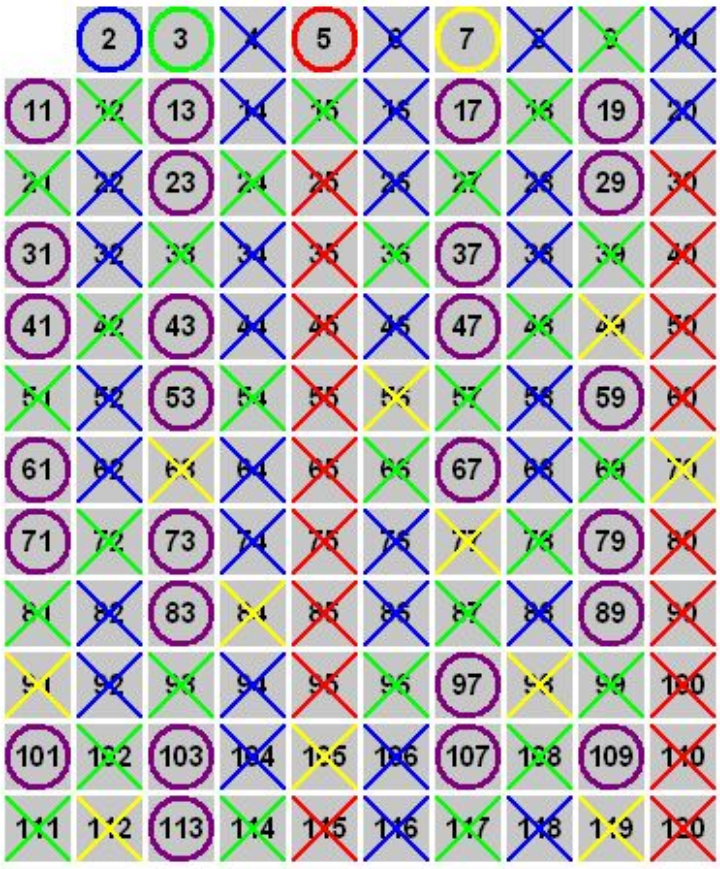
Semestre P23

[SR02] Devoir 3 : Crible d'Ératosthène

Marie Duprat, Nicolas Taupin



Prime numbers

2	3	5	7
11	13	17	19
23	29	31	37
41	43	47	53
59	61	67	71
73	79	83	89
97	101	103	107
109	113		

Responsable de l'UV : Hicham Lakhlef

Table des matières

1 Tache 1

1.1 Déroulement de l'algorithme

En déroulant l'algorithme avec n = 20, on obtient ce résultat :

Initialisation de A :

i	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A[i]	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V

i = 2 :

i	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A[i]	V	V	F	V	F	V	F	V	F	V	F	V	F	V	F	V	F	V	F

i = 3 :

i	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A[i]	V	V	F	V	F	V	F	F	F	V	F	V	F	F	F	V	F	V	F

i = 4 :

i	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A[i]	V	V	F	V	F	V	F	F	F	V	F	V	F	F	F	V	F	V	F

Output : 2, 3, 5, 7, 11, 13, 17, 19

1.2 Bornes des boucles de l'algorithme

La boucle de l'intérieure de l'algorithme commence à i^2, et pas à 0 ou i, car le premier multiple de i non traité pour le moment est toujours i^2. En effet, on commence par éliminer les multiples de 2 (le premier étant $2^2 = 4$), et une fois arrivé à i = 3, les multiples de 3 étant sous la forme $m_3 = 3 * i$ avec $i \in [1; +\infty]$, et le cas i = 2 ayant déjà été traité, on peut commencer avec $m_3 = 3 * 3 = 3^2$. On applique le même raisonnement pour les autres entiers i : les cas antérieurs à i^2 ont à chaque fois déjà été traités.

La première boucle s'exécute jusqu'à $\sqrt{n}$ car, d'après une propriété mathématiques, si un entier naturel n n'admet aucun diviseur premier inférieur ou égal à $\sqrt{n}$, alors n est premier. Si $\sqrt{n}$ n'est pas un entier, on exécute la boucle jusqu'au dernier entier strictement inférieur à n, soit la partie entière de $\sqrt{n}$.

2 Tache 2

Il était question ici d'implémenter une version séquentielle du crible d'Ératosthène.

Pour ce faire, nous récupérons en ligne de commande la valeur n correspondant à l'entier limite jusqu'auquel l'utilisateur souhaite obtenir les nombres premiers. Ensuite, après avoir initialisé le tableau de booléen (en prenant la valeur 1 pour vrai et 0 pour faux), on implémente la double boucle afin de mettre à jour les valuers du tableau.

Enfin, on affiche les nombres premiers, correspondant aux éléments du tableau qui ont encore pour valeur 1.
Ainsi, on obtient le code suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main (int argc, char* argv []) {
    // Tache 2

    // Implementation du crible en version sequentielle

    int input;

    if (argc < 2) {
        printf("Veuillez fournir un entier n en argument.\n");
        return 1;
    }

    input = atoi(argv[1]);
    int array[input];

    // Initialisation du tableau booleen (1 pour vrai, 0 pour faux)
    for (int i = 2; i<input; i++) {
           array[i] = 1;
    }

    for (int j = 0; j<sqrt(input); j++) {
        if (array[j] == 1) {
            int m = j*j;
            while (m<input) {
                array[m] = 0;
                m += j;
            }
        }
    }
    for (int k = 2; k<input; k++) {
        if (array[k] == 1) {
            printf("%d\n", k);
        }
    }

    return 0;

}
```

3 Tache 3

Dans la tache 3, il fallait implémenter une version parallèle de notre algorithme, en distribuant l'exploration de $i^2, i^2 + i, ..., n$ sur k threads. Pour réaliser cette tache, nous avons implémenté deux fonctions : thread_execution() et creat_thread().

Dans le main de notre programme, après avoir récupéré les valeurs de k et de n, passées en paramètres, nous créons les threads qui seront utilisés en appelant la fonction creat_thread(k). Dans la fonction creat_thread, nous créeons k threads à l'aide de la fonction pthread_create(). Nous associons à chaque thread un index, que l'on appelle numéro dans notre cas, et qui servira plus tard pour l'attribution de chaque thread à son "segment de travail". Nous passons cet index en argument. Enfin, nous associons à chaque thread la fonction thread_execution() que nous allons détailler maintenant.

```
void creat_thread(int k) {
        //on cree k threads
        for (int i = 0; i<k; i++) {
                int* numero = malloc(sizeof(int));
                *numero = i;
                if ((pthread_create(&(thread_array[i]), NULL, thread_execution
                    , numero))) {
                        fprintf(stderr, "pthread_create error \n");
                        exit(EXIT_FAILURE);
                }

        }

}
```

Ainsi, dans la fonction thread_execution, on peut retrouver le calcul de notre crible. Mais ce calcul sera réparti sur tous les threads que nous avons crée.

Pour faire ceci, pour chaque thread, nous calculons une plage de calcul délimité par borne inférieur (lower_bound) et une borne supérieur (upper_bound). Chaque thread sera chargé de trouver dans le tableau final les nombres premiers entre ces deux bornes.

Pour pouvoir réaliser cela, pour chaque j allant de 2 à $\sqrt{n}$, nous calculons le nombre d'éléments qu'il y a (par élément nous entendons le nombre de multiples de j qu'il y a entre j^2 et $\sqrt{n}$.
Nous savons que au total, notre programme devra analyser chacun de ces éléments. Il reste à répartir ces éléments entre nos threads. Pour cela, nous divisons ce nombre d'éléments par le nombre de threads puis nous attribuons des bornes inférieures et supérieures en fonction de l'index de chaque thread. Ainsi pour chaque j, chaque thread aura une plage de calcul différentes (car chaque index est différent et propre à un thread). Enfin, chaque thread itère dans le tableau de booléens et met à 0 les mutliples de j entre les deux bornes.

```
void *thread_execution(void *num) {
        int lower_bound;
        int upper_bound;
        int value = *(int *)num;

        for (int j = 2; j < sqrt(n); j++) {

                //Calcul slices
                int nb_elem = ceil((n-(j*j))/j);

                lower_bound = ceil(j * j + (nb_elem * value / k) * j);
                upper_bound = ceil(j * j + (nb_elem * (value + 1) / k) * j);

                for (int h = lower_bound; h <= upper_bound; h += j) {
                        array[h] = 0;
                }
        }

        pthread_exit(NULL);

}
```

Dans le main, on attend la terminaison de tous les threads avec pthread_join() avant d'afficher le contenu du tableau de booléens vrais (les nombres premiers donc).
Ainsi, on obtient :

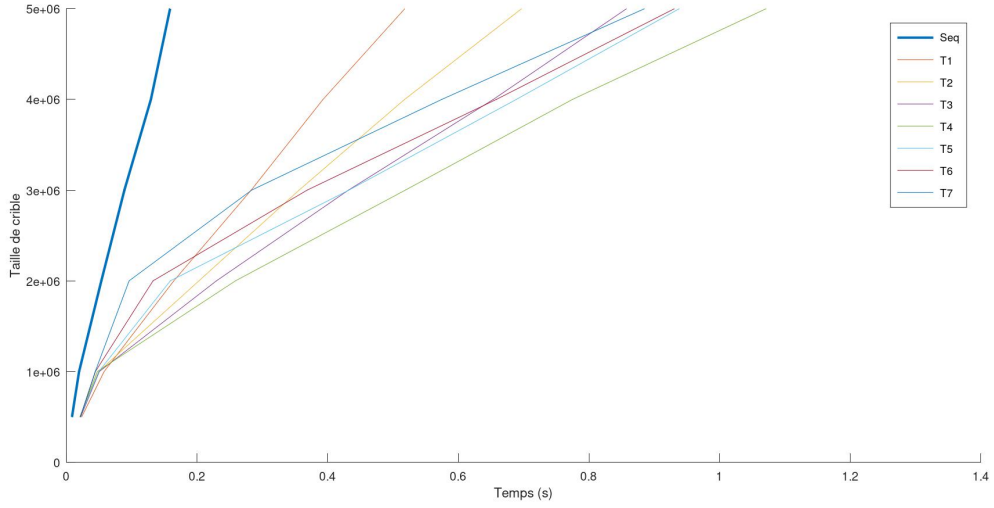
```
void creat_thread(int k);
void *thread_execution(void *num);

int compteur = 0;
int k, n;
pthread_t *thread_array;
int *array;

int main (int argc, char* argv[]) {

        if (argc < 3) {
        printf("Veuillez fournir deux valeurs en argument.\n");
        return 1;
    }

    k = atoi(argv[1]);
    n = atoi(argv[2]);

    array = malloc((n+1)*sizeof(int));
    thread_array = malloc((k)*sizeof(pthread_t));

        // Initialisation du tableau booleen (1 pour vrai, 0 pour faux)
        for (int i = 2; i<n; i++) {
                array[i] = 1;
        }
        // Creation des threads

        creat_thread(k);

        for (int w = 0; w < k; w++) {
                pthread_join(thread_array[w], NULL);
        }

        //printf("Les nombres premiers nombres presents avant %d sont : \n", n
```

```
    );
    for (int l = 2; l < n; l++) {
        if (array[l] == 1) {
            // Verification des valeurs des nombres premiers
            //printf("%d\n", l);
            compteur++;
        }
    }

    // Verification du nombre de nombres premiers
    //printf("\n Il y a %d nombres premiers", compteur);

    return 0;

}
```

4 Tache 4

Nous avons pendant cette tâche pu "benchmarker" notre programme thread selon le nombree de threads utilisés, puis le comparer à l'algorithme séquentiel.

Nous avons réalisé ce test via un script bash, celui-ci fait tourner 7 fois (une fois par thread) le programme threadé avec 10 itérations, et fait la moyenne du temps de calcul de ces 10 itérations pour chaque thread. Nous avons comparé ces valeurs avec une moyenne de 10 itérations de l'algorithme séquentiel puis avons comparé les résultats dans le graphe ci-dessous.



A notre grande surprise, le séquentiel reste le plus rapide, et plus le nombre de thread augmente, plus le temps de calcul est long. Nous nous attendions à tout le contraire, pensant que le nombre de threads allait grandement améliorer la recherche dans le tableau.

Nos pistes de réponses seraient que la création de threads prend un certain temps ce qui ralentirait le programme, surtout dans un programme aussi court que celui-ci. De plus, la notion de thread semble être énormément liée au hardware, ayant fait les tests sur une machine virtuelle possédant 4 coeurs virtuels, les résultats ne seraient vraiment pas les mêmes sur un ordinateur avec plus de coeurs et plus puissant. Après avoir échangé avec d'autres étudiants ayant des ordinateurs plus puissants, nous avons remarqué que le temps d'exécution peut être variable mais que pour eux aussi, le séquentiel reste le plus rapide.

5 Tache 5

5.1 Accélération de la boucle

Afin d'accélérer le travail, nous avons fait en sorte que la boucle dans notre calcul d'Erastosthène s'incrémente de 2 en 2 en partant de i = 3. En effet, comme tous les nombres pairs ne sont pas des nombres premiers, par définition, il ne sert à rien d'itérer sur ceux-ci. En plus de cela, lors du calcul avec i = 2, ceux-ci sont déjà tous mis à 0. Nous avons donc dû prendre en compte le cas particulier du 2 qui est le seul nombre premier pair. Ainsi, nous avons effectué 1'itération sur le 2 puis $\sqrt{n}$ sur tous les impairs à partir de 3 (en itérant de 2 en 2 à partir de 3).

Ainsi, nous obtenons la fonction thread_execution suivante :

```
void *thread_execution(void *num) {
        int lower_bound;
        int upper_bound;
        int value = *(int *)num;

        /* cas particulier du 2 */
        int nb_elem = ceil((n-(2*2))/2);

        lower_bound = ceil(2 * 2 + (nb_elem * value / k) * 2);
        upper_bound = ceil(2 * 2 + (nb_elem * (value + 1) / k) * 2);
        printf("[%d ; %d]\n", lower_bound, upper_bound);

        for (int h = lower_bound; h <= upper_bound; h += 2) {
                array[h] = 0;
        }

        /* boucle avec un pas de 2i */
        for (int j = 3; j < sqrt(n); j+=2) {

                //Calcul slices
                int nb_elem = ceil((n-(j*j))/j);

                lower_bound = ceil(j * j + (nb_elem * value / k) * j);
                upper_bound = ceil(j * j + (nb_elem * (value + 1) / k) * j);

                for (int h = lower_bound; h <= upper_bound; h += j) {
                        array[h] = 0;
                }
        }

        pthread_exit(NULL);

}
```

5.2 Optimisation mémoire

Dans un deuxième temps, nous avons cherché à optimiser la place, c'est à dire la taille que prends le tableau contenant les booléens. En effet, celui-ci peut très vite devenir grand, par exemple lorsque nous atteignons 5 millions de valeurs. Dans la même optique que pour l'optimisation de boucle, nous savons que tous les nombres pairs ne sont pas premiers. Il est donc inutile de les mettre dans le tableau.

Pour cette partie, nous ne sommes fixé comme objectif que d'améliorer l'occupation de place du programme, donc nous sommes partis de la base de la tache 3. Nous avons tout d'abord initialisé un tableau à (n+1)/2 en taille. En effet, comme les multiples de 2 ne sont plus là, on peut diviser la taille de notre tableau par 2. Le +1 est nécessaire pour prendre en compte les 3 premières valeurs (cas particulier du 2, 3, 5).

Ensuite, dans la boucle du calcul d'Erastosthène, nous vérifions si la valeur actuelle est un multiple de 2, si oui on ne fait rien (on sait que ce ne sera pas premier). Si ce n'est pas un mutliple de 2, on met à 0 l'indice i/2-1 de notre tableau (pour prendre en compte la diminution de taille de notre tableau.

Nous avons enfin initialisé les 3 premières cases de notre tableau à 1 manuellement (3, 5 et 7), car elles ne sont pas prises en compte par la nouvelle boucle.

Ainsi, nous avons la fonction thread_execution suivante :

```
void *thread_execution(void *num) {
        int lower_bound;
        int upper_bound;
        int value = *(int *)num;

        /* boucle avec un pas de 2i */
        for (int j = 2; j < sqrt(n); j++) {

                //Calcul slices
                int nb_elem = ceil((n-(j*j))/j);

                lower_bound = ceil(j * j + (nb_elem * value / k) * j);
                upper_bound = ceil(j * j + (nb_elem * (value + 1) / k) * j);


                for (int h = lower_bound; h <= upper_bound; h += j ) {
                        if (h%2 != 0){
                                int e;
                                e = (h/2) - 1;

                                array[e] = 0;

                        }

                }
        }

        pthread_exit(NULL);
}
```

Et le main suivant :

```
int main (int argc, char* argv[]) {

        [...]

        // Initialisation du tableau booleen (1 pour vrai, 0 pour faux)
        array[0] = 1; //3
        array[1] = 1; //5
        array[2] = 1; //7
        for (int i = 3; i<(n+1)/2; i++) {
                array[i] = 1;
        }

        // Creation des threads
        [...]

        printf("Les nombres premiers avant %d sont : \n", n);
        printf("2\n");
        for (int l = 0; l < ((n+1)/2)-1; l++) {
                if (array[l] == 1) {
                        // Verification des valeurs des nombres premiers
                        printf("%d\n", 2*l+3);
                        compteur++;
                }
        }

    [...]

}
```