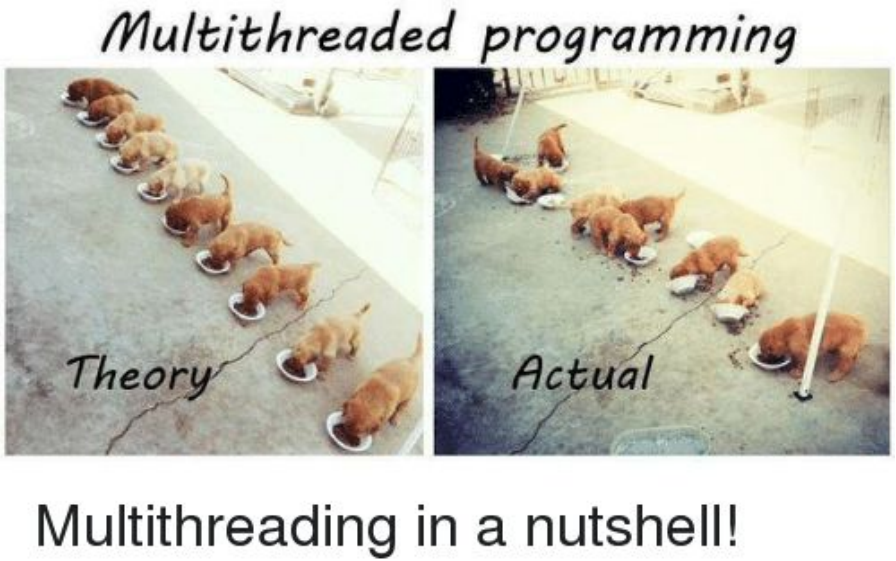


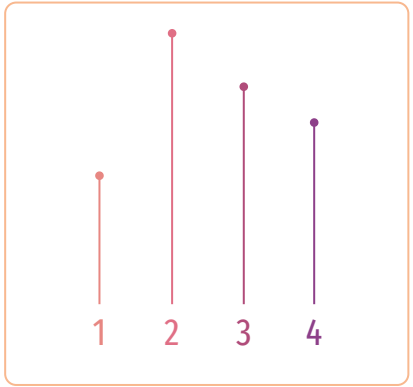
Threading Frenzy

Approche de l'évaluation du multithreading pour une variété de langages

BABONNEAU Matisse
DUVIVIER Alexandre
PAROL-GUARINO Volodia

Nos objectifs

- Evaluer le multithreading que l'on connaît à l'INSA
- Donner un avis sur les meilleurs langages
- Argumenter l'usage de chaque langage
- Illustrer les avantages et inconvénients



Les phases

Evaluation de multithreading

Evaluation suivant un même algorithme (approche de Pi)

Evaluation avec les outils des langages (l’exemple des tris)

Les langages que nous testons

Evaluation suivant un même algorithme (approche de Pi)

Pi - Protocole de test

- Même algorithme dans tous les langages
- Variation du nombre de pas par thread
- Variation du nombre threads
- Multiples exécutions pour lisser les imperfections

Environnement :

Intel CORE i7 4810MQ (2.8GHz) 4 cores
8 threads

Evaluation avec les outils des langages (l'exemple des tris)

Tri - Protocole de test

Environnement :

Intel CORE i7 4810MQ (2.8GHz) 4 cores
8 threads

- Variation de l'implémentation dans chaque langage
- Variation du nombre d'éléments dans le tableau (N éléments entre 0 et N)
- Variation du nombre threads
- Multiples exécutions pour lisser les imperfections

Phase 1: Évaluation suivant un même algorithme (approche de Pi)

Pi - Data format

- Chaque programme utilise le même formatage.
- Automatise le traitements des données.

Format:

```
3.1415946535888706      // Pi
0.10193371772766113     // Duration
```

Données en entrée :

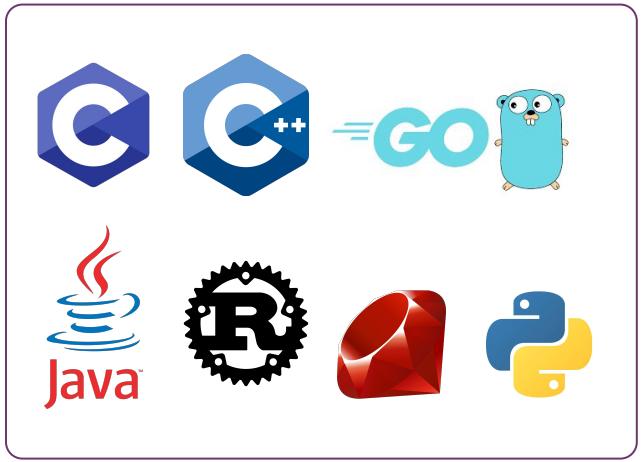
- N : Nombre d'itérations : 10, 100, 1000, 10000, 25000, 50000, 100000, 250000, 500000, 1000000

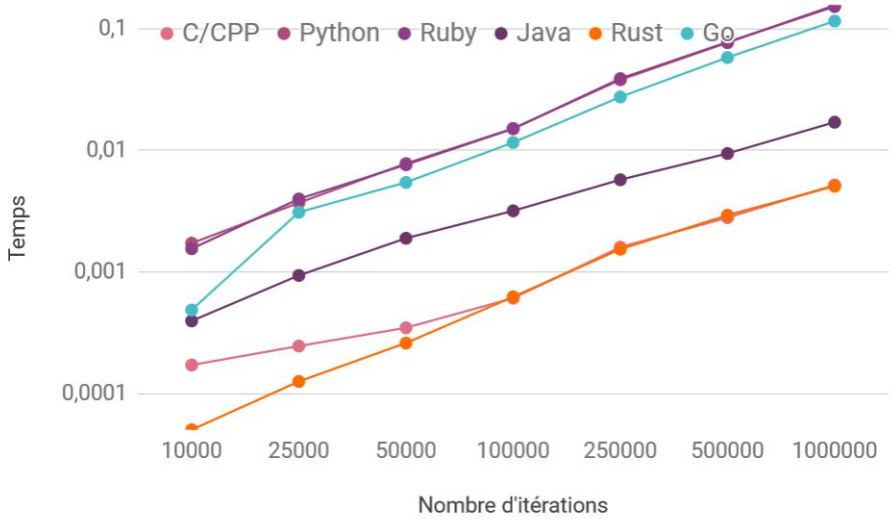
- NT : Nombre de threads (quand applicable) : 1 2 4 8

Pi - approche séquentielle

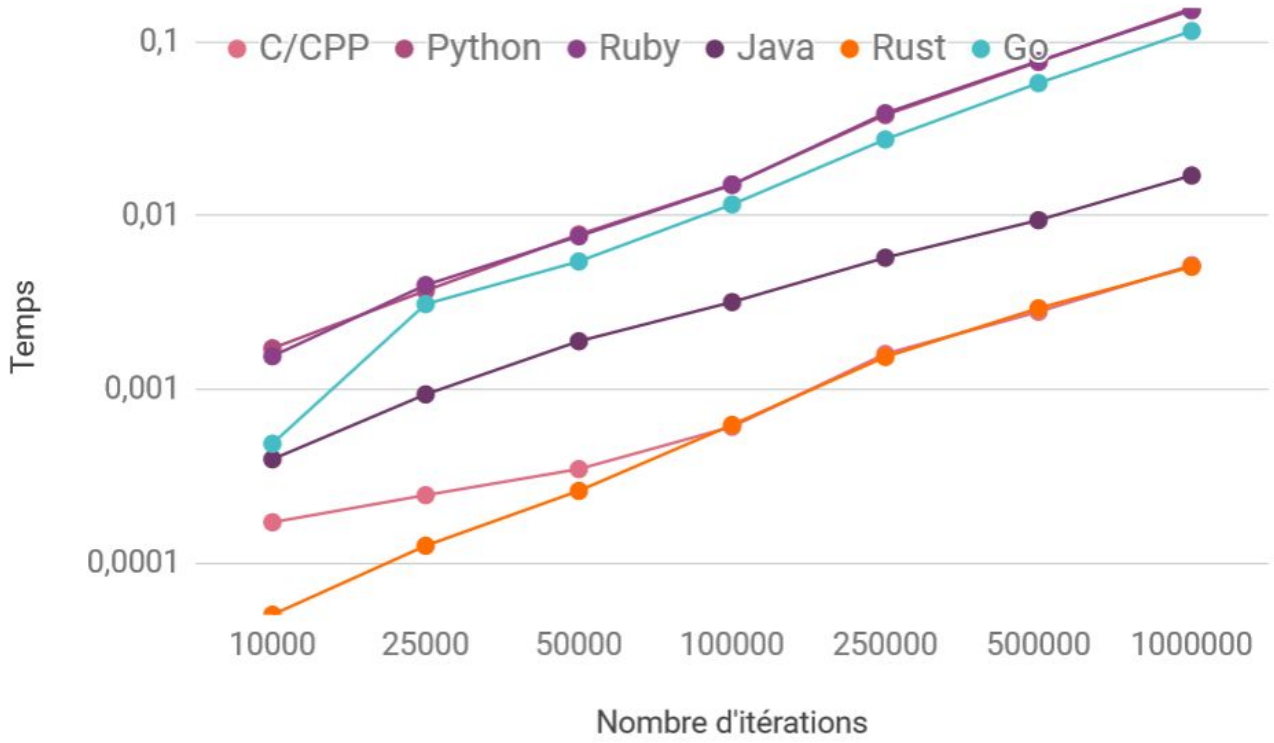
```
nb_steps = 1000000
sum = 0.0
steps = 1 / nb_steps

for (ii = 1 ;  ii <= nb_steps ; ++ii){
    x = (i-0.5)*steps
    sum += 4.0 / (1.0 + x * x)
}

p = steps * sum
```

- Un premier test
- Observer sans parallélisme





Pi - approche séquentielle



Pi - approche parallèle

```
compute(start, end, steps){
    sum = 0.0
    x = 0.0

    for (ii = start ;  ii <= end ; ++ii){
        x = (ii-0.5)*steps
        sum += 4.0 / (1.0 + x * x)
    }

    return sum
}

main() {
    nb_steps = 1000000
    threads = 4
    sum = 0.0
    steps = 1 / nb_steps

    div = (int) nb_steps / threads
    last_div = div + nb_steps - div * threads // add what's missing to the last thread

    // start threads-1 threads
    for (ii = 1; ii < threads - 1 ; ++ii){
        start_thread((id_thread) -> {
            sum += compute(id_thread * div, (id_thread+1)*div, steps)
        })
    }

    sum += compute((threads-1)*div, (threads-1)*div + last_div, steps)

    wait_for_all_threads()

    p = steps * sum
}
```

- L'implémentation diffère selon le langage et les bibliothèques.
- Néanmoins il s'agit toujours du même algo.
- Variation du nombre de pas par thread
- Variation du nombre threads
- Multiples exécutions pour lisser les imperfections

Pi - approche parallèle selon un langage

Le Go est né pour la monté en charge. Il embarque de manière native le parallélisme via les **goroutines**.

Simple, efficace:

go

```go
package main

import (
    "fmt"
    "time"
)

func run(name string) {
    for i := 0; i < 3; i++ {
        time.Sleep(1 * time.Second)
        fmt.Println(name, " : ", i)
    }
}

func main() {
    debut := time.Now()
    go run("Hatim")
    go run("Robert")
    run("Alex")
    fin := time.Now()
    fmt.Println(fin.Sub(debut))
}
```

```
Robert  :  0
Hatim  :  0
Alex  :  0
Hatim  :  1
Robert  :  1
Alex  :  1
Robert  :  2
Hatim  :  2
Alex  :  2
3.0022266s
```

De nombreux outils comme les **channels** :

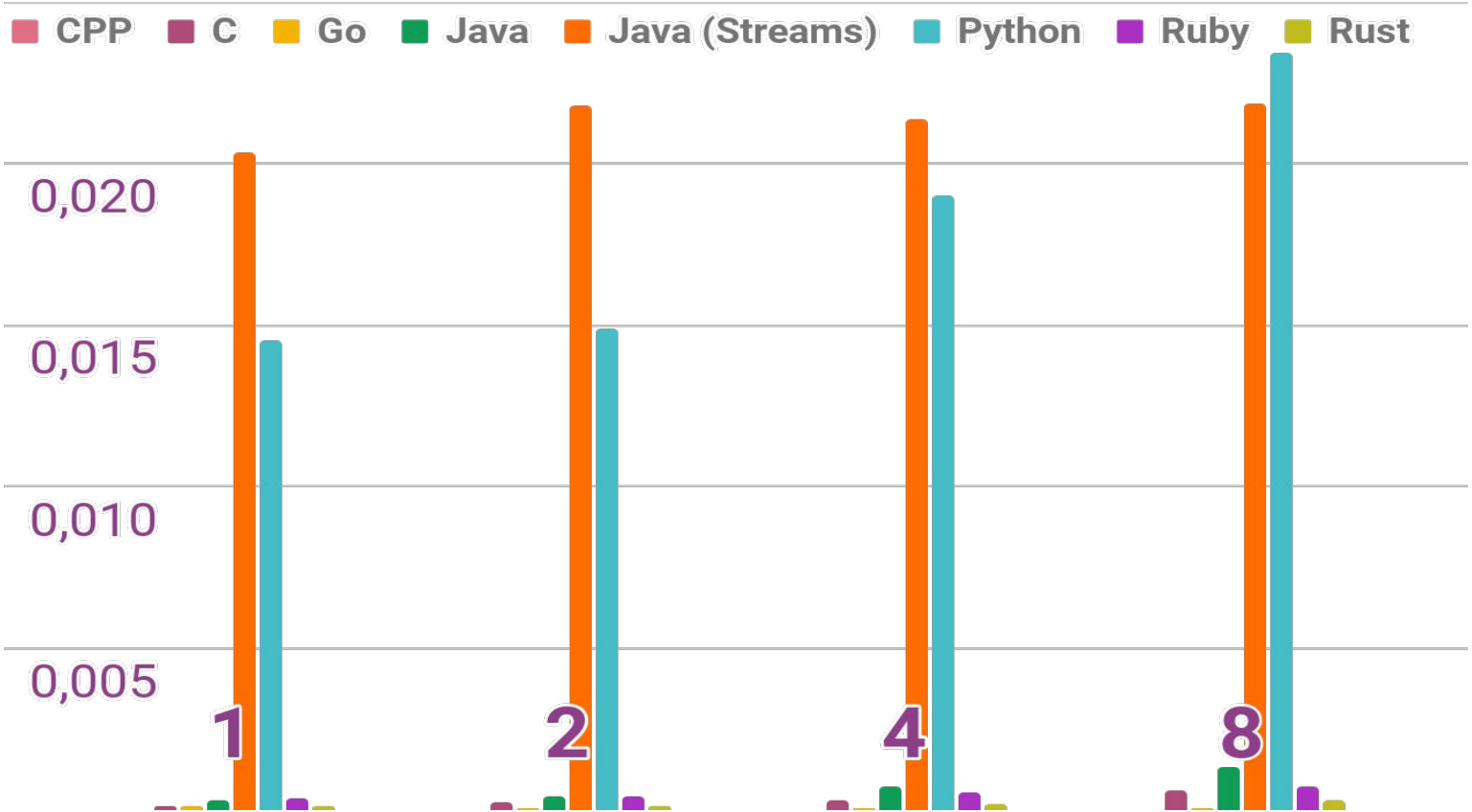
- Echange de données
- Synchronisation du parallélisme

```go
package main

import "fmt"

func ping(pings chan<- string, msg string) {
    pings <- msg
}

func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

Pi - approche parallèle

N=1000

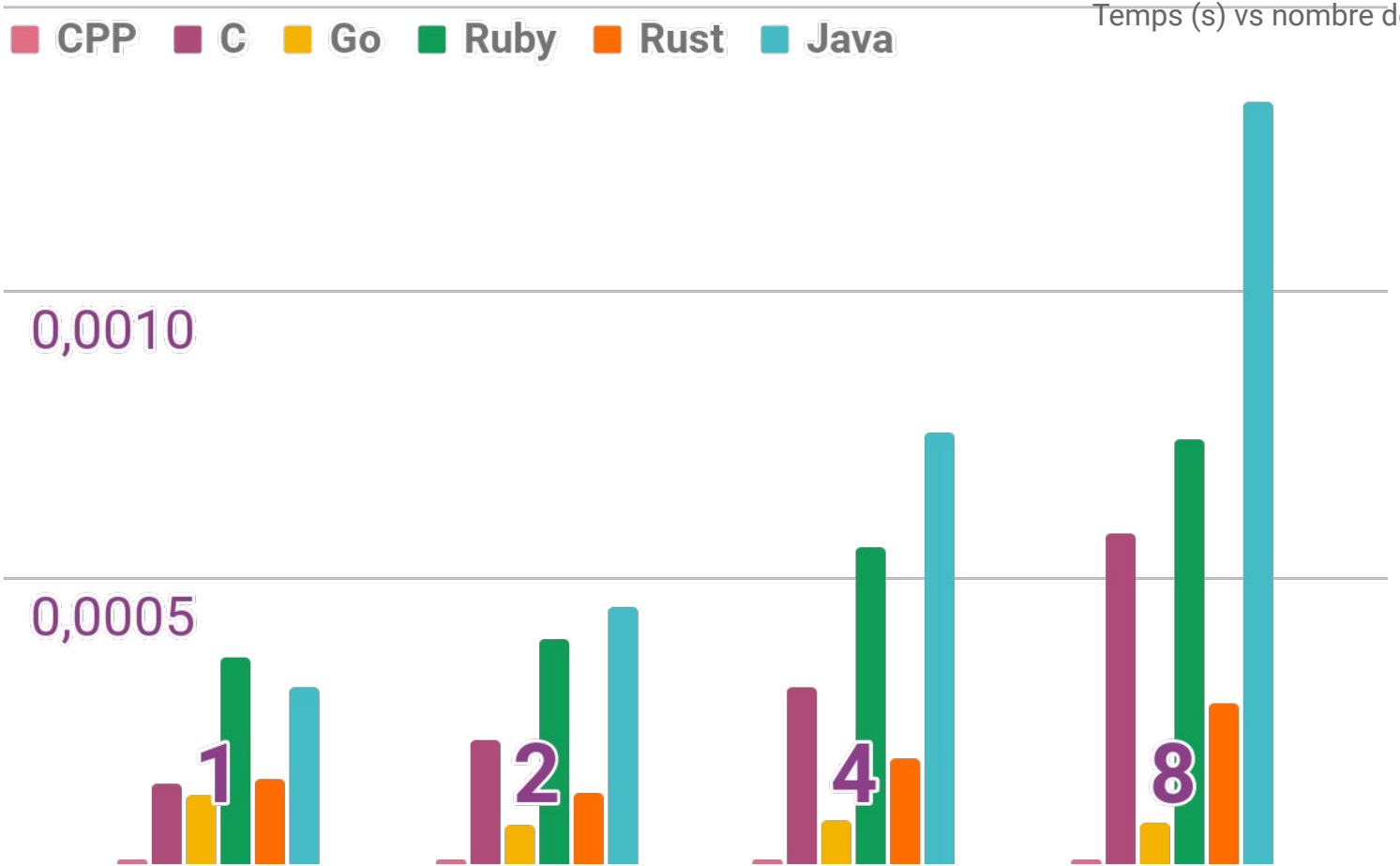
Temps (s) vs nombre de threads



N=1000

Pi - approche parallèle

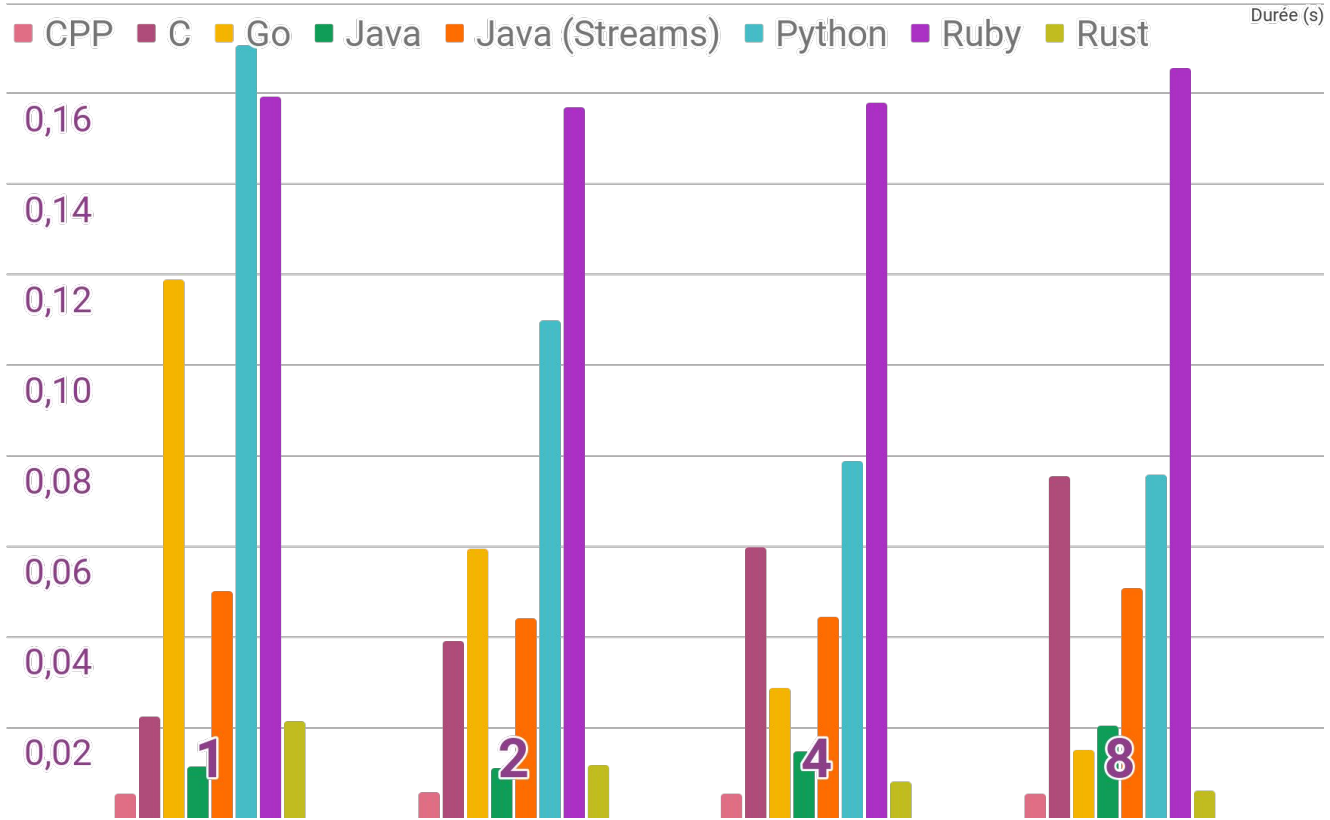
Temps (s) vs nombre de threads



Pi - approche parallèle

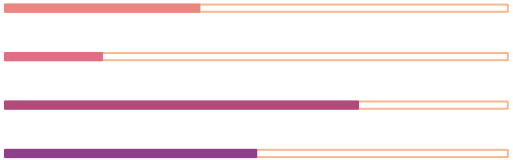
N=1000000

Temps (s) vs nombre de threads

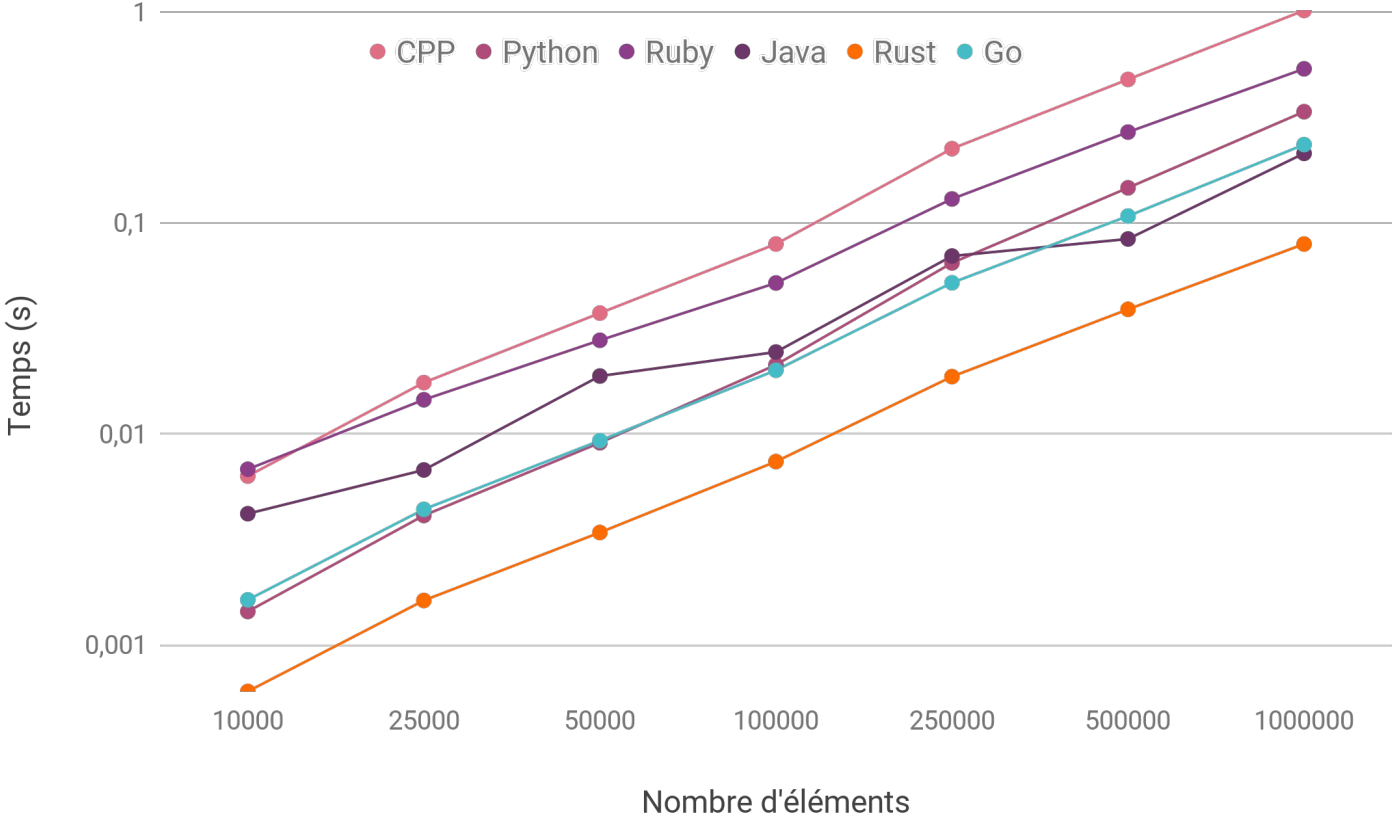


Phase 2: Evaluation avec les outils des langages (l’exemple des tris)

Pourquoi ne pas avoir choisi une seule implémentation d’un tri et l’écrire dans tous les langages ?



Comparaison des durées de calculs en séquentiel



C++ - Boost

- Importance : des pans entiers sont intégrées à chaque version dans la std
- Dispose d'un wrapper autour de pthreads pour le OOP
- Dispose d'une variété d'algorithmes de tri type *State of the Art*
- Implémentations optimisées singlethread (seul quicksort double pivot pour Java)
- Implémentations multithread avec une attention particulière sur l'usage mémoire
- Java utilise un sort-merge

Francisco Tapia
Contribué à optimiser et créer des implémentations de tri sur Boost
Dernier en date: 2016

C++ - Boost

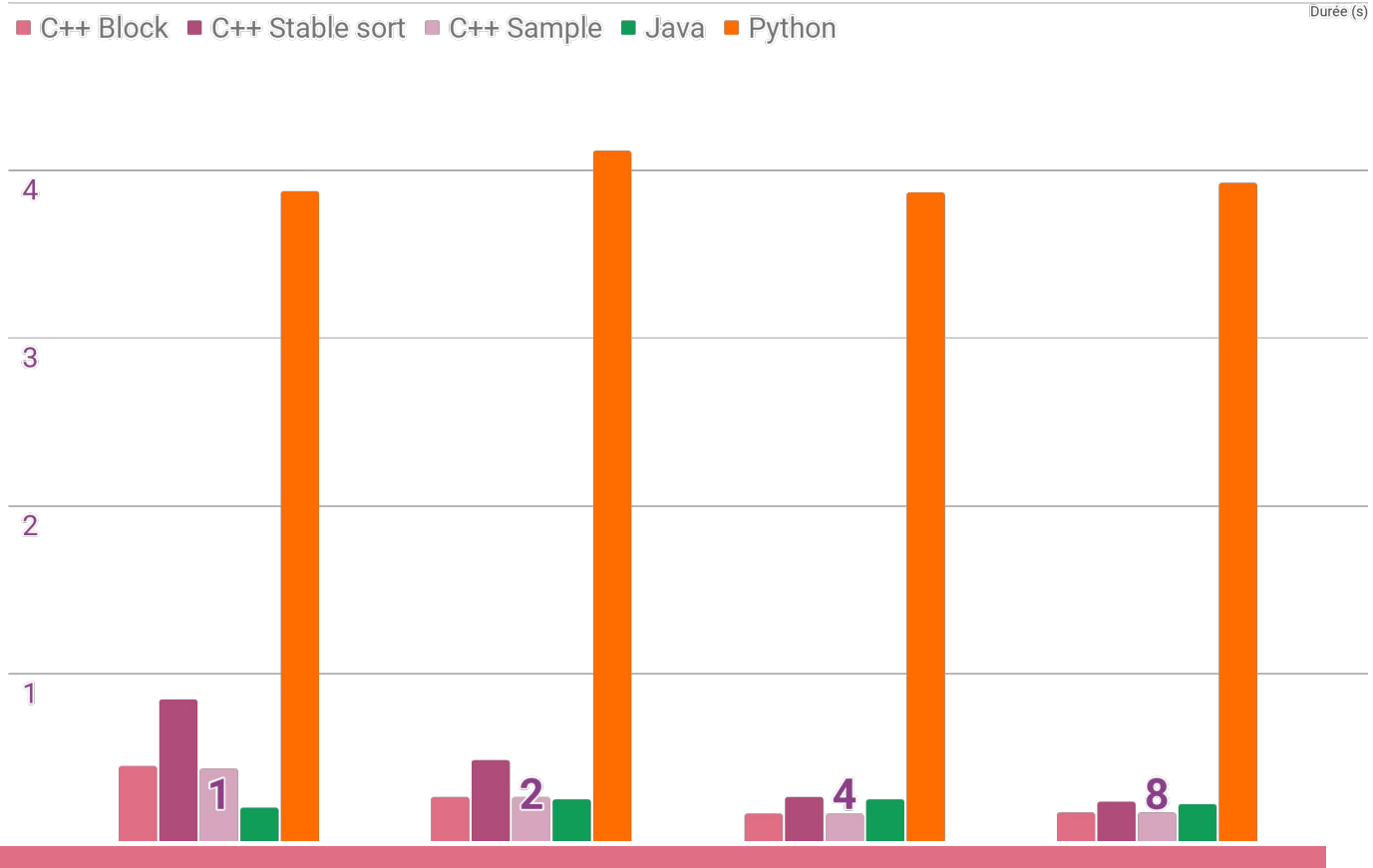
Francisco Tapia
Contribué à optimiser et créer des implémentations de tri sur Boost
Dernier en date: 2016

	Stable	Mémoire additionnelle	Meilleur, moyen, pire cas
block_indirect_sort	non	Block_size * n_threads	N, N LogN , N LogN
sample_sort	oui	N	N, N LogN , N LogN
parallel_stable_sort	oui	N/2	N, N LogN , N LogN

block_size : paramètre interne déterminé en fonction de la taille des objets à trier

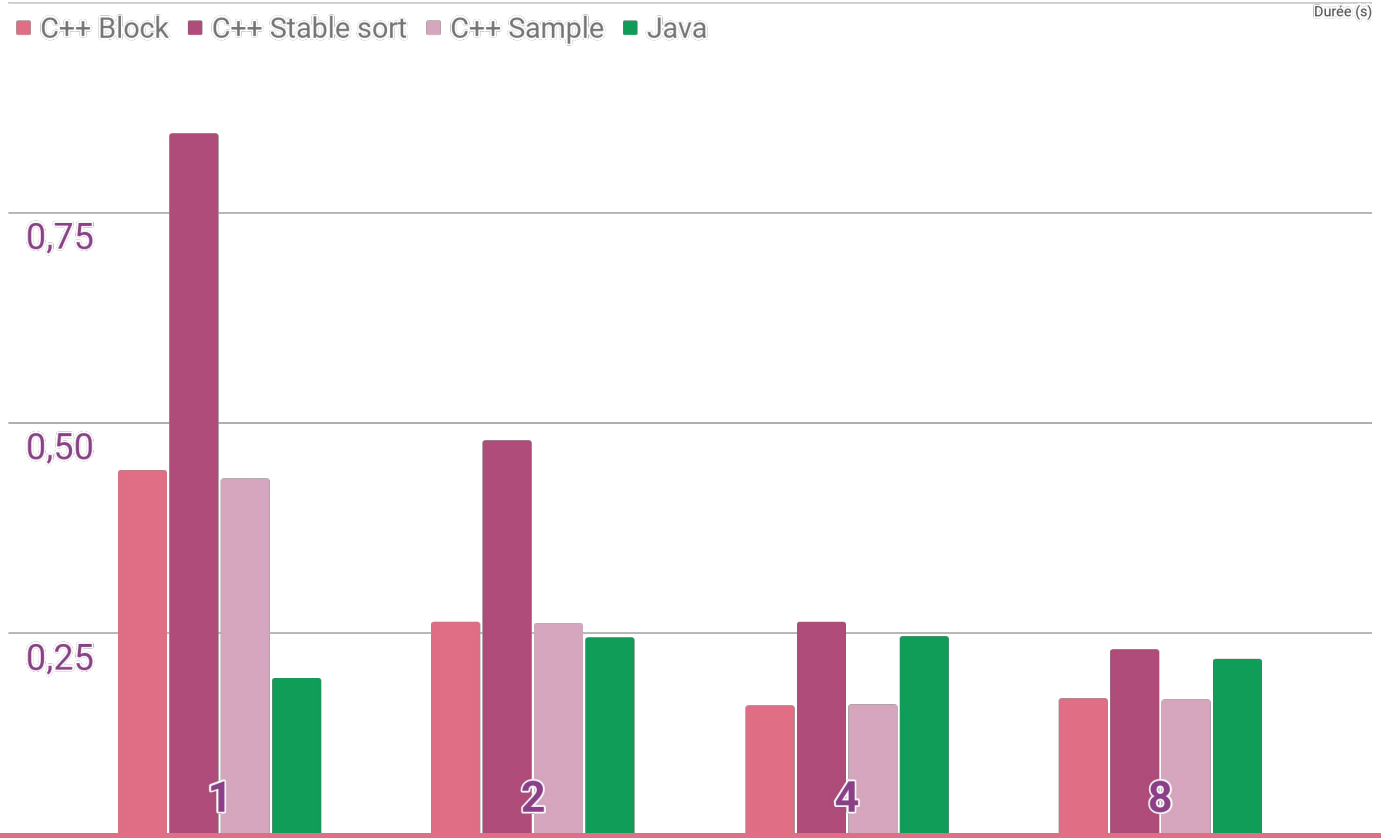
N=1000000

Comparaison des durées de calculs en parallèle

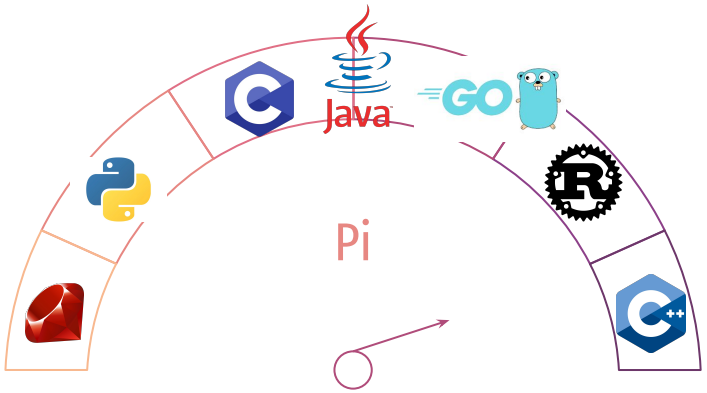


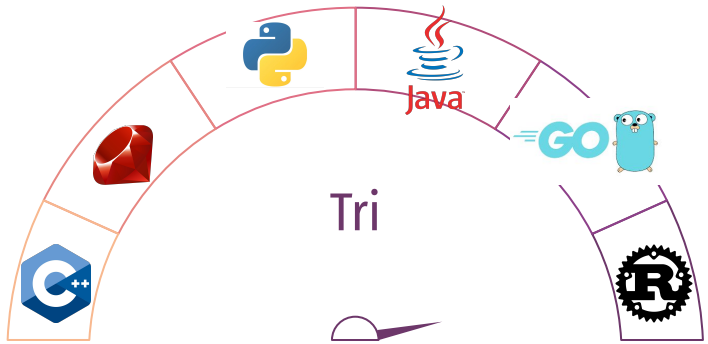
N=1000000

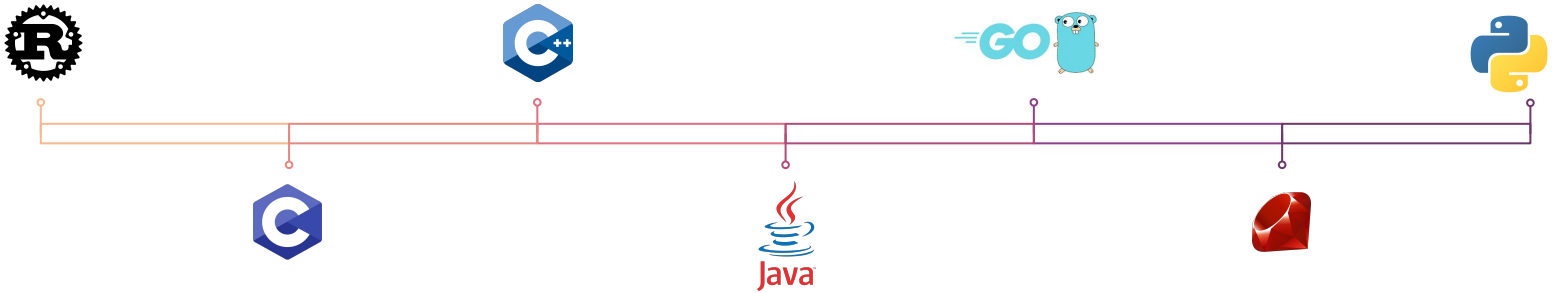
Comparaison des durées de calculs en parallèle (Python exclus)



Résumé


Java
GO
Pi
Rapidité


Java
GO
Tri
Rapidité


Java
Facilité du code

L’approche des langages

L’implémentation performante n’a plus qu'à être trouvée/codée ?

L’implémentation performante n’a plus qu'à être choisie

L’implémentation performante n’a plus qu'à être utilisée

Né pour que l’implémentation soit performante

L’implémentation performante est guidée

L’implémentation performante est celle du langage d’à côté

L’implémentation ?
(Mais au moins le code prend 1 ligne)

Merci !

https://github.com/Matbabs/PerformanceEvaluation

Avez-vous des questions ?

Comparaison de l'efficacité des implémentations

1. La loi d'Amdahl donne la proportion *p* de parallélisme d'une implémentation de *s* threads

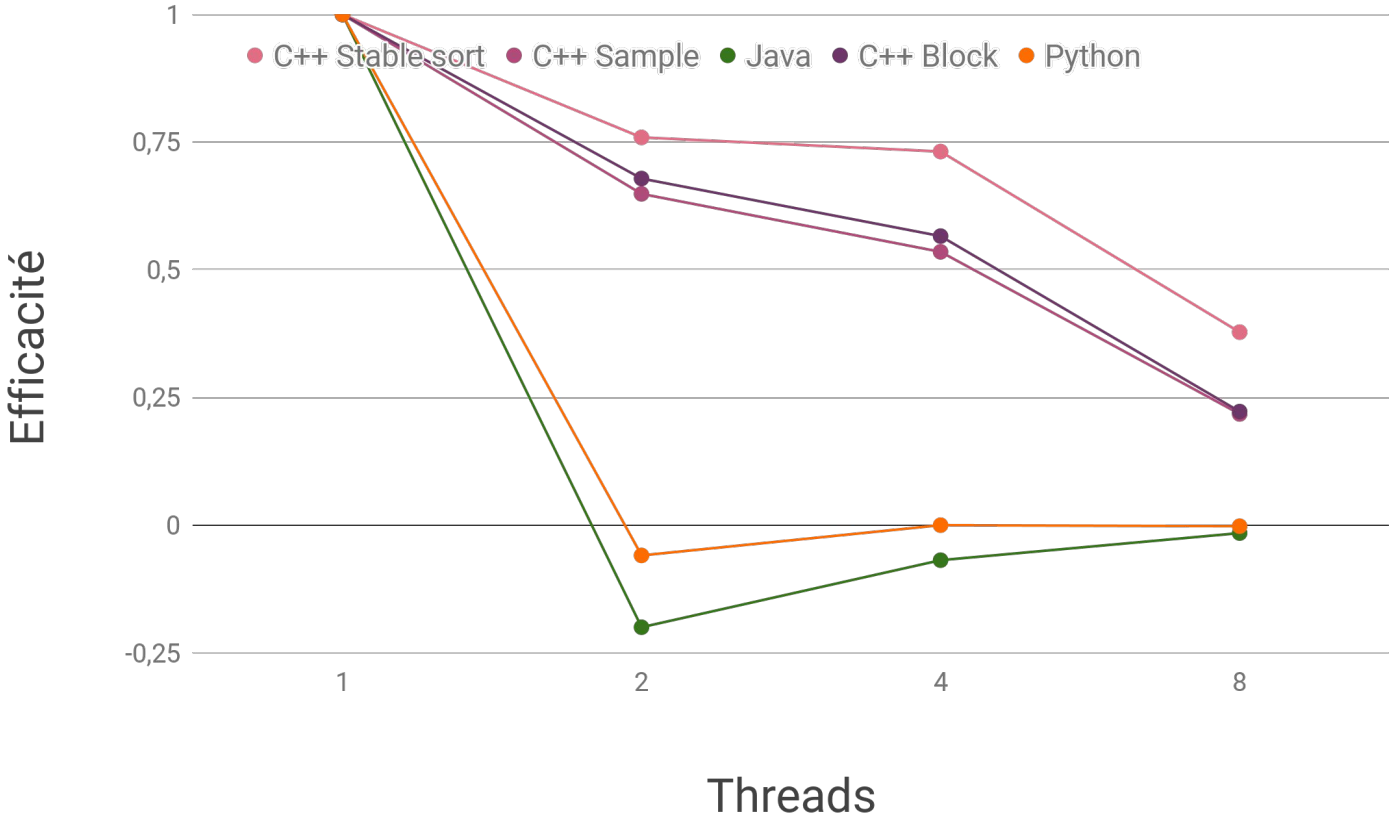
$$S_{\text{latence}}(s) = \frac{1}{1 - p + \frac{p}{s}}$$

2. Elle donne également le speedup de la partie parallèle en enlevant la partie séquentielle
3. L'efficacité est le ratio entre le speedup total *S* et le speedup exclusivement de la partie parallèle *s*

$$\eta = \frac{S}{s}$$

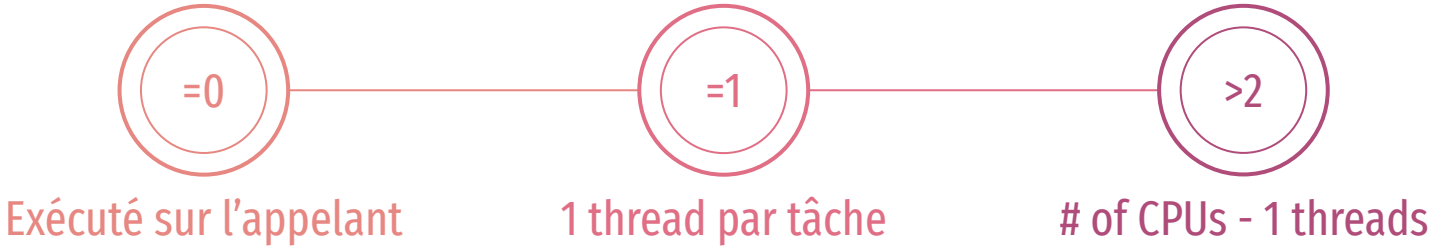
N=1000000

Comparaison de l'efficacité des implémentations



Java - ForkJoin common Pool

- Utilisation des ForkJoinPool, et en particulier d'une instance par défaut : la commonPool
- Dimensionnement suivant les processeurs système
- Réservée aux tâches non bloquantes (par exemple trier un tableau)
- Vise à contrôler la mémoire utilisée par les threads et leur exécution



- Utilisation dans les Streams
- Utilisation dans le parallelSort

Java - sort vs parallelSort

- Sort() en singlethread
- Optimisé pour des petits ensembles
- ParallelSort possède une valeur pivot en-dessous de laquelle il déclenche le sort() singlethread

Éléments **ou** 1 seul coeur pour déclencher sort()