocation does not use gccct because it has
inline-method-cache and the table size is limited. Basically
rb_funcall* is not used frequently, so 1023 entries can be enough.
We will revisit the table size if it is not enough.

Revision 1ecda213668644d656eb0d60654737482447dd92 - 01/29/2021 07:22 AM - ko1 (Koichi Sasada)

global call-cache cache table for rb_funcall*

rb_funcall* (rb_funcall(), rb_funcallv(), ...) functions invokes
Ruby's method with given receiver. Ruby 2.7 introduced inline method
cache with static memory area. However, Ruby 3.0 reimplemented the
method cache data structures and the inline cache was removed.

Without inline cache, rb_funcall* searched methods everytime.
Most of cases per-Class Method Cache (pCMC) will be helped but
pCMC requires VM-wide locking and it hurts performance on
multi-Ractor execution, especially all Ractors calls methods
with rb_funcall*.

This patch introduced Global Call-Cache Cache Table (gccct) for
rb_funcall*. Call-Cache was introduced from Ruby 3.0 to manage
method cache entry atomically and gccct enables method-caching
without VM-wide locking. This table solves the performance issue
on multi-ractor execution.
[Bug #17497]

Ruby-level method invocation does not use gccct because it has
inline-method-cache and the table size is limited. Basically
rb_funcall* is not used frequently, so 1023 entries can be enough.
We will revisit the table size if it is not enough.

Revision 1ecda213 - 01/29/2021 07:22 AM - ko1 (Koichi Sasada)

global call-cache cache table for rb_funcall*

rb_funcall* (rb_funcall(), rb_funcallv(), ...) functions invokes
Ruby's method with given receiver. Ruby 2.7 introduced inline method
cache with static memory area. However, Ruby 3.0 reimplemented the
method cache data structures and the inline cache was removed.

Without inline cache, rb_funcall* searched methods everytime.
Most of cases per-Class Method Cache (pCMC) will be helped but
pCMC requires VM-wide locking and it hurts performance on
multi-Ractor execution, especially all Ractors calls methods
with rb_funcall*.

This patch introduced Global Call-Cache Cache Table (gccct) for
rb_funcall*. Call-Cache was introduced from Ruby 3.0 to manage
method cache entry atomically and gccct enables method-caching
without VM-wide locking. This table solves the performance issue
on multi-ractor execution.
[Bug #17497]

Ruby-level method invocation does not use gccct because it has
inline-method-cache and the table size is limited. Basically
rb_funcall* is not used frequently, so 1023 entries can be enough.
We will revisit the table size if it is not enough.

Revision 813fe4c256f89babebb8ab53821ae5eb6bb138c6 - 02/13/2021 02:51 AM - ko1 (Koichi Sasada)

opt_equality_by_mid for rb_equal_opt

This patch improves the performance of sequential and parallel
execution of rb_equal() (and rb_eql()).
[Bug #17497]

rb_equal_opt (and rb_eql_opt) does not have own cd and it waste
a time to initialize cd. This patch introduces opt_equality_by_mid()
to check equality without cd.

Furthermore, current master uses "static" cd on rb_equal_opt
(and rb_eql_opt) and it hurts CPU caches on multi-thread execution.
Now they are gone so there are no bottleneck on parallel execution.

Revision 813fe4c256f89babebb8ab53821ae5eb6bb138c6 - 02/13/2021 02:51 AM - ko1 (Koichi Sasada)

opt_equality_by_mid for rb_equal_opt

This patch improves the performance of sequential and parallel
execution of rb_equal() (and rb_eql()).
[Bug #17497]

rb_equal_opt (and rb_eql_opt) does not have own cd and it waste
a time to initialize cd. This patch introduces opt_equality_by_mid()
to check equality without cd.

Furthermore, current master uses "static" cd on rb_equal_opt
(and rb_eql_opt) and it hurts CPU caches on multi-thread execution.
Now they are gone so there are no bottleneck on parallel execution.

Revision 813fe4c2 - 02/13/2021 02:51 AM - ko1 (Koichi Sasada)

opt_equality_by_mid for rb_equal_opt

This patch improves the performance of sequential and parallel
execution of rb_equal() (and rb_eql()).
[Bug #17497]

rb_equal_opt (and rb_eql_opt) does not have own cd and it waste
a time to initialize cd. This patch introduces opt_equality_by_mid()
to check equality without cd.

Furthermore, current master uses "static" cd on rb_equal_opt
(and rb_eql_opt) and it hurts CPU caches on multi-thread execution.
Now they are gone so there are no bottleneck on parallel execution.

Revision de6072a22edbaab3793cf7f976cc9e0118d0df40 - 03/11/2021 11:24 AM - naruse (Yui NARUSE)

merge revision(s)
abdc634f64a440afcdc7f23c9757d27aab4db8a9,083c5f08ec4e95c9b75810d46f933928327a5ab3,1ecda213668644d656eb0d60654737482447dd92,
813fe4c256f89babebb8ab53821ae5eb6bb138c6: [Backport #17497]

```
    remove unused decl

    ---
     internal/vm.h | 6 ------
     vm_args.c     | 2 --
     2 files changed, 8 deletions(-)

    Check stack overflow in recursive glob_helper [Bug #17162]

    ---
     dir.c         |  2 ++
     internal/vm.h |  1 +
     vm_eval.c     | 10 ++++++++++
     3 files changed, 13 insertions(+)

    global call-cache cache table for rb_funcall*

    rb_funcall* (rb_funcall(), rb_funcallv(), ...) functions invokes
    Ruby's method with given receiver. Ruby 2.7 introduced inline method
    cache with static memory area. However, Ruby 3.0 reimplemented the
    method cache data structures and the inline cache was removed.

    Without inline cache, rb_funcall* searched methods everytime.
    Most of cases per-Class Method Cache (pCMC) will be helped but
    pCMC requires VM-wide locking and it hurts performance on
    multi-Ractor execution, especially all Ractors calls methods
    with rb_funcall*.

    This patch introduced Global Call-Cache Cache Table (gccct) for
    rb_funcall*. Call-Cache was introduced from Ruby 3.0 to manage
    method cache entry atomically and gccct enables method-caching
    without VM-wide locking. This table solves the performance issue
    on multi-ractor execution.
```

```
    [Bug #17497]

    Ruby-level method invocation does not use gccct because it has
    inline-method-cache and the table size is limited. Basically
    rb_funcall* is not used frequently, so 1023 entries can be enough.
    We will revisit the table size if it is not enough.
    ---
     debug_counter.h |   3 +
     vm.c            |  12 +++
     vm_callinfo.h   |  12 ---
     vm_core.h       |   5 +
     vm_eval.c       | 288 ++++++++++++++++++++++++++++++++++++++++++--------------------
     vm_insnhelper.c |  11 ++-
     vm_method.c     |  14 ++-
     7 files changed, 255 insertions(+), 90 deletions(-)

    opt_equality_by_mid for rb_equal_opt

    This patch improves the performance of sequential and parallel
    execution of rb_equal() (and rb_eql()).
    [Bug #17497]

    rb_equal_opt (and rb_eql_opt) does not have own cd and it waste
    a time to initialize cd. This patch introduces opt_equality_by_mid()
    to check equality without cd.

    Furthermore, current master uses "static" cd on rb_equal_opt
    (and rb_eql_opt) and it hurts CPU caches on multi-thread execution.
    Now they are gone so there are no bottleneck on parallel execution.
    ---
     vm_insnhelper.c | 99 +++++++++++++++++++++++++++++++++++++++++++---------------------------
     1 file changed, 63 insertions(+), 36 deletions(-)
```

Revision de6072a22edbaab3793cf7f976cc9e0118d0df40 - 03/11/2021 11:24 AM - naruse (Yui NARUSE)

merge revision(s) abdc634f64a440afcdc7f23c9757d27aab4db8a9,083c5f08ec4e95c9b75810d46f933928327a5ab3,1ecda213668644d656eb0d60654737482447dd92, 813fe4c256f89babebb8ab53821ae5eb6bb138c6: [Backport #17497]

```
    remove unused decl

    ---
     internal/vm.h | 6 ------
     vm_args.c     | 2 --
     2 files changed, 8 deletions(-)

    Check stack overflow in recursive glob_helper [Bug #17162]

    ---
     dir.c         |  2 ++
     internal/vm.h |  1 +
     vm_eval.c     | 10 ++++++++++
     3 files changed, 13 insertions(+)

    global call-cache cache table for rb_funcall*

    rb_funcall* (rb_funcall(), rb_funcallv(), ...) functions invokes
    Ruby's method with given receiver. Ruby 2.7 introduced inline method
    cache with static memory area. However, Ruby 3.0 reimplemented the
    method cache data structures and the inline cache was removed.

    Without inline cache, rb_funcall* searched methods everytime.
    Most of cases per-Class Method Cache (pCMC) will be helped but
    pCMC requires VM-wide locking and it hurts performance on
    multi-Ractor execution, especially all Ractors calls methods
    with rb_funcall*.

    This patch introduced Global Call-Cache Cache Table (gccct) for
    rb_funcall*. Call-Cache was introduced from Ruby 3.0 to manage
    method cache entry atomically and gccct enables method-caching
    without VM-wide locking. This table solves the performance issue
    on multi-ractor execution.
    [Bug #17497]

    Ruby-level method invocation does not use gccct because it has
```

```
    inline-method-cache and the table size is limited. Basically
    rb_funcall* is not used frequently, so 1023 entries can be enough.
    We will revisit the table size if it is not enough.
    ---
     debug_counter.h |   3 +
     vm.c            |  12 +++
     vm_callinfo.h   |  12 ---
     vm_core.h       |   5 +
     vm_eval.c       | 288 ++++++++++++++++++++++++++++++++++++++++++----------------
     vm_insnhelper.c |  11 ++-
     vm_method.c     |  14 ++-
     7 files changed, 255 insertions(+), 90 deletions(-)

    opt_equality_by_mid for rb_equal_opt

    This patch improves the performance of sequential and parallel
    execution of rb_equal() (and rb_eql()).
    [Bug #17497]

    rb_equal_opt (and rb_eql_opt) does not have own cd and it waste
    a time to initialize cd. This patch introduces opt_equality_by_mid()
    to check equality without cd.

    Furthermore, current master uses "static" cd on rb_equal_opt
    (and rb_eql_opt) and it hurts CPU caches on multi-thread execution.
    Now they are gone so there are no bottleneck on parallel execution.
    ---
     vm_insnhelper.c | 99 +++++++++++++++++++++++++++++++++++---------------------------
     1 file changed, 63 insertions(+), 36 deletions(-)
```

Revision de6072a2 - 03/11/2021 11:24 AM - naruse (Yui NARUSE)

merge revision(s)
abdc634f64a440afcdc7f23c9757d27aab4db8a9,083c5f08ec4e95c9b75810d46f933928327a5ab3,1ecda213668644d656eb0d60654737482447dd92,
813fe4c256f89babebb8ab53821ae5eb6bb138c6: [Backport #17497]

```
    remove unused decl

    ---
     internal/vm.h | 6 ------
     vm_args.c     | 2 --
     2 files changed, 8 deletions(-)

    Check stack overflow in recursive glob_helper [Bug #17162]

    ---
     dir.c         |  2 ++
     internal/vm.h |  1 +
     vm_eval.c     | 10 ++++++++++
     3 files changed, 13 insertions(+)

    global call-cache cache table for rb_funcall*

    rb_funcall* (rb_funcall(), rb_funcallv(), ...) functions invokes
    Ruby's method with given receiver. Ruby 2.7 introduced inline method
    cache with static memory area. However, Ruby 3.0 reimplemented the
    method cache data structures and the inline cache was removed.

    Without inline cache, rb_funcall* searched methods everytime.
    Most of cases per-Class Method Cache (pCMC) will be helped but
    pCMC requires VM-wide locking and it hurts performance on
    multi-Ractor execution, especially all Ractors calls methods
    with rb_funcall*.

    This patch introduced Global Call-Cache Cache Table (gccct) for
    rb_funcall*. Call-Cache was introduced from Ruby 3.0 to manage
    method cache entry atomically and gccct enables method-caching
    without VM-wide locking. This table solves the performance issue
    on multi-ractor execution.
    [Bug #17497]

    Ruby-level method invocation does not use gccct because it has
    inline-method-cache and the table size is limited. Basically
    rb_funcall* is not used frequently, so 1023 entries can be enough.
    We will revisit the table size if it is not enough.
```

```
    ---
     debug_counter.h |   3 +
     vm.c            |  12 +++
     vm_callinfo.h   |  12 ---
     vm_core.h       |   5 +
     vm_eval.c       | 288 ++++++++++++++++++++++++++++++++++++++++++--------------------
     vm_insnhelper.c |  11 ++-
     vm_method.c     |  14 ++-
     7 files changed, 255 insertions(+), 90 deletions(-)

    opt_equality_by_mid for rb_equal_opt

    This patch improves the performance of sequential and parallel
    execution of rb_equal() (and rb_eql()).
    [Bug #17497]

    rb_equal_opt (and rb_eql_opt) does not have own cd and it waste
    a time to initialize cd. This patch introduces opt_equality_by_mid()
    to check equality without cd.

    Furthermore, current master uses "static" cd on rb_equal_opt
    (and rb_eql_opt) and it hurts CPU caches on multi-thread execution.
    Now they are gone so there are no bottleneck on parallel execution.
    ---
     vm_insnhelper.c | 99 ++++++++++++++++++++++++++++++++++++-----------------------------
     1 file changed, 63 insertions(+), 36 deletions(-)
```

History

#1 - 01/03/2021 06:23 PM - MSP-Greg (Greg L)

Using various 2021-01-03 versions of master, got the following times, a few were averaged by eye:

```
────────── Windows 10 mingw ──────────
         fib    |  cpu     |  sleep
native   41 ms  |  940 ms  |  220 ms
ruby    345 ms  |  205 ms  |  243 ms

────────── WSL2/Ubuntu 20.04 ─────────
         fib    |  cpu     |  sleep
native   40 ms  |  530 ms  |  202 ms
ruby    330 ms  |  150 ms  |  203 ms

────────── Windows 10 mswin ──────────
         fib    |  cpu     |  sleep
native   88 ms  | 1080 ms  |  215 ms
ruby    690 ms  |  290 ms  |  235 ms
```

New, fast desktop. Always interested in comparing the three platforms/OS's. Ubuntu usually wins...

#2 - 01/03/2021 06:41 PM - marcandre (Marc-Andre Lafortune)

Thanks for running this on other platforms.

From the numbers, it looks like you are running on an 8-core machine, right? If so, the "fib" and "sleep" numbers are what should be expected, but "cpu" is 3-5x slower on Ractor when it should be 8x times faster...

#3 - 01/03/2021 08:01 PM - MSP-Greg (Greg L)

10 core i9. I've set up enough systems in my life (I used DOS); prefer new systems to last a while...

#4 - 01/05/2021 04:23 AM - ko1 (Koichi Sasada)

Thank you for the report. Let me investigate more.

(just curious) why is the name "CPU"?

#5 - 01/05/2021 04:44 AM - marcandre (Marc-Andre Lafortune)

ko1 (Koichi Sasada) wrote in #note-4:

> Thank you for the report. Let me investigate more.
>
> (just curious) why is the name "CPU"?

I wanted to compare CPU-bound processes (which should benefit from Ractor) to IO-bound processes (which should have similar benchmarks if my backport isn't too inefficient). I used sleep instead of IO because I'm lazy . It's only when I saw the issue that I tested other CPU-bound methods and added "fib".

#6 - 01/05/2021 07:47 AM - ko1 (Koichi Sasada)

flash report;

```
Warning[:experimental] = false if defined? Warning[]

def task_inject
  (1..10_000_000).inject(:+)
end

alias task task_inject
# p method(:task)

MODE = (ARGV.shift || :r_parallel).to_sym
TN = 4

case MODE
when :serial
  TN.times{ task }
when :r_serial
  exit(1) unless defined? Ractor

  TN.times{
    Ractor.new{
      task
    }.take
  }
when :r_parallel
  exit(1) unless defined? Ractor

  TN.times.map{
    Ractor.new{
      task
    }
  }.each{|r| r.take}
else
  raise
end

print "%4d" % GC.count
```

and

```
                              user     system      total        real
serial/26_mini               0  0.000000   0.000248   1.318308 (  1.318555)
serial/27_mini               0  0.000000   0.000627   1.209881 (  1.209730)
serial/master_mini           0  0.000000   0.000430   1.997904 (  1.997656)
serial/miniruby              0  0.000000   0.000254   1.723801 (  1.723786)
serial/26_ruby               0  0.000000   0.000481   1.256867 (  1.256746)
serial/27_ruby               0  0.000000   0.008709   1.098332 (  1.098257)
serial/master_ruby           0  0.000000   0.000312   1.915706 (  1.916034)
serial/ruby                  0  0.000000   0.000288   1.921821 (  1.921793)
r_serial/26_mini         N/A
r_serial/27_mini         N/A
r_serial/master_mini         1  0.000000   0.000388   2.460095 (  2.460922)
r_serial/miniruby            1  0.000000   0.000359   2.784072 (  2.784779)
r_serial/26_ruby         N/A
r_serial/27_ruby         N/A
r_serial/master_ruby         1  0.000000   0.000216   2.690338 (  2.690321)
r_serial/ruby                1  0.000000   0.000237   2.982560 (  2.983885)
r_parallel/26_mini       N/A
r_parallel/27_mini       N/A
r_parallel/master_mini       1  0.000000   0.000210  23.172113 (  6.316598)
r_parallel/miniruby          1  0.000000   0.000248  25.933848 (  7.054210)
r_parallel/26_ruby       N/A
r_parallel/27_ruby       N/A
r_parallel/master_ruby       1  0.000000   0.000214  25.243151 (  6.805798)
r_parallel/ruby              1  0.000000   0.000181  28.647737 (  7.991565)
```

- on serial execution, master is x2 slower than 2.6/2.7
- on serial execution with quiet ractor (multi-ractor-mode), master is -x2.5 times slower than 2.6/2.7

- on parallel execution with ractos, master is x7 slower than 2.6/2.7

#7 - 01/05/2021 08:08 AM - ko1 (Koichi Sasada)

master_ruby and ruby was same, so not needed.

with version information:

```
26_mini  ruby 2.6.7p148 (2020-06-14 revision 67884) [x86_64-linux]
27_mini  ruby 2.7.3p139 (2020-10-11 revision d1ba554551) [x86_64-linux]
miniruby ruby 3.1.0dev (2021-01-05T07:50:00Z master e91160f757) [x86_64-linux]
26_ruby  ruby 2.6.7p148 (2020-06-14 revision 67884) [x86_64-linux]
27_ruby  ruby 2.7.3p139 (2020-10-11 revision d1ba554551) [x86_64-linux]
ruby     ruby 3.1.0dev (2021-01-05T07:50:00Z master e91160f757) [x86_64-linux]
```

```
method: task_range_inject
                              user     system      total        real
serial/26_mini               0  0.000330   0.000000   1.334362 (  1.334173)
serial/27_mini               0  0.000401   0.000000   1.111259 (  1.111157)
serial/miniruby              0  0.000388   0.000000   1.803708 (  1.803610)
serial/26_ruby               0  0.000283   0.000000   1.274182 (  1.274051)
serial/27_ruby               0  0.000243   0.000000   1.102547 (  1.102493)
serial/ruby                  0  0.000340   0.000000   1.892771 (  1.892611)
r_serial/26_mini         N/A
r_serial/27_mini         N/A
r_serial/miniruby            1  0.000271   0.000000   2.484844 (  2.485867)
r_serial/26_ruby         N/A
r_serial/27_ruby         N/A
r_serial/ruby                1  0.000308   0.000000   2.736252 (  2.737059)
r_parallel/26_mini       N/A
r_parallel/27_mini       N/A
r_parallel/miniruby          1  0.000232   0.000000  20.958254 (  5.828039)
r_parallel/26_ruby       N/A
r_parallel/27_ruby       N/A
r_parallel/ruby              1  0.000396   0.000000  22.323984 (  6.165640)
```

#8 - 01/05/2021 09:07 AM - ko1 (Koichi Sasada)

with perf (with --call-graph dwarf) option, I may figure out the big difference:

master:

```
-   63.72%     6.23%  miniruby  miniruby  [.] inject_op_i
   - 57.49% inject_op_i
      - 55.13% rb_funcallv_public
         - rb_call (inlined)
            - 26.94% rb_call0
               + 17.69% rb_callable_method_entry_with_refinements
               + 3.30% stack_check (inlined)
                 2.75% rb_method_call_status (inlined)
               + 1.03% rb_class_of (inlined)
            - 25.91% rb_vm_call0
               - vm_call0_body (inlined)
                  + 13.27% vm_call0_cfunc (inlined)
                    0.50% vm_passed_block_handler (inlined)
                    0.50% rb_vm_check_ints (inlined)
              1.21% rb_current_execution_context (inlined)
              0.59% rb_vm_call_kw
        1.73% rb_sym2id
        0.63% rb_enum_values_pack
   + 4.38% _start
   + 0.53% 0x564cbc05b517
```

ruby 2.7:

```
-   46.93%     9.09%  miniruby  miniruby  [.] inject_op_i
   - 37.85% inject_op_i
      - 34.18% rb_funcallv_with_cc
         - 23.15% vm_call0_body
            + 17.07% vm_call0_cfunc (inlined)
              0.81% rb_vm_check_ints (inlined)
         + 4.22% vm_search_method (inlined)
        2.94% rb_sym2id
        0.73% rb_enum_values_pack
   + 8.13% _start
```

#9 - 01/05/2021 03:55 PM - marcandre (Marc-Andre Lafortune)

Just to be clear, there may be two different issues:

1. Ruby 2.x vs Ruby 3.0 performance regression. This can be important to figure out, but is *not* why I opened this issue.
2. Threads vs Ractor in Ruby 3.0. My tests are all in Ruby 3.0. This is what this issue is about.

#10 - 01/06/2021 03:41 PM - inversion (Yura Babak)

I also made 2 posts about strange performance testing results (with sources) and some conclusions.
In my case, 2 ractors work 3-times longer than doing the same payload in the main thread.

https://www.reddit.com/r/ruby/comments/kpmt73/ruby_30_ractors_performance_test_strange_results/
https://www.reddit.com/r/ruby/comments/krq5xe/when_ruby3_ractors_are_good_and_when_not_yet/

The test:

```
require 'digest/sha2'

N = 1_500_000

def workload
 N.times do |n|
  Digest::SHA2.base64digest(n.to_s)
 end
end

puts 'in the main thread'
t_start = Time.now
 workload
puts "Total: %.3f" % (Time.now - t_start)
puts


worker1 = Ractor.new do
 Ractor.receive
 print '['; workload; ']'
end
worker2 = Ractor.new do
 Ractor.receive
 print '['; workload; ']'
end

puts 'in 2 ractors'
t_start = Time.now
 worker1.send 'start'
 worker2.send 'start'
 print '='
 print worker1.take
 print worker2.take
puts
puts "Total: %.3f" % (Time.now - t_start)
```

#11 - 01/18/2021 06:11 PM - keithrbennett (Keith Bennett)

I too have seen strange results testing ractors. I used the code at https://github.com/keithrbennett/keithrbennett-ractor-test/blob/master/my_ractor.rb to do some arbitrary but predictable work. I have a 24-core Ryzen 9 CPU, and I compared using 1 ractor with using 24. With 24, htop reported that all the CPU's were at 100% most of the time, yet the elapsed time using 24 CPU's was only about a third less than when using 1 CPU. Also, the CPU's seemed to be working collectively about ten times harder with 24 CPU's. Here is the program output:

```
1 CPU:
Many HTOP readings are < 100% for all CPU's
 time ractor/my_ractor.rb ruby '*.rb'
Running the following command to find all filespecs to process: find -L ruby -type f -name '*.rb' -print
Processing 8218 files in 1 slices, whose sizes are:
[8218]
ractor/my_ractor.rb ruby '*.rb'  2513.90s user 6.75s system 99% cpu 42:03.01 total


24 CPU's:
% time ractor/my_ractor.rb ruby '*.rb' ; espeak finished
Running the following command to find all filespecs to process: find -L ruby -type f -name '*.rb' -print
Processing 8218 files in 24 slices, whose sizes are:
```

```
[343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343, 343,
 343, 329]
ractor/my_ractor.rb ruby '*.rb'  22986.42s user 14.98s system 1134% cpu 33:47.96 total
```

(In the command, ruby refers to the directory in which I've cloned the Github Ruby repo.)

Here is the current content of the test program:

```
#!/usr/bin/env ruby

require 'amazing_print'
require 'etc'
require 'set'
require 'shellwords'
require 'yaml'

raise "This script requires Ruby version 3 or later." unless RUBY_VERSION.split('.').first.to_i >= 3


# An instance of this parser class is created for each ractor.
class RactorParser

  attr_reader :dictionary_words

  def initialize(dictionary_words)
    @dictionary_words = dictionary_words
  end

  def parse(filespecs)
    filespecs.inject(Set.new) do |found_words, filespec|
      found_words | process_one_file(filespec)
    end
  end

  private def word?(string)
    dictionary_words.include?(string)
  end

  private def strip_punctuation(string)
    punctuation_regex = /[[:punct:]]/
    string.gsub(punctuation_regex, ' ')
  end

  private def file_lines(filespec)
    command = "strings #{Shellwords.escape(filespec)}"
    text = `#{command}`
    strip_punctuation(text).split("\n")
  end

  private def line_words(line)
    line.split.map(&:downcase).select { |text| word?(text) }
  end

  private def process_one_file(filespec)
    file_words = Set.new
    file_lines(filespec).each do |line|
      line_words(line).each { |word| file_words << word }
    end

    # puts "Found #{file_words.count} words in #{filespec}."
    file_words
  end
end


class Main

  BASEDIR =  ARGV[0] || '.'
  FILEMASK = ARGV[1]
  CPU_COUNT = Etc.nprocessors

  def call
    check_arg_count
    slices = get_filespec_slices
    ractors = create_and_populate_ractors(slices)
```

```
    all_words = collate_ractor_results(ractors)
    yaml = all_words.to_a.sort.to_yaml
    File.write('ractor-words.yaml', yaml)
    puts "Words are in ractor-words.yaml."
  end

  private def check_arg_count
    if ARGV.length > 2
      puts "Syntax is ractor [base_directory] [filemask], and filemask must be quoted so that the shell does n
ot expand it."
      exit -1
    end
  end

  private def collate_ractor_results(ractors)
    ractors.inject(Set.new) do |all_words, ractor|
      all_words | ractor.take
    end
  end

  private def get_filespec_slices
    all_filespecs = find_all_filespecs
    slice_size = (all_filespecs.size / CPU_COUNT) + 1
    # slice_size = all_filespecs.size # use this line instead of previous to test with 1 ractor
    slices = all_filespecs.each_slice(slice_size).to_a
    puts "Processing #{all_filespecs.size} files in #{slices.size} slices, whose sizes are:\n#{slices.map(&:si
ze).inspect}"
    slices
  end

  private def create_and_populate_ractors(slices)
    words = File.readlines('/usr/share/dict/words').map(&:chomp).map(&:downcase).sort

    slices.map do |slice|
      ractor = Ractor.new do
        filespecs = Ractor.receive
        dictionary_words = Ractor.receive
        RactorParser.new(dictionary_words).parse(filespecs)
      end
      ractor.send(slice)
      ractor.send(words)
      ractor
    end
  end

  private def find_all_filespecs
    filemask = FILEMASK ? %Q{-name '#{FILEMASK}'} : ''
    command = "find -L #{BASEDIR} -type f #{filemask} -print"
    puts "Running the following command to find all filespecs to process: #{command}"
    `#{command}`.split("\n")
  end
end

Main.new.call
```

#12 - 01/27/2021 11:49 PM - keithrbennett (Keith Bennett)

I've updated the software I used to measure this, and moved it to https://github.com/keithrbennett/keithrbennett-ractor-test.

#13 - 01/29/2021 07:22 AM - ko1 (Koichi Sasada)

- Status changed from Open to Closed

Applied in changeset git|1ecda213668644d656eb0d60654737482447dd92.

---

global call-cache cache table for rb_funcall*

rb_funcall* (rb_funcall(), rb_funcallv(), ...) functions invokes
Ruby's method with given receiver. Ruby 2.7 introduced inline method
cache with static memory area. However, Ruby 3.0 reimplemented the
method cache data structures and the inline cache was removed.

Without inline cache, rb_funcall* searched methods everytime.
Most of cases per-Class Method Cache (pCMC) will be helped but

pCMC requires VM-wide locking and it hurts performance on multi-Ractor execution, especially all Ractors calls methods with rb_funcall*.

This patch introduced Global Call-Cache Cache Table (gccct) for rb_funcall*. Call-Cache was introduced from Ruby 3.0 to manage method cache entry atomically and gccct enables method-caching without VM-wide locking. This table solves the performance issue on multi-ractor execution.
[Bug #17497]

Ruby-level method invocation does not use gccct because it has inline-method-cache and the table size is limited. Basically rb_funcall* is not used frequently, so 1023 entries can be enough. We will revisit the table size if it is not enough.

#14 - 01/29/2021 08:41 AM - ko1 (Koichi Sasada)

quoted from https://github.com/ruby/ruby/pull/4129#issuecomment-769613184

---

call the following methods as a task:

```
def task_range_inject
  (1..20_000_000).inject(:+)
end
```

with

- 4 times sequentially
- 4 times sequentially with a sleeping ractor
- 4 ractors in parallel

on

```
26_mini      ruby 2.6.7p150 (2020-12-09 revision 67888) [x86_64-linux]
27_mini      ruby 2.7.3p140 (2020-12-09 revision 9b884df6dd) [x86_64-linux]
master_mini ruby 3.1.0dev (2021-01-29T05:27:53Z master 9241211538) [x86_64-linux]
miniruby     ruby 3.1.0dev (2021-01-29T06:21:39Z gh-4129 f996e15ff6) [x86_64-linux]
26_ruby      ruby 2.6.7p150 (2020-12-09 revision 67888) [x86_64-linux]
27_ruby      ruby 2.7.3p139 (2020-10-11 revision d1ba554551) [x86_64-linux]
master_ruby ruby 3.1.0dev (2021-01-29T05:27:53Z master 9241211538) [x86_64-linux]
ruby         ruby 3.1.0dev (2021-01-29T06:21:39Z gh-4129 f996e15ff6) [x86_64-linux]
```

result:

```
                               user     system      total        real
serial/26_mini                  0  0.000145   0.000039   2.782685 (  2.783691)
serial/27_mini                  0  0.000133   0.000036   2.320257 (  2.320305)
serial/master_mini              0  0.000141   0.000039   3.756926 (  3.756963)
serial/miniruby                 0  0.000136   0.000037   2.598088 (  2.598126)
serial/26_ruby                  0  0.000143   0.000038   2.695175 (  2.704443)
serial/27_ruby                  0  0.000139   0.000038   2.391679 (  2.401067)
serial/master_ruby              0  0.000139   0.000038   4.391577 (  4.391626)
serial/ruby                     0  0.000128   0.000035   3.109923 (  3.109991)
r_serial/26_mini              N/A
r_serial/27_mini              N/A
r_serial/master_mini            1  0.000146   0.000040   5.133049 (  5.133056)
r_serial/miniruby               1  0.000133   0.000037   2.597336 (  2.597300)
r_serial/26_ruby              N/A
r_serial/27_ruby              N/A
r_serial/master_ruby            1  0.000147   0.000040   5.910876 (  5.910907)
r_serial/ruby                   1  0.000135   0.000037   2.875752 (  2.875727)
r_parallel/26_mini            N/A
r_parallel/27_mini            N/A
r_parallel/master_mini          1  0.000100   0.000028  39.297123 ( 10.160359)
r_parallel/miniruby             1  0.000110   0.000030   2.703400 (  0.695634)
r_parallel/26_ruby            N/A
r_parallel/27_ruby            N/A
r_parallel/master_ruby          1  0.000122   0.000034  38.941810 ( 10.072950)
r_parallel/ruby                 1  0.000131   0.000036   2.980137 (  0.757672)
```

#15 - 01/29/2021 08:46 AM - ko1 (Koichi Sasada)

inversion (Yura Babak) wrote in #note-10:

> I also made 2 posts about strange performance testing results (with sources) and some conclusions.
> In my case, 2 ractors work 3-times longer than doing the same payload in the main thread.

With digest benchmark:

```
                              user     system      total        real
serial/26_ruby                0  0.000239   0.000079   2.497317 (  2.503279)
serial/27_ruby                0  0.000972   0.000324   2.306552 (  2.310275)
serial/master_ruby            0  0.000293   0.000098   3.776824 (  3.776623)
serial/ruby                   0  0.000190   0.000063   2.668395 (  2.668287)
r_serial/26_ruby            N/A
r_serial/27_ruby            N/A
r_serial/master_ruby          1  0.000407   0.000000   5.579597 (  5.579969)
r_serial/ruby                 1  0.000476   0.000000   2.682626 (  2.683627)
r_parallel/26_ruby          N/A
r_parallel/27_ruby          N/A
r_parallel/master_ruby        1  0.000242   0.000000  48.298173 ( 12.959255)
r_parallel/ruby               1  0.000242   0.000000   3.782164 (  0.984832)
```

seems solved.

#16 - 01/29/2021 08:47 AM - ko1 (Koichi Sasada)

keithrbennett (Keith Bennett) wrote in #note-11:

> I too have seen strange results testing ractors. I used the code at https://github.com/keithrbennett/keithrbennett-ractor-test/blob/master/my_ractor.rb to do some arbitrary but predictable work. I have a 24-core Ryzen 9 CPU, and I compared using 1 ractor with using 24. With 24, htop reported that all the CPU's were at 100% most of the time, yet the elapsed time using 24 CPU's was only about a third less than when using 1 CPU. Also, the CPU's seemed to be working collectively about ten times harder with 24 CPU's. Here is the program output:

could you check it again?

#17 - 01/31/2021 08:57 PM - keithrbennett (Keith Bennett)

@ko1 (Koichi Sasada) - My apologies for not responding sooner. I guess I have not configured this forum correctly to receive notifications, I'll look into that.

I've tested my benchmark against Ruby head, and performance with multiple cores seem to have degraded. Perhaps I have made an error in my approaches, I don't know. I will paste my results below. My OS is "Ubuntu 20.04.2 LTS" (Kubuntu).

In case it's useful, I've made my script easier to use; it automatically tests and compares 1 ractor with (CPU_count) ractors. You can find it at https://github.com/keithrbennett/keithrbennett-ractor-test/blob/master/ractor-file-strings-test.rb. Information about configuring it, what it does, etc., is included in comments at the top of the script.

```
Small Data Set:

Ruby 3.0.0:

                          1 CPU         24 CPU's           Factor
-----------------------------------------------------------------
User                    6.75500         52.34200          7.74863
System                  0.00000          0.02800          0.00415
Total                   6.83900         52.40700          7.66296
Real                    6.83300          4.10500          0.60076


2021-01-31 Ruby head (ruby 3.1.0dev (2021-01-31T09:48:28Z master 22b8ddfd10) [x86_64-linux]):

                          1 CPU         24 CPU's           Factor
-----------------------------------------------------------------
User                    6.18000         56.27400          9.10583
System                  0.00400          0.02800          0.00453
Total                   6.26800         56.34200          8.98883
Real                    6.26100          4.26200          0.68072


=================================================================

Larger Data Set:

Ruby 3.0.0:
```

```
                         1 CPU         24 CPU's           Factor
-------------------------------------------------------------------
User                  51.01000        499.67200          9.79557
System                 0.04000          0.25900          0.00508
Total                 51.32300        500.12900          9.74473
Real                  51.31200         45.56600          0.88802


2021-01-31 Ruby head (ruby 3.1.0dev (2021-01-31T09:48:28Z master 22b8ddfd10) [x86_64-linux]):

                         1 CPU         24 CPU's           Factor
-------------------------------------------------------------------
User                  47.08900        486.34400         10.32819
System                 0.03200          0.20300          0.00431
Total                 47.39500        486.74800         10.27003
Real                  47.38400         43.95100          0.92755
```

#18 - 02/12/2021 08:02 AM - ko1 (Koichi Sasada)

- Status changed from Closed to Assigned

keithrbennett (Keith Bennett) wrote in #note-17:

> I've tested my benchmark against Ruby head, and performance with multiple cores seem to have degraded. Perhaps I have made an error in my approaches, I don't know. I will paste my results below. My OS is "Ubuntu 20.04.2 LTS" (Kubuntu).

I confirmed with the following script

```
WORDS = Ractor.make_shareable File.readlines('/usr/share/dict/words').map(&:chomp).map(&:downcase).sort

def try
  File.readlines(__dir__ + '/compar.c').each{|line|
    line.split.map(&:downcase).select { |text|
      WORDS.include? text
    }
  }
end

Warning[:experimental] = false

require 'benchmark'

Benchmark.bm{|x|
  x.report{
    4.times{try}
  }
  x.report{
    4.times.map{
      Ractor.new{ try }
    }.each(&:take)
  }
}

__END__

       user     system      total        real
   4.501388   0.001541   4.502929 (  4.502980)
  16.763446   0.000018  16.763464 (  4.335964)
```

It compare with sequential 4 times try method and 4 times try methods on ractors in parallel.
To compare with real, 4.5 vs 4.3 sec. It is not slow, but not first with 4 cores.

The reason seems WORDS.include? text. I'll investigate more.

#19 - 02/13/2021 02:52 AM - ko1 (Koichi Sasada)

- Status changed from Assigned to Closed

Applied in changeset git|813fe4c256f89babebb8ab53821ae5eb6bb138c6.

---

opt_equality_by_mid for rb_equal_opt

This patch improves the performance of sequential and parallel
execution of rb_equal() (and rb_eql()).
[Bug #17497]

rb_equal_opt (and rb_eql_opt) does not have own cd and it waste
a time to initialize cd. This patch introduces opt_equality_by_mid()
to check equality without cd.

Furthermore, current master uses "static" cd on rb_equal_opt
(and rb_eql_opt) and it hurts CPU caches on multi-thread execution.
Now they are gone so there are no bottleneck on parallel execution.

#20 - 02/13/2021 02:52 AM - ko1 (Koichi Sasada)

maybe git|813fe4c256f89babebb8ab53821ae5eb6bb138c6 solved the issue.
could you check it?

#21 - 02/13/2021 02:53 AM - ko1 (Koichi Sasada)

- Backport changed from 2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN to 2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN, 3.0: REQUIRED

#22 - 02/15/2021 09:40 PM - keithrbennett (Keith Bennett)

Koichi -

Huge improvement! Thank you!

I installed Ruby head and now have the following output from ruby -v:

```
ruby 3.1.0dev (2021-02-15T09:29:35Z master 37b90bcdc1) [x86_64-linux]
```

I made minor modifications to your script (see https://gist.github.com/keithrbennett/18f10124354d62eb8ba5feafaa9b39dc) and then ran it in the Ruby project root directory and got the following results:

On my Linux (Kubuntu 20.04.2) desktop:

```
Measuring first sequentially on main ractor and then with 24 ractors:

       user     system      total        real
  14.969907   0.003891  14.973798 ( 14.977699)
  29.087580   0.051934  29.139514 (  1.243316)

  0.515  User time difference factor
 12.047  Real time difference factor
```

And then on my 2015 Mac:

```
Measuring first sequentially on main ractor and then with 4 ractors:

       user     system      total        real
  10.477194   0.047028  10.524222 ( 10.605862)
  18.226199   0.068098  18.294297 (  5.101498)

  0.575  User time difference factor
  2.079  Real time difference factor
```

It's interesting that the real time difference factor on both machines is so close to ((the number of CPU's and ractors) / 2.0).

The original script I used to test (at https://github.com/keithrbennett/keithrbennett-ractor-test/blob/master/ractor-file-strings-test.rb) was not very good at distributing work among the ractors equally, and this made the real time observations less reliable, since the real time was really the real time of the longest running ractor. Your script is much better in that way. It would be interesting to test more parts of the standard library though, such as the Set instantiations and merges I had used; if I have time I'll look into that.

P.S. Sorry it took so long to respond; given that notifications don't seem to work, I need to develop a habit of manually checking here every day.

#23 - 03/11/2021 11:25 AM - naruse (Yui NARUSE)

- Backport changed from 2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN, 3.0: REQUIRED to 2.5: UNKNOWN, 2.6: UNKNOWN, 2.7: UNKNOWN, 3.0: DONE

ruby_3_0 de6072a22edbaab3793cf7f976cc9e0118d0df40 merged revision(s)
abdc634f64a440afcdc7f23c9757d27aab4db8a9,083c5f08ec4e95c9b75810d46f933928327a5ab3,1ecda213668644d656eb0d60654737482447dd92,
813fe4c256f89babebb8ab53821ae5eb6bb138c6.