



Parallélisme & Architecture multicores

Compte rendu du TP 1

Nom & Prénom :

Selina CHEGGOUR
Thérèse MBOCK

Professeur :

Cecile BELLEUDY

Master 2 parcours ESTEL

2021/2022

1 Introduction

Le parallélisme consiste, d'une manière générale, à exécuter un algorithme en utilisant plusieurs processeurs plutôt qu'un seul. Par conséquent, un algorithme, parallélisable, pourrait être divisé en plusieurs tâches qui peuvent être exécutées simultanément sur plusieurs processeurs. D'un point de vue technologique, les constructeurs semblent être arrivés devant un mur de fréquence. L'amélioration des performances passe donc aujourd'hui par d'autres évolutions technologiques et, parmi elles, l'augmentation du nombre de cœurs. Un microprocesseur multi-cœur est un processeur possédant plusieurs cœurs physiques. Au fil du temps, le nombre de cœurs ne cesse d'augmenter dans l'objectif d'améliorer toujours plus la puissance des ordinateurs. L'augmentation du nombre de cœurs a résulté en un certain nombre de problèmes. Cette amélioration technique ne suffit donc parfois pas à améliorer de façon significative les performances d'un ordinateur. Des compromis seront établis entre le nombre de coeurs, la consommation et le parallélisme selon l'application.

2 Objectifs

Notre objectif durant ce TP est d'analyser ainsi qu'étudier l'architecture et la consommation d'un nano ordinateur Raspberry Pi 3 constitué d'un processeur à 4 coeurs.

3 Configuration de la machine

3.1 Architecture

Le Raspberry Pi n'est rien d'autre qu'un ordinateur réduit à sa plus simple expression : une unique carte à processeur ARM. Il existe plusieurs générations de Raspberry Pi, et durant ce TP nous utiliserons le modèle 3 B+ (Raspberry Pi 3+) qui est constitué d'un processeur Broadcom BCM2837B0 64 bit à quatre cœurs ARM Cortex-A53 cadencé à 1,4 GHz, une puce CYW43455 supportant le WiFi Dual-band 802.11ac et la version 4.2 du Bluetooth, et enfin d'une prise en charge du power over Ethernet grâce à un élément supplémentaire. Il dispose de pins GPIO qui permettent la connexion de cartes d'extension ou d'autres composants électroniques pour réaliser des montages. Notre Raspberry a pour fréquence minimale 0.6 GHz et pour fréquence maximale 1.4 GHz. Il possède 6 gouverneurs : Ondemand, Schedutil, Powersave, Performance, Conservative, Userspace.

Raspberry Pi 3B+
Broadcom BCM2837B0
Quad core Cortex-A53@1.4GHz
VideoCore IV@250-400MHz
1 Go LPDDR2 SDRAM
4 x USB 2.0
HDMI unique
Pas Support
2.4GHz/5GHz 802.11ac
Wi-Fi , 300Mbps Ethernet , Bluetooth 4.0
5V / 2,5A Micro USB

FIG. 1: Fiche technique du Raspberry Pi 3 B+

3.2 Consommation

Nous avons utilisé le code de la figure suivante composé d'opérations entre tableaux dans une boucle while afin d'effectuer l'analyse de la consommation de la carte en utilisant les 4 coeurs en même temps (avec parallélisme). Cela se fera pour différentes configurations (différents gouverneurs), où le temps et la consommation seront relevés. Ces éléments nous permettront de calculer l'énergie par cœur (temps de compilation x puissance maximale, où la puissance maximale = tension (5V) x courant).

```

import time
import random
T1 = time.time()

A = [random.randint(0,100) for i in range(100)]
B = [random.randint(0,100) for i in range(100)]
C = []
for j in range (50000):
    for i in range (100):
        C.append(A[i]*B[i])
T2 = time.time()
print(T2-T1)

```

Pour commencer notre analyse, nous avons configuré notre carte sur le gouverneur **on-demand**. Cela nous a permis de le caractériser.
Les caractéristiques relevées sont les suivantes :

```

1400000 40485
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat time_in_state
600000 273354
700000 23706
800000 9286
900000 2683
1000000 1262
1100000 0
1200000 0
1300000 0
1400000 44446
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat trans_table
From : To
: 600000 700000 800000 900000 1000000 1100000 1200000 1300000 1400000
600000: 0 1559 591 99 35 0 0 0 85
700000: 1535 0 153 72 20 0 0 0 74
800000: 581 153 0 41 12 0 0 0 46
900000: 96 48 46 0 5 0 0 0 50
1000000: 55 24 12 7 0 0 0 0 23
1100000: 0 0 0 0 0 0 0 0 0
1200000: 0 0 0 0 0 0 0 0 0
1300000: 0 0 0 0 0 0 0 0 0
1400000: 182 70 31 26 49 0 0 0 0
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat time_in_state
600000 274146
700000 23740
800000 9326
900000 2694
1000000 1262
1100000 0
1200000 0
1300000 0
1400000 48060
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $

```

FIG. 2: Temps & freq relevés

Nous relevons un temps de 23.80 secondes pour la compilation du code. En ce qui concerne la consommation, elle vaut 0.39 A à la fréquence minimale (0.6 GHz) et 0.79 A à la fréquence maximale (1.4 GHz). Nous avons donc une énergie maximale de 94.01 J.

Nous avons configuré ensuite notre carte sur le gouverneur **powersave**. Puis, nous avons compilé le code de détection de contour. Cela nous a permis de le caractériser. Les caractéristiques relevées sont les suivantes :

```

pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat time_in_state
36 29 50 1 3 0 0 0
600000 430667
700000 34476
800000 14414
900000 2942
1000000 1432
1100000 36
1200000 88
1300000 38
1400000 52631
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat trans_table
From : To
: 600000 700000 800000 900000 1000000 1100000 1200000 1300000 1400000
600000: 0 2552 1058 107 36 0 0 0 95
700000: 2522 0 179 72 22 0 0 0 78
800000: 1055 174 0 52 14 0 0 0 50
900000: 105 51 58 0 13 0 0 0 23
1000000: 56 25 14 15 0 7 0 0 0
1100000: 0 0 0 2 4 0 6 0 0
1200000: 0 0 0 0 1 4 0 5 0
1300000: 0 0 0 0 0 1 3 0 0
1400000: 110 71 36 29 50 0 0 1 0
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat time_in_state
600000 434841
700000 34476
800000 14414
900000 2942
1000000 1432
1100000 36
1200000 88
1300000 38
1400000 52631
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $

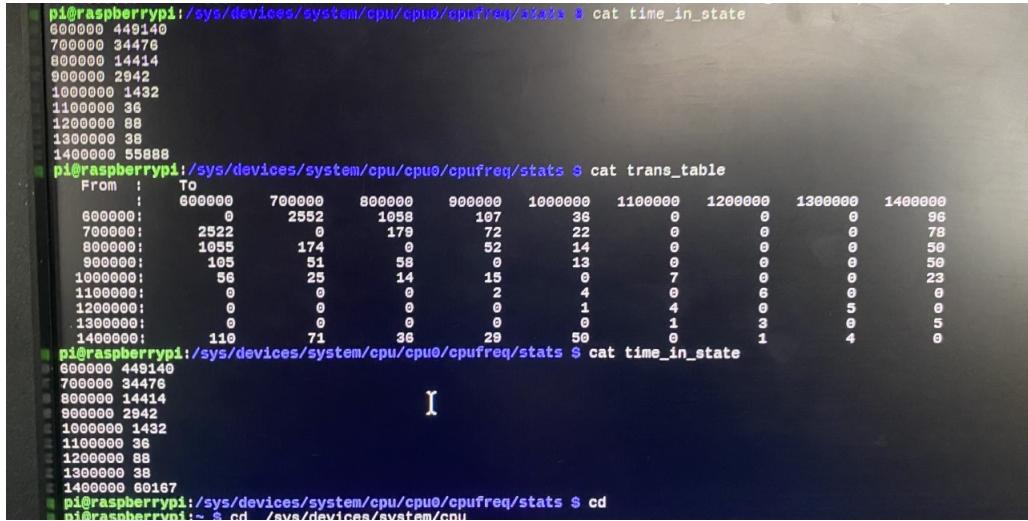
```

FIG. 3: Temps & freq relevés

Nous relevons un temps de 27.49 secondes pour la compilation du code. En ce qui concerne

la consommation, elle vaut 0.39 A à la fréquence minimale (0.6 GHz) et 0.59 A à la fréquence maximale (1.4 GHz). Nous avons donc une énergie maximale de 81.09 J.

Nous avons configuré par la suite notre carte sur le gouverneur **performance**. Puis, nous avons compilé le code de détection de contour. Cela nous a permis de le caractériser. Les caractéristiques relevées sont les suivantes :



The screenshot shows a terminal session on a Raspberry Pi. The user runs several commands to gather performance data:

- `pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat time_in_state`: Displays the time spent in various states (600000, 700000, 800000, 900000, 1000000, 1100000, 1200000, 1300000, 1400000) in microseconds. The output includes values like 600000: 449140, 700000: 34476, etc.
- `pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat trans_table`: Displays the transition table between frequencies. The table has columns for From and To frequencies (e.g., 600000, 700000, ..., 1400000) and rows for transitions. Values range from 0 to 2552.
- `pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat time_in_state`: Shows the same time in state data as the first command.
- `pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cd`: Changes the directory back to the root.

FIG. 4: Temps & freq relevés

Nous relevons un temps de 26.25 secondes pour la compilation du code. En ce qui concerne la consommation, elle vaut 0.43 A à la fréquence minimale (0.6 GHz) et 0.8 A à la fréquence maximale (1.4 GHz). Nous avons donc une énergie maximale de 105 J.

Pour finir, nous avons configuré notre carte sur le gouverneur **conservative**. Puis, nous avons compilé le code de détection de contour. Cela nous a permis de le caractériser. Les caractéristiques relevées sont les suivantes :

```

pi@raspberrypi:~$ cat /sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state
600000 404936
700000 34371
800000 14329
900000 2873
1000000 1239
1100000 0
1200000 0
1300000 0
1400000 49360
pi@raspberrypi:~$ cat /sys/devices/system/cpu/cpu0/cpufreq/stats/trans_table
From : To
: 600000 700000 800000 900000 1000000 1100000 1200000 1300000 1400000
600000: 0 2542 1658 107 36 0 0 0 0
700000: 2512 0 172 72 22 0 0 0 95
800000: 1655 168 0 47 14 0 0 0 0
900000: 105 50 54 0 7 0 0 0 0
1000000: 56 25 14 11 0 2 0 0 0
1100000: 0 0 0 0 2 0 2 0 0
1200000: 0 0 0 0 0 0 0 0 0
1300000: 0 0 0 0 0 0 0 2 0
1400000: 110 71 36 29 50 0 0 0 0
pi@raspberrypi:~$ cat /sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state
600000 406300
700000 34394
800000 14365
900000 2886
1000000 1364
1100000 8
1200000 22
1300000 16
1400000 52561
pi@raspberrypi:~$
```

FIG. 5: Temps & freq relevés

Nous relevons un temps de 25.12 secondes pour la compilation du code. En ce qui concerne la consommation, elle vaut 0.39 A à la fréquence minimale (0.6 GHz) et 0.77 A à la fréquence maximale (1.4 GHz). Nous avons donc une énergie maximale de 96.71 J.

Remarques :

- Le gouverneur le plus consommateur d'énergie est le Performance tandis que le moins consommateur est le Powersave.
- Le gouverneur le plus rapide est le Ondemand tandis que le plus lent est le Powersave.
- On a noté des sauts réguliers entre la fréquence minimale et maximale pour le gouverneurs Ondemand.
- Pour le Powersave, la fréquence de la carte est restée stable à sa fréquence minimale (0.6 GHz).
- Pour le Performance, la fréquence de la carte est restée stable à sa fréquence maximale (1.4 GHz).
- On a noté des sauts réguliers entre différentes fréquences pour le gouverneur Conservative.

4 Application d'un Filtre gris

4.1 Sans parallélisme (taux de parallélisme : 0)

Dans cette partie, une fonction appliquant un filtre gris sur une image donnée a été codée puis exécutée sans parallélisme le tout sur le gouverneur Ondemand. Le code de cette fonction est joint à ce document PDF sous le nom **Filtre_gris_sans_parallelisme.py**. A gauche, nous avons l'image avant l'utilisation du filtre. A droite, nous avons l'image filtrée.



FIG. 6: Image avant application du filtre



FIG. 7: Image après application du filtre

Nous relevons un temps de 33.09 secondes pour la compilation du code. En ce qui concerne la consommation maximale, elle vaut 0.59 A. Nous avons donc une énergie maximale de 97.6J.

4.2 Avec parallélisme

Dans un second temps, une fonction appliquant un filtre gris sur une image donnée a été codée puis exécutée avec différents taux de parallélisme.

4.2.1 Taux de parallélisme : 2

Le code correspondant au taux de parallélisme 2 est joint à ce document PDF sous le nom **Filtre_gris_avec_parallelisme_taux2.py**.

Pour ce faire, le code va couper l'image donnée en 2 sous images puis activer 2 coeurs pour filtrer chacun une partie de l'image.



FIG. 8: Sous image 1 avant application du filtre



FIG. 9: Sous image 2 avant application du filtre

Les 2 images seront ensuite filtrées puis recollées en une image. Les figures suivantes démontrent les résultats obtenus à chaque étape.



FIG. 10: Sous image 1 après application du filtre



FIG. 11: Sous image 2 après application du filtre



FIG. 12: Reconstitution de l'image

Nous relevons un temps de 14.70 secondes pour la compilation du code. En ce qui concerne la consommation maximale, elle vaut 0.77 A. Nous avons donc une énergie maximale de 56.59J.

4.2.2 Taux de parallélisme : 4

Le code correspondant au taux de parallélisme 4 est joint à ce document PDF sous le nom **Filtre_gris_avec_parallelisme_taux4.py**.

Ce code effectue le même processus que précédemment mais pour un taux de parallélisme 4. Dans un premier temps, l'image sera coupée en 4 sous images.



FIG. 13: Sous image 1 avant application du filtre



FIG. 14: Sous image 2 avant application du filtre



FIG. 15: Sous image 3 avant application du filtre



FIG. 16: Sous image 4 avant application du filtre

Dans un second temps, les 4 sous images seront filtrées chacune via un cœur du processeur par le filtre gris. Enfin, les sous images filtrées seront regroupées en une seule image.



FIG. 17: Sous image 1 après application du filtre



FIG. 18: Sous image 2 après application du filtre



FIG. 19: Sous image 3 après application du filtre



FIG. 20: Sous image 4 après application du filtre



FIG. 21: Reconstitution de l'image

Nous relevons un temps de 18.47 secondes pour la compilation du code. En ce qui concerne la consommation maximale, elle vaut 0.81 A. Nous avons donc une énergie maximale de 74.80J.

Remarques :

- Le taux 2 de parallélisme est le plus rapide.
- Le taux 4 de parallélisme est le plus consommateur.

5 Détection de contours

Dans cette dernière partie, nous étudions un traitement vidéo capable de faire de la détection de contours. Notre étude s'effectuera sur un seul frame. En connaissant le nombre de frame dont on dispose dans une séquence vidéo, on pourra généraliser notre étude au traitement vidéo.

Cas de figure :

Un algorithme capable de détecter une présence pour de la vidéo surveillance a été proposé pour un frame.

Le code correspondant au taux de parallélisme 4 est joint à ce document PDF sous le nom **Detection_de_contour.py**.

Nous avons d'abord une image de référence où aucun objet n'a été signalé. Sur la seconde image, une voiture passe, on pourra donc détecter un objet comme sur une vidéo surveillance.



FIG. 22: Image de référence



FIG. 23: Passage de la voiture

Les figures suivantes illustrent les étapes pour une détection de contours à l'aide du code joint.

Étape 01 :

Le code va appliquer un filtre gris aux 2 images données.



FIG. 24: Image de référence

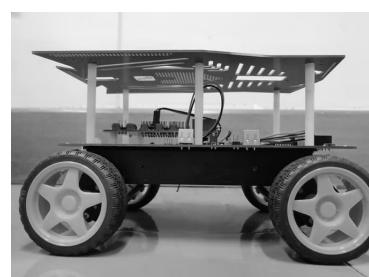


FIG. 25: Passage de la voiture

Étape 02 :

Le code va découper les 2 images filtrées en gris en 4 sous images.

Découpage de l'image de référence :



FIG. 26: Sous image de référence 1



FIG. 27: Sous image de référence 2



FIG. 28: Sous image de référence 3



FIG. 29: Sous image de référence 4

Découpage de l'image passage de la voiture :



FIG. 30: Sous image de la voiture 1



FIG. 31: Sous image de la voiture 2



FIG. 32: Sous image de la voiture 3



FIG. 33: Sous image de la voiture 4

Étape 03 :

Le code va détecter les contours entre les 8 sous images filtrées en gris.

La détection de contours se définit par la différence de pixels à chaque niveau de couleur (R, G et B) entre l'image filtrée de référence et l'image filtrée du passage de la voiture. La moyenne des 3 quantités de couleur de chaque pixel sera comparée à un seuil m . Lorsqu'elle est supérieure à ce seuil le pixel résultant sera blanc. Dans le cas contraire, le pixel résultant sera noir. C'est de cette façon que nous obtiendrons le contour de la voiture.

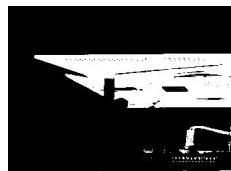


FIG. 34: Détection de contour sous image1



FIG. 35: Détection de contour sous image2



FIG. 36: Détection de contour sous image3



FIG. 37: Détection de contour sous image4

Étape 04 :

Le code va reconstituer l'image avec le contour détecté.

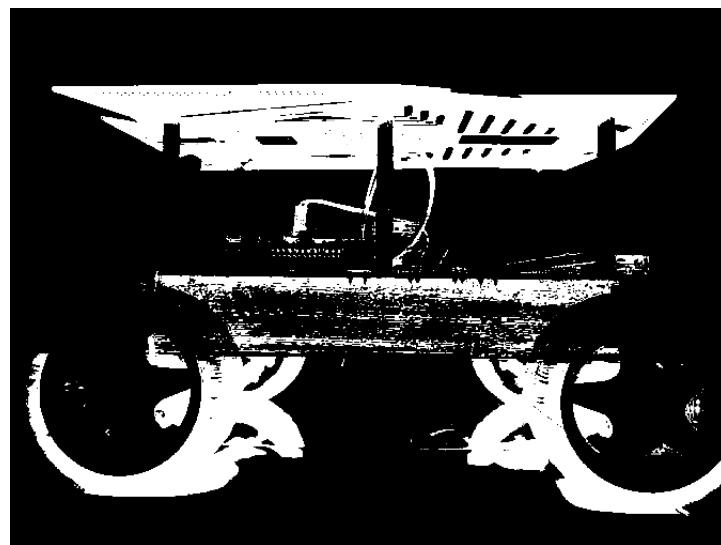


FIG. 38: Reconstitution de l'image après détection de contour

La présence de la voiture a été détectée à l'aide du code proposé durant ce TP.

6 Détection de contours et analyse des gouverneurs

Dans la première section du TP, nous avons fait connaissance des gouverneurs disponibles sur notre carte Raspberry Pi. Dans cette partie, nous allons faire une analyse de l'algorithme de détection de contours à travers les différents gouverneurs vus précédemment.

Pour commencer notre analyse, nous avons configuré notre carte sur le gouverneur **on-demand**. Puis, nous avons compilé le code de détection de contour. Cela nous a permis de le caractériser.

Les caractéristiques relevées sont les suivantes :

```
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat time_in_state
1300000: 0 38 48 54 64
1400000: 6 0 0 0 0
600000 25670
700000 2500
800000 1383
900000 1148
1000000 514
1100000 0
1200000 0
1300000 0
1400000 541684

pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat trans_table
From : To
       : 600000 700000 800000 900000 1000000 1100000 1200000 1300000 1400000
600000: 0 102 28 23 4 0 0 0 31
700000: 113 0 23 15 7 0 0 0 37
800000: 35 32 0 7 3 0 0 0 46
900000: 24 21 10 0 3 0 0 0 49
1000000: 7 2 8 8 0 0 0 0 24
1100000: 0 0 0 0 0 0 0 0 0
1200000: 0 0 0 0 0 0 0 0 0
1300000: 0 0 0 0 0 0 0 0 0
1400000: 0 38 48 54 32 0 0 0 0

pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat time_in_state
600000 25974
700000 2500
800000 1383
900000 1159
1000000 527
1100000 0
1200000 0
1300000 0
1400000 545734

pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $
```

FIG. 39: Temps & freq relevés

Nous relevons un temps de 29.97 secondes pour la compilation du code. En ce qui concerne la consommation maximale, elle vaut 0.81 A. Nous avons donc une énergie maximale de 121.37J.

Nous avons configuré ensuite notre carte sur le gouverneur **powersave**. Puis, nous avons compilé le code de détection de contour. Cela nous a permis de le caractériser.

Les caractéristiques relevées sont les suivantes :

```
pi@raspberrypi:/sys/devices/system/cpu/cpufreq/stats $ cat time_in_state
600000 81706
700000 3067
800000 1477
900000 1189
1000000 547
1100000 0
1200000 0
1300000 0
1400000 546099
pi@raspberrypi:/sys/devices/system/cpu/cpufreq/stats $ cat trans_table
  From : To
    : 600000 700000 800000 900000 1000000 1100000 1200000 1300000 1400000
600000:   0 151   35   25     5     0     0     0     0     33
700000: 162   0 23 15     7     0     0     0     0     40
800000: 40   32   0   8     4     0     0     0     0     41
900000: 26   22 10   0     3     0     0     0     0     49
1000000: 8   2   9   8     0     0     0     0     0     24
1100000: 0   0   0   0     0     0     0     0     0     0
1200000: 0   0   0   0     0     0     0     0     0     0
1300000: 0   0   0   0     0     0     0     0     0     0
1400000: 13   40   48   54    32     0     0     0     0
pi@raspberrypi:/sys/devices/system/cpu/cpufreq/stats $ cat time_in_state
600000 87785
700000 3067
800000 1477
900000 1189
1000000 547
1100000 0
1200000 0
1300000 0
1400000 546099
pi@raspberrypi:/sys/devices/system/cpu/cpufreq/stats $
```

FIG. 40: Temps & freq relevés

Nous relevons un temps de 57.36 secondes pour la compilation du code. En ce qui concerne la consommation maximale, elle vaut 0.59 A. Nous avons donc une énergie maximale de 169.21J.

Nous avons configuré par la suite notre carte sur le gouverneur **performance**. Puis, nous avons compilé le code de détection de contour. Cela nous a permis de le caractériser. Les caractéristiques relevées sont les suivantes :

```
pi@raspberrypi:/sys/devices/system/cpu/cpufreq/stats $ cat time_in_state
600000 102606
700000 3067
800000 1477
900000 1189
1000000 547
1100000 0
1200000 0
1300000 0
1400000 571961
pi@raspberrypi:/sys/devices/system/cpu/cpufreq/stats $ cat trans_table
  From : To
    : 600000 700000 800000 900000 1000000 1100000 1200000 1300000 1400000
600000:   0 151   35   25     5     0     0     0     0     34
700000: 162   0 23 15     7     0     0     0     0     40
800000: 40   32   0   8     4     0     0     0     0     41
900000: 26   22 10   0     3     0     0     0     0     49
1000000: 8   2   9   8     0     0     0     0     0     24
1100000: 0   0   0   0     0     0     0     0     0     0
1200000: 0   0   0   0     0     0     0     0     0     0
1300000: 0   0   0   0     0     0     0     0     0     0
1400000: 13   40   48   54    32     0     0     0     0
pi@raspberrypi:/sys/devices/system/cpu/cpufreq/stats $ cat time_in_state
600000 102606
700000 3067
800000 1477
900000 1189
1000000 547
1100000 0
1200000 0
1300000 0
1400000 575967
pi@raspberrypi:/sys/devices/system/cpu/cpufreq/stats $
```

FIG. 41: Temps & freq relevés

Nous relevons un temps de 29.70 secondes pour la compilation du code. En ce qui concerne la consommation maximale, elle vaut 0.62 A. Nous avons donc une énergie maximale de 92.07J.

Pour finir, nous avons configuré notre carte sur le gouverneur **conservative**. Puis, nous avons compilé le code de détection de contour. Cela nous a permis de le caractériser. Les caractéristiques relevées sont les suivantes :

```
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat time_in_state
600000 173145
700000 6024
800000 2900
900000 1887
1000000 1057
1100000 498
1200000 375
1300000 386
1400000 624907
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat trans_table
From : To
      : 600000 700000 800000 900000 1000000 1100000 1200000 1300000 1400000
600000:   0    715   203   91    59   21    32    36    134
700000: 758   0     81    32   19     8    12     7    89
800000: 260   113   0     26   14     4     9     7    64
900000: 103   44    26   0     16    15     5     6    87
1000000: 44    21    29   27   0     36     4     13    47
1100000: 19    12    15   14    32   0     15     6    24
1200000: 16    9     11   15    20   16     0     5    42
1300000: 12    9     13   14    14   9     13     0    49
1400000: 79    74   119   83   47   28    44    53     0
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $ cat time_in_state
600000 173517
700000 6054
800000 2920
900000 1896
1000000 1072
1100000 503
1200000 383
1300000 391
1400000 628295
pi@raspberrypi:/sys/devices/system/cpu/cpu0/cpufreq/stats $
```

FIG. 42: Temps & freq relevés

Nous relevons un temps de 30.26 secondes pour la compilation du code. En ce qui concerne la consommation maximale, elle vaut 0.84 A. Nous avons donc une énergie maximale de 127.09J.

- Le gouverneur le plus consommateur d'énergie est le Powersave tandis que le moins consommateur est le Performance.
- Le gouverneur le plus rapide est le Performance tandis que le plus lent est le Powersave.
- Nous observons les mêmes remarques que précédemment (avec le code de la boucle while) au niveau des transtables.

7 Conclusion

Ce TP nous a permis d'étudier et d'analyser une génération de Raspberry : 3 B+. Comme nous avons visualisé les conséquences du parallélisme à travers l'étude de différentes configurations et leur consommation. Sur ce, pour les différentes applications, des compromis entre le temps d'exécution et la consommation devraient être établis afin de d'avoir une configuration optimale pour ces dernières.