SR02 : TD 8 (Threads : Crible d'Eratosthenes)

Tache 1

Dérouler l'exécution de cette algorithme avec n=20

1.

A) Initialisation du tableau
A = [Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai, Vrai]

B) Boucle for sur i
Parcour des valeurs de i de 2 à $\sqrt{20}$, soit i = 2, 3, 4.

i = 2:

```
A[2] est vrai,
On met à faux toutes les valeurs de j = 4, 6, 8, 10, 12, 14, 16, 18, 20
dans le tableau A.
Le tableau A devient : [Vrai, Vrai, Faux, Vrai, Faux, Vrai, Faux,
Vrai, Faux, Vrai, Faux, Vrai, Faux, Vrai, Faux, Vrai, Faux, Vrai, Faux]
```

i = 3:

```
A[3] est vrai,
On met à faux toutes les valeurs de j = 9, 12, 15, 18 dans le tableau A.
Le tableau A devient : [Vrai, Vrai, Faux, Vrai, Faux, Vrai, Faux, Faux,
Faux, Vrai, Faux, Vrai, Faux, Faux, Faux, Vrai, Faux, Vrai, Faux]
```

i = 4:

```
A[4] est faux, on passe à l'itération suivante.
```

C) Résultat final
Après avoir parcouru toutes les valeurs de i, le tableau A contient les informations sur les nombres premiers jusqu'à 20. Les indices pour lesquels A[i] est vrai correspondent aux nombres premiers.

Le tableau A final est le suivant : [Vrai, Vrai, Faux, Vrai, Faux, Vrai, Faux, Faux, Faux, Vrai, Faux, Vrai, Faux, Faux, Faux, Vrai, Faux, Vrai, Faux]

Donc, les nombres premiers jusqu'à 20 sont : 2, 3, 5, 7, 11, 13, 17, 19.

2. La boucle interne commence à i^2 au lieu de 0 ou i pour des raisons d'efficacité.

L'algorithme utilise le crible d'Ératosthène pour trouver les nombres premiers jusqu'à n. L'idée du crible d'Ératosthène est que tous les multiples d'un nombre ne sont pas premiers, ils peuvent donc être supprimés. Ainsi, au lieu de parcourir tous les multiples de i, la boucle interne commence à i^2 pour éliminer tous les multiples précédents qui ont déjà été marqués comme non premiers par des valeurs de i plus petites.

Supposons que i soit un nombre premier. Si nous commençons à i ou à une valeur inférieure à i^2, on passe sur des multiples de nombres premiers précédents, ce qui serait redondant. Par exemple, si i = 3, en commencant à $3^2 = 9$ on ne passe pas par 6 (multiple de 2) et 3 (multiple de 3), car ils ont déjà été testés précédemment.

En commençant à i^2, on est sur que chaque multiple antérieur a déjà été marqué comme non premier, ce qui permet de réduire le nombre d'itérations. Ce départ à i^2 constitue donc principalement une optimisation de l'algorithme puisque les itérations avant i^2 ne seraient pas nécéssaires.

3. La première boucle s'exécute jusqu'à $\sqrt{n}$ pour des raisons d'efficacité. L'objectif de cette boucle est de marquer tous les multiples des nombres premiers jusqu'à $\sqrt{n}$ comme non premiers.

Si $\sqrt{n}$ n'est pas un entier, nous prenons la partie entière inférieur de $\sqrt{n}$.

Dans le cas où $\sqrt{n}$ est un entier, la première boucle s'exécute normalement de 2 à $\sqrt{n}$ inclus, couvrant ainsi tous les multiples des nombres premiers jusqu'à $\sqrt{n}$.

L'objectif est de limiter le nombre d'itérations nécessaires, car après $\sqrt{n}$, nous n'avons plus besoin de vérifier les multiples des nombres premiers plus grands que $\sqrt{n}$, car ils ont déjà été traités lors des itérations précédentes.(Cette propriété est une propriété toujours vrai pour les nombres premiers.)

Lorsque nous cherchons tous les nombres premiers inférieurs ou égale à N, nous pouvons effectivement itérer sur les entiers i de 2 à N.

Cependant, il n'est pas nécessaire de vérifier tous les nombres jusqu'à N. Il suffit de vérifier jusqu'à la racine carrée de N. Cela est dû au fait que si un nombre non premier a un facteur qui est plus grand que sa racine carrée, il aurait déjà été multiplié par un autre facteur plus petit qui est inférieur à sa racine carrée.

Par conséquent, en vérifiant seulement jusqu'à la racine carrée de N, nous couvrons tous les cas et nous évitons les tests redondants.

"D'après le théorème des diviseurs premiers, si n n'est divisible par aucun des nombres premiers inférieur ou égaux à sa racine carrée, on peut affirmer qu'il est premier."

Cela permet d'optimiser l'algorithme et d'améliorer son efficacité.

Tache 2

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

void Eratosthenes(unsigned long n, unsigned long *tab) {
    for (unsigned long i = 2; i <= n; i++) {
        tab[i] = 1;
    }

    unsigned long racine = $\sqrt{n}$;
    for (unsigned long i = 2; i <= racine; i++) {
        if (tab[i] == 1) {
            for (unsigned long j = i * i; j <= n; j += i) {
                tab[j] = 0;
            }
        }
    }
}

void PrintTab(unsigned long n, unsigned long *tab) {
    for (unsigned long i = 2; i <= n; i++) {
        if (tab[i] == 1) {
            printf("%lu ", i);
        }
    }
    printf("\n");
}

int main(int argc, char **argv) {
    if(argc != 2) {
        printf("Usage: %s <n>\n", argv[0]);
        return 1;
    }
    unsigned long n = atol(argv[1]);

    if (n > 1) {
        unsigned long *tab = (unsigned long *)malloc((n+1) * sizeof(unsigned long));
        clock_t start, end;
        double temps_execution;

        start = clock();
        Eratosthenes(n, tab);
        end = clock();
```

```
        temps_execution = ((double) (end - start)) / CLOCKS_PER_SEC;
        printf("%f\n", temps_execution);
        // PrintTab(100, tab);

        free(tab);
    } else {
        printf("n doit être supérieur à 1.\n");
    }

    return 0;
}
```

Tache 3

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#define k 7

typedef struct
{
    unsigned long *tab;
    unsigned long i;
    unsigned long n;
    unsigned long racine;
    pthread_mutex_t *mutex;
} ArgumentThread;

void *fonctionThread(void *argument)
{
    ArgumentThread *arg = (ArgumentThread *)argument;
    unsigned long *tab = arg->tab;
    unsigned long i = arg->i;
    unsigned long n = arg->n;
    for (unsigned long t = i; t <= arg->racine; t += k)
    {
        if (tab[t] == 1)
        {
            pthread_mutex_lock(arg->mutex);
            for (unsigned long j = t * t; j <= n; j += t)
            {
                tab[j] = 0;
            }
            pthread_mutex_unlock(arg->mutex);
        }
    }
    pthread_exit(NULL);
}

void Eratosthenes(unsigned long n, unsigned long *tab)
{
    pthread_t *thread = malloc(sizeof(pthread_t) * k);
    unsigned long indexThread = 0;
    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex, NULL);

    for (int cpt = 0; cpt < k; cpt++)
```

```
    {
        ArgumentThread *arg = malloc(sizeof(ArgumentThread));
        arg->tab = tab;
        arg->i = 2 + cpt;
        arg->n = n;
        arg->racine = $\sqrt{n}$;
        arg->mutex = &mutex;

        pthread_create(&thread[indexThread], NULL, fonctionThread, arg);
        indexThread++;
    }
    for (unsigned long i = 0; i < indexThread; i++)
    {
        pthread_join(thread[i], NULL);
    }
    free(thread);
    pthread_mutex_destroy(&mutex);
}

void PrintTab(unsigned long n, unsigned long *tab)
{
    // Affichage des nombres premiers
    for (unsigned long i = 2; i <= n; i++)
    {
        printf("%ld ", tab[i]);
    }
    printf("\n");
}

void PrintIndex(unsigned long n, unsigned long *tab)
{
    int cpt = 0;
    // Affichage des nombres premiers
    for (unsigned long i = 2; i <= n; i++)
    {
        if (tab[i] == 1)
        {
            cpt++;
            // printf("%ld ", i);
        }
    }
    // printf("%d\n", cpt);
    // printf("\n");
}

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

```c
{
    if (argc != 2)
    {
        printf("Usage: %s n\n", argv[0]);
        return 0;
    }
    unsigned long n = atol(argv[1]);

    if (n > 1)
    {
        unsigned long *tab = (unsigned long *)malloc((n + 1) * sizeof(unsigned long));
        clock_t start, end;
        double temps_execution;
        // Initialisation du tableau
        for (unsigned long i = 2; i <= n; i++)
        {
            tab[i] = 1;
        }
        start = clock();
        Eratosthenes(n, tab);
        end = clock();
        temps_execution = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("%f\n", temps_execution);

        PrintIndex(n, tab);
        // PrintTab(n, tab);
        free(tab);
    }
    else
    {
        printf("n doit être supérieur à 1.\n");
    }

    return 0;
}
```

Tache 4: Interprétation des résultats

La similarité des points pour 500 000 et 1 000 000 indique que les temps d'exécution sont assez uniformes, ce qui suggère que les différentes implémentations ne génèrent pas de performances nettement meilleures.

À partir de la valeur 2 000 000, une dispersion des valeurs devient apparente, ce qui met en évidence les différences d'implémentation de manière plus marquée. On peut observer deux groupes principaux de valeurs. Le premier groupe se situe autour de 0,030 secondes, tandis que le second groupe se situe autour de 0,036 secondes.

Le premier groupe est constitué des implémentations utilisant des threads pour les valeurs de k égales à 1, 2, 3, 4 et 6. Le second groupe comprend l'implémentation séquentielle ainsi que celle utilisant des threads pour les valeurs de k égales à 5 et 7.

En utilisant la valeur 4 000 000, ces résultats précédents sont confirmés, et la valeur optimale de k (k=3) est obtenue. On observe que pour les valeurs de k égales à 1, 2, 4 et 6, les temps d'exécution restent assez proches. Ainsi, les différences de nombre de threads n'ont pas d'impact significatif sur le temps d'exécution des programmes.

Intervalle de confiance

Afin de calculer l'intervalle de confiance sur ces valeurs, nous pouvons poser X la variable aléatoire repésentant le temps d'exécution du programme. Ensuite, nous devons calculer la moyenne de x ($\overline{x}$) ainsi que son équart type (σ).
Nous pouvons poser un niveau de confiance α^*.
Dans notre cas, on peut prendre un niveau de confiance $\alpha^* = 95\%$. Si

$$X \sim \mathcal{N}(\mu, \sigma^2)$$

On pose :
Le quantile d'une loi de normale centrée réduite d'ordre α^* : u_{α^*}.
Le nombre de valeurs : n.
L'intervalle de confiance est donné par la formule :

$$IC = [\overline{x} - u_{1-\frac{\alpha^*}{2}} \frac{\sigma}{\sqrt{n}}, \overline{x} + u_{1-\frac{\alpha^*}{2}} \frac{\sigma}{\sqrt{n}}] \tag{1}$$

Si la valeur observée se trouve dans l'intervalle de confiance, elle est considérée comme correct, sinon, on considère qu'il y a une erreur.

Tache 5

Accélération de la boucle interne

Les deux cas distincts à traiter sont les suivants :

Cas spécial pour le nombre 2 : Le nombre 2 est le seul nombre premier pair. Comme tous les autres nombres pairs sont exclus d'être premiers, nous devons traiter le cas spécial du nombre 2 séparément. Par conséquent, nous devons effectuer la boucle interne avec un pas de 2 au lieu de i lorsque i est égal à 2.

Boucle interne pour les nombres impairs : Tous les autres nombres premiers sont impairs. Par conséquent, nous pouvons ignorer les nombres pairs lors de la boucle interne pour améliorer l'efficacité de l'algorithme. Nous effectuons donc la boucle interne avec un pas de 2i pour tous les nombres premiers impairs, en commençant par i^2.

En résumé, le premier cas distinct est celui du nombre 2 où nous utilisons un pas de 2 dans la boucle interne, tandis que le deuxième cas distinct est celui des autres nombres premiers impairs, pour lesquels nous utilisons également un pas de 2i dans la boucle interne.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#define k 7

typedef struct
{
    unsigned long *tab;
    unsigned long i;
    unsigned long n;
    unsigned long racine;
    pthread_mutex_t *mutex;
} ArgumentThread;

void *fonctionThread(void *argument)
{
    ArgumentThread *arg = (ArgumentThread *)argument;
    unsigned long *tab = arg->tab;
    unsigned long i = arg->i;
    unsigned long n = arg->n;
    for (unsigned long t = i; t <= arg->racine; t += k)
    {
        if (tab[t] == 1)
        {
            pthread_mutex_lock(arg->mutex);
```

```
            if(t == 2)
            {
                for (unsigned long j = t * t; j <= n; j += t)
                {
                    tab[j] = 0;
                }
            }
            else
            {
                for (unsigned long j = t * t; j <= n; j += 2*t)
                {
                    tab[j] = 0;
                }
            }
            pthread_mutex_unlock(arg->mutex);
        }
    }
    pthread_exit(NULL);
}

void Eratosthenes(unsigned long n, unsigned long *tab)
{
    pthread_t *thread = malloc(sizeof(pthread_t) * k);
    unsigned long indexThread = 0;
    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex, NULL);

    for (int cpt = 0; cpt < k; cpt++)
    {
        ArgumentThread *arg = malloc(sizeof(ArgumentThread));
        arg->tab = tab;
        arg->i = 2 + cpt;
        arg->n = n;
        arg->racine = $\sqrt{n}$;
        arg->mutex = &mutex;

        pthread_create(&thread[indexThread], NULL, fonctionThread, arg);
        indexThread++;
    }
    for (unsigned long i = 0; i < indexThread; i++)
    {
        pthread_join(thread[i], NULL);
    }
    free(thread);
    pthread_mutex_destroy(&mutex);
}
```

```c
void PrintTab(unsigned long n, unsigned long *tab)
{
    // Affichage des nombres premiers
    for (unsigned long i = 2; i <= n; i++)
    {
        printf("%ld ", tab[i]);
    }
    printf("\n");
}

void PrintIndex(unsigned long n, unsigned long *tab)
{
    int cpt = 0;
    // Affichage des nombres premiers
    for (unsigned long i = 2; i <= n; i++)
    {
        if (tab[i] == 1)
        {
            cpt++;
            // printf("%ld ", i);
        }
    }
    // printf("%d\n", cpt);
    // printf("\n");
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        printf("Usage: %s n\n", argv[0]);
        return 0;
    }
    unsigned long n = atol(argv[1]);

    if (n > 1)
    {
        unsigned long *tab = (unsigned long *)malloc((n + 1) * sizeof(unsigned long));
        clock_t start, end;
        double temps_execution;
        // Initialisation du tableau
        for (unsigned long i = 2; i <= n; i++)
        {
            tab[i] = 1;
        }
```

```
        start = clock();
        Eratosthenes(n, tab);
        end = clock();
        temps_execution = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("%f\n", temps_execution);

        PrintIndex(n, tab);
        // PrintTab(n, tab);
        free(tab);
    }
    else
    {
        printf("n doit être supérieur à 1.\n");
    }

    return 0;
}
```

Réduction de l'espace mémoire

Pour réduire l'espace mémoire utilisé par l'algorithme, nous pouvons utiliser un tableau plus petit et ne stocker que les booléens pour les nombres impairs supérieurs à 3. Voici comment effectuer cette modification :

Initialisez un tableau isPrime de valeurs booléennes indexées de 0 à $\frac{n-2}{2}$, toutes initialisées à vrai. Le tableau isPrime sera moitié moins grand que le tableau A précédemment utilisé.
Pour i = 0, 1, 2, ..., $\frac{\sqrt{n}-3}{2}$:
Si isPrime[i] est vrai, cela signifie que le nombre $k = 2i + 3$ est premier.
Effectuez la boucle interne avec un pas de 2k, en commençant à $j = \frac{k^2-3}{2}$ et augmentant j de k à chaque itération ($j+ = k$).
Marquez isPrime[j] comme faux.
Les nombres premiers seront les nombres impairs supérieurs à 2 (i.e., $k = 2i + 3$) pour lesquels isPrime[i] est vrai.

Cette modification permet de réduire l'espace mémoire utilisé par le tableau en ne stockant que les booléens pour les nombres impairs supérieurs à 3. Ainsi, nous évitons de stocker les booléens pour les nombres pairs qui ne peuvent pas être premiers, ce qui réduit la taille du tableau et l'espace mémoire requis.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#define k 1

typedef struct
```

```
{
    unsigned long *tab;
    unsigned long i;
    unsigned long n;
    unsigned long racine;
    pthread_mutex_t *mutex;
} ArgumentThread;


void *fonctionThread(void *argument)
{
    ArgumentThread *arg = (ArgumentThread *)argument;
    unsigned long *tab = arg->tab;
    unsigned long i = arg->i;
    unsigned long n = arg->n;
    for (unsigned long t = i; t <= (arg->racine); t += k)
    {
        if (tab[t] == 1)
        {
            unsigned long l = 2 * t + 3;
            pthread_mutex_lock(arg->mutex);
            for (unsigned long j = (l * l - 3) / 2; j < n / 2 - 1; j += l)
            {
                tab[j] = 0;
            }
            pthread_mutex_unlock(arg->mutex);
        }
    }
    pthread_exit(NULL);
}

void Eratosthenes(unsigned long n, unsigned long *tab)
{
    pthread_t *thread = malloc(sizeof(pthread_t) * k);
    unsigned long indexThread = 0;
    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex, NULL);

    for (int cpt = 0; cpt < k; cpt++)
    {
        ArgumentThread *arg = malloc(sizeof(ArgumentThread));
        arg->tab = tab;
        arg->i = 0 + cpt;
        arg->n = n;
        arg->racine = $\sqrt{n}$;
        arg->mutex = &mutex;
```

```
        pthread_create(&thread[indexThread], NULL, fonctionThread, arg);
        indexThread++;
    }
    for (unsigned int i = 0; i < indexThread; i++)
    {
        pthread_join(thread[i], NULL);
    }
    free(thread);
    pthread_mutex_destroy(&mutex);
}

void PrintTab(unsigned long n, unsigned long *tab)
{
    // Affichage des nombres premiers
    printf("2 ");
    for (unsigned long i = 0; i < (n / 2) -1; i++)
    {
        if (tab[i] == 1)
        {
            unsigned long prime = 2 * i + 3;
            printf("%ld ", prime);
        }
    }
    printf("\n");
}

void PrintIndex(unsigned long n, unsigned long *tab)
{
    int cpt = 1; // Initialisé à 1 pour inclure le nombre premier 2
    for (unsigned long i = 0; i < (n / 2) - 1; i++)
    {
        // printf("%ld ", tab[i]);
        if (tab[i] == 1)
        {
            cpt++;
        }
    }
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        printf("Usage: %s n\n", argv[0]);
        return 0;
```

```
    }
    unsigned long n = atol(argv[1]);

    if (n > 1)
    {
        unsigned long *tab = (unsigned long *)malloc(((n / 2) - 1) * sizeof(unsigned long));
        clock_t start, end;
        double temps_execution;
        // Initialisation du tableau
        for (unsigned long i = 0; i < (n / 2) - 1; i++)
        {
            tab[i] = 1;
        }
        start = clock();
        Eratosthenes(n, tab);
        end = clock();
        temps_execution = ((double)(end - start)) / CLOCKS_PER_SEC;
        printf("%f\n", temps_execution);

        PrintIndex(n, tab);
        // PrintTab(n, tab);
        free(tab);
    }
    else
    {
        printf("n doit être supérieur à 1.\n");
    }

    return 0;
}
```

Interprétation des résultats

Ces deux algorithmes que nous avons utilisés ont permis de générer deux graphiques illustrant les résultats obtenus après l'application de différentes optimisations. En observant ces graphiques, nous constatons, comme précédemment, que les valeurs restent assez similaires pour les entrées de 500 000 et 1 000 000. Cependant, elles deviennent plus intéressantes et exploitables lorsque les entrées atteignent 2 000 000 et 4 000 000.

En ce qui concerne l'optimisation de la boucle interne, nous remarquons que les valeurs de k égales à 2, 4, 5 et 6 sont les plus efficaces, tandis que l'algorithme séquentiel ainsi que les valeurs de k égales à 1, 3 et 7 sont moins performants. Cela suggère que les valeurs de k sélectionnées pour les threads ont une influence significative sur les performances de l'algorithme, et que certaines valeurs sont plus adaptées que d'autres pour tirer pleinement parti du parallélisme.

En ce qui concerne l'optimisation de l'espace mémoire, nous remarquons que la valeur de k égale à 5 se distingue particulièrement des autres valeurs. Cela indique que k = 5 est l'option optimale pour utiliser les threads dans ce cas spécifique. Il est intéressant de noter que, dans les deux cas d'optimisation, la valeur 5 se révèle être la plus efficace pour l'utilisation des threads. De plus, cette valeur permet d'obtenir un temps d'exécution bien meilleur que celui de l'algorithme séquentiel.

Enfin, en combinant les valeurs optimales de k pour chaque cas (5 pour l'optimisation de la boucle interne et 3 pour la version parallèle précédemment mentionnée), nous remarquons qu'avec une entrée de 4 000 000, la version parallèle simple prend environ 0,055 seconde pour s'exécuter, la version optimisée de la boucle interne prend environ 0,035 seconde, et la version optimisée de l'espace mémoire prend environ 0,024 seconde. Ces résultats indiquent que les optimisations permettent de diviser par deux le temps d'exécution du programme. Cette réduction significative du temps d'exécution est d'une importance capitale lorsqu'il s'agit de calculer des nombres premiers de grande taille.

En conclusion, ces optimisations s'avèrent extrêmement efficaces, car elles permettent de gagner un temps précieux et de calculer des nombres plus grands dans un laps de temps fini. Ces améliorations offrent également l'avantage de réduire la consommation de ressources nécessaires aux calculs.