

# Mesurer le temps d'exécution d'un programme

Stéphane Bortzmeyer

<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 18 septembre 2006. Dernière mise à jour le 5 novembre 2006

<https://www.bortzmeyer.org/mesurer-temps-execution.html>

---

On lit très souvent des discussions sur l'optimisation d'un programme informatique, discussions pleines d'argument du genre "X est plus rapide que Y" où X et Y sont des algorithmes, des SGBD, des langages de programmation... Très souvent, les asséneurs de ces arguments n'ont aucun fait sur lequel appuyer leurs affirmations, car ils n'ont jamais **mesuré**.

"*Premature optimisation is the root of all evil*" a affirmé le célèbre C.A.R. Hoare. Il voulait insister sur l'importance qu'il y a à faire un programme correct, avant de faire un programme rapide. Mais on peut aussi appliquer son dicton à l'importance qu'il y a à **tester** les techniques pour voir laquelle est réellement plus rapide (ou moins gourmande en espace ou autres ressources).

Sur un système Unix, la manière la plus évidente de tester est d'utiliser la commande `time`, ici pour tester un programme Perl :

```
% time perl parse-with-regexp.pl `cat ./test-file`  
...  
... 0.07s user 0.03s system 24% cpu 0.421 total
```

On voit que le programme a mis 0,421 secondes à s'exécuter, mais que seul un quart de ce temps à été consacré à du travail du CPU. Soit le programme faisait beaucoup d'entrées/sorties, soit d'autres programmes s'exécutaient au même moment sur la machine.

Un autre bon exemple d'utilisation de `time` est présenté par Denis Barbier qui montre que, certes, `grep` fonctionne partiellement en UTF-8 (un problème que j'ai pour passer à UTF-8 <<https://www.bortzmeyer.org/pas-encore-utf8.html>>) mais qu'il est beaucoup plus lent avec cet encodage <<http://lists.debian.org/debian-devel-french/2005/09/msg00029.html>>.

La principale limite de `time` est qu'elle prend en compte l'initialisation du programme (pour Perl, c'est le chargement du compilateur et de la machine virtuelle, puis la compilation du code, avant même que son exécution ne commence). Si je testais une base de données, `time` prendrait en compte le temps de connexion en plus du temps d'exécution de la requête. Si je testais une opération réseau, par exemple avec `wget`, `time` prendrait en compte le temps de résolution DNS, ce dont je n'ai pas forcément envie.

Il est donc généralement préférable de ne déclencher le chronomètre qu'après que l'initialisation aie été faite. C'est ce que fait `echoping` <<http://echoping.sourceforge.net/>>, qui ne lance ledit chronomètre (en appelant `gettimeofday`) qu'après que la résolution de noms en adresse aie été faite (par `getaddrinfo`) et juste avant de tenter une connexion (par `connect`).

Bien sûr, la mesure reste un monde compliqué et plein d'embûches. Par exemple, il est recommandé d'exécuter la commande plusieurs fois, pour tenir compte du résultat des caches. C'est pour cette raison que la plupart des "benchmarks" ne signifient pas grand'chose.

Voici un exemple de mesure de performance d'un serveur réseau. Ici, on veut comparer deux serveurs HTTP, Apache et `thttpd`. Pour distinguer le temps de résolution de noms (qui dépend du DNS) du temps de connexion TCP (qui, si le serveur est une machine Unix, ne dépend que du noyau, pas du serveur), du temps total, on peut utiliser `curl`, qui dispose d'options comme `--write-out`, qui permettent de n'afficher que le temps considéré comme "utile" :

```
% curl --silent --output /dev/null --write-out 'DNS: %{time_namelookup} s\nConnect:%{time_connect} s\nTotal.
DNS: 0.001 s
Connect:0.001 s
Total: 0.126 s
```

Et le même test avec `echoping` <<http://echoping.sourceforge.net/>> :

```
% echoping -v -h / www.demaziere.fr
...
TCP Latency: 0.046543 seconds
Sent (92 bytes)...
Application Latency: 0.065827 seconds
610 bytes read from server.
Elapsed time: 0.118573 seconds
```

On voit ici que l'essentiel du temps est bien dû au serveur. `echoping` <<http://echoping.sourceforge.net/>>, lui, dispose de davantage d'options statistiques, par exemple `-n` qui permet de répéter un test pour s'assurer de sa reproductibilité et de calculer moyenne et surtout médiane, une valeur en général bien plus utile sur Internet. Essayons pour comparer nos serveurs HTTP :

```
% echoping -n 4 -h / preston.sources.org
...
Median time: 0.007902 seconds (32397 bytes per sec.)
...
```

Ce premier test était fait sur un serveur Apache. Avec un serveur `thttpd`, sur la même machine, on obtient :

```
...
Median time: 0.011717 seconds (21849 bytes per sec.)
...
```

Bref, `thttpd` n'est pas plus rapide, dans ces conditions. Bien sûr, ce test n'épuise pas tout le problème. `thttpd` résiste peut-être mieux à la charge, par exemple. Mais cela montre l'importance de mesurer les performances, pas de simplement croire les auteurs du logiciel, qui présentent toujours leur logiciel comme "plus meilleur".

Il existe de nombreux autres outils de mesure des performances des serveurs réseau. Par exemple, pour mesurer les performances <http://www.generic-nic.net/sheets/practical/nameserver-en> d'un serveur DNS, on utilise souvent le programme `queryperf` <https://www.bortzmeyer.org/performances-serveur-dns.html>, distribué avec BIND. Et en local ?

Tous les langages de programmation offrent aujourd'hui un moyen de mesurer le temps d'exécution. Par exemple, pour Python, on a le module `timeit`.

Voici un exemple de test de performance, entre plusieurs modules d'accès à une base de données PostgreSQL. Si on avait voulu comparer plusieurs SGBD, la méthode aurait été la même. On commence par créer une base de données, à l'aide des instructions SQL suivants :

```
CREATE TABLE Registrars (id SERIAL UNIQUE,
                          created TIMESTAMP NOT NULL DEFAULT now() , name TEXT UNIQUE NOT
                          NULL);
INSERT INTO Registrars (name) VALUES ('Indomco');
INSERT INTO Registrars (name) VALUES ('Verisign');
INSERT INTO Registrars (name) VALUES ('Gandi');

CREATE TABLE Domains (id SERIAL UNIQUE, name TEXT UNIQUE NOT NULL,
                      created TIMESTAMP NOT NULL DEFAULT now(),
                      registrar INTEGER REFERENCES Registrars(id) NOT NULL);
```

puis le programme Python suivant va se connecter à la base (en utilisant `tel` ou `tel` module, selon la valeur de la variable d'environnement `PYTHON_DB`) :

```
import timeit
import os
import sys

repetitions = 100
inserts = 1000
selects = inserts
deletes = inserts

domain_prefix = "example"

try:
    if os.environ['PYTHON_DB'] == 'psycopg':
        import psycopg
        db_module = psycopg
    elif os.environ['PYTHON_DB'] == 'pypg':
        import pypg
        db_module = pypg
    elif os.environ['PYTHON_DB'] == 'popy':
        import PoPy
        db_module = PoPy
```

---

<https://www.bortzmeyer.org/mesurer-temps-execution.html>

```

""" TODO: PoPy has a problem with the paramstyle pyformat. http://www.python.org/peps/pep-0249.html
paramstyle

String constant stating the type of parameter marker
formatting expected by the interface. Possible values are
[2]:

    'qmark'          Question mark style,
                    e.g. '...WHERE name=?'
    'numeric'       Numeric, positional style,
                    e.g. '...WHERE name=:1'
    'named'         Named style,
                    e.g. '...WHERE name=:name'
    'format'        ANSI C printf format codes,
                    e.g. '...WHERE name=%s'
    'pyformat'      Python extended format codes,
                    e.g. '...WHERE name=%(name)s'

but PoPy crashes:
    src/samples/db-comparison % python test.py
Traceback (most recent call last):
  File "test.py", line 46, in ?
    run(connection, cursor)
  File "test.py", line 25, in run
    {'name': "%s-%i" % (domain_prefix, i), 'registrar': 'Gandi'})
PoPy.ProgrammingError: ERROR: column "gandi" does not exist

"""
else:
    raise Exception("Unsupported DB module %s" %
                    os.environ['PYTHON_DB'])
except KeyError:
    sys.stderr.write("You must define the environment variable PYTHON_DB\n")
    sys.exit(1)

insertion_time = 0
selection_time = 0
deletion_time = 0

def run(operation, iterations, connection, cursor):
    if operation == "INSERT":
        cursor.execute("SELECT id FROM Registrars WHERE name='Gandi'")
        id = cursor.fetchone()[0]
        for i in range(1, iterations):
            cursor.execute("""
                INSERT INTO Domains (name, registrar) VALUES
                ('%s', %i)""" %
                ("%s-%i" % (domain_prefix, i), id))
            connection.commit()
    elif operation == "SELECT":
        for i in range(1, iterations):
            cursor.execute("""
                SELECT Domains.id, Domains.name, Domains.created,
                Registrars.name AS registrar
                FROM Domains, Registrars WHERE
                Domains.name = '%s' AND Domains.registrar = Registrars.id""" %
                ("%s-%i" % (domain_prefix, i)))
            connection.commit()
    elif operation == "DELETE":
        for i in range(1, iterations):
            cursor.execute("""
                DELETE FROM Domains WHERE name = '%s'""" %
                ("%s-%i" % (domain_prefix, i)))
            connection.commit()
    else:
        raise Exception("Unknown operation: %s" % operation)

def run_all(connection, cursor):
    global insertion_time, selection_time, deletion_time

```

```

    t = timeit.Timer(stmt="""
run("INSERT", inserts, connection, cursor)
""", setup="""
from __main__ import run, inserts, connection, cursor
""")
    insertion_time = insertion_time + t.timeit(number=1)
    t = timeit.Timer(stmt="""
run("SELECT", selects, connection, cursor)
""", setup="""
from __main__ import run, selects, connection, cursor
""")
    selection_time = selection_time + t.timeit(number=1)
    t = timeit.Timer(stmt="""
run("DELETE", deletes, connection, cursor)
""", setup="""
from __main__ import run, deletes, connection, cursor
""")
    deletion_time = deletion_time + t.timeit(number=1)

if os.environ['PYTHON_DB'] != 'pypg':
    # pypg does not support connection strings :-(
    connection = db_module.connect("dbname=db-comp-registry")
else: # Broken pypg
    connection = db_module.connect(host='localhost', database='db-comp-registry')
cursor = connection.cursor()
# Clean the DB
cursor.execute("DELETE FROM Domains")
connection.commit()

if __name__ == '__main__':
    print "Result with %s" % db_module
    sys.stderr.write("%i repetitions:\n" % repetitions)
    for i in range(1, repetitions+1):
        sys.stderr.write("%i... " % i)
        sys.stderr.flush()
    run_all(connection, cursor)
    sys.stderr.write("\n")
    print "INSERT: %.1f msec/pass" % (insertion_time*1000/inserts/repetitions)
    print "SELECT (with join): %.1f msec/pass" % (selection_time*1000/selects/repetitions)
    print "DELETE: %.1f msec/pass" % (deletion_time*1000/deletes/repetitions)

```

Son exécution donne :

```

INSERT: 0.5 msec/pass
SELECT (with join): 4.5 msec/pass
DELETE: 3.9 msec/pass

```

ce qui indique que le programme met une demi-milliseconde par insertion, soit 2000 insertions par seconde. On peut refaire tourner le programme avec d'autres modules d'accès à la base pour voir s'il ya des différences de performance.

Empruntées à Andrew Bromage, et pour compléter la citation de Hoare ci-dessus, une belle liste de lois sur l'optimisation :

- *"Rule 1. Don't do it."*
- *"Rule 2. (For experts only) Don't do it yet."*
- *"Rule 3. (For real experts only) When you've found a performance issue, profile first, find where the bottle-necks are, and then optimise. For if you guess, you'll guess wrong."*
- *"Rule 4. (For serious guru experts only) Sometimes, performance matters. When it matters, it really matters. If you find yourself in this situation, then the performance that you need must be designed in; it can't be added later. So go ahead and spend ten minutes optimising your code on the back of an envelope before writing anything."*

<https://www.bortzmeyer.org/mesurer-temps-execution.html>