lili

Que doit faire le programme ?

- On a **deux grands tableaux de nombres**, appelés `A` et `B`, avec **100 millions** de valeurs.
- On doit comparer **chaque élément** des deux tableaux et mettre le **plus petit** dans un troisième tableau `C`.
- Pour aller plus vite, on va utiliser **plusieurs threads** (des petits programmes qui travaillent en parallèle).

Comment répartir le travail entre les threads ?

Il y a **trois méthodes** pour distribuer le travail :

1. **Répartition cyclique** : chaque thread prend un élément sur `A` et `B`, puis le suivant est pris par un autre thread, et ainsi de suite.
2. **Répartition par blocs** : chaque thread traite un bloc de plusieurs éléments (ex: 2048 à la fois).
3. **Farming (à la demande)** : un thread prend un bloc de travail, et dès qu'il a fini, il en demande un autre.

Les contraintes

- On peut utiliser **de 1 à 1024 threads**.
- On doit mesurer le **temps de calcul moyen sur 10 essais**.
- On peut tester avec ou sans **migration des threads** entre les processeurs.

Travail demandé

1. **Coder le programme en C** qui prend en entrée le **nombre de threads**, la **méthode de répartition**, et si la **migration** est autorisée ou non.
2. **Écrire un script shell** qui lance automatiquement le programme avec **plusieurs valeurs de paramètres**.
3. Modifier le programme pour **afficher les résultats en format CSV** (fichier tableur) et faire des **graphiques**.

4. **Analyser les résultats** : quelle configuration est **la plus rapide** sur la machine utilisée ?

À rendre :

- Une **explication** du programme en français.
- Le **code source en C**.
- Le **script shell**.
- Les **résultats et graphiques** de performance.
- L'**analyse** des résultats.

Explication détaillée du programme C et du script shell

Programme C (min_array.c)

Inclusions et définitions

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <string.h>
#include <errno.h>

#ifdef __linux__
#include <sched.h>
#endif
```

Ces lignes incluent les bibliothèques nécessaires. `pthread.h` est pour la programmation multithread, `time.h` pour mesurer le temps, et `sched.h` (sur Linux) pour l'affinité des threads.

```
#define ARRAY_SIZE 100000000
#define BLOCK_SIZE 2048
#define NB_MEASURE 10
```

Ces constantes définissent:

- `ARRAY_SIZE` : la taille des tableaux (100 millions d'éléments)
- `BLOCK_SIZE` : la taille de chaque bloc pour les méthodes par blocs
- `NB_MEASURE` : le nombre de mesures à effectuer pour obtenir une moyenne

Variables globales

```
double *A, *B, *C;
int nb_threads;
int migration_allowed;
char method[16];
```

- `A` , `B` , `C` : les trois tableaux (A et B pour les entrées, C pour les résultats)
- `nb_threads` : nombre de threads à utiliser
- `migration_allowed` : indique si les threads peuvent migrer entre les cœurs
- `method` : la méthode de parallélisation choisie

```
int current_block = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int *blocks_processed;
int total_blocks;
```

Variables pour la méthode "farming":

- `current_block` : compteur partagé du bloc actuel
- `mutex` : verrou pour protéger l'accès à `current_block`
- `blocks_processed` : tableau pour compter combien de blocs chaque thread a traité
- `total_blocks` : nombre total de blocs à traiter

```
typedef struct {
    int id;
} thread_arg_t;
```

Structure pour passer l'identifiant du thread comme argument.

Fonctions utilitaires

```
double get_time_in_seconds() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec + ts.tv_nsec / 1e9;
}
```

Cette fonction mesure le temps avec précision en secondes.

```
void set_thread_affinity(int thread_id) {
#ifdef __linux__
    if (!migration_allowed) {
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);
        CPU_SET(thread_id, &cpuset);
        int s = pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuse
t);
        if (s != 0) {
            fprintf(stderr, "Erreur setting affinity pour thread %d: %s\n", thread_i
d, strerror(errno));
        }
    }
#endif
}
```

Cette fonction fixe l'affinité d'un thread à un cœur CPU spécifique si `migration_allowed` est 0.

Méthodes de parallélisation

1. Méthode cyclique par élément

```
void *thread_cyclic(void *arg) {
    thread_arg_t *targ = (thread_arg_t*)arg;
    int id = targ->id;
    set_thread_affinity(id);
    for (long i = id; i < ARRAY_SIZE; i += nb_threads) {
        C[i] = (A[i] < B[i]) ? A[i] : B[i];
```

```
    }
    return NULL;
}
```

Chaque thread traite les éléments de manière cyclique: le thread 0 prend les éléments 0, n, 2n, ..., le thread 1 prend les éléments 1, n+1, 2n+1, etc.

2. Méthode par blocs

```
void *thread_block(void *arg) {
    thread_arg_t *targ = (thread_arg_t*)arg;
    int id = targ->id;
    set_thread_affinity(id);
    int nb_blocks = (ARRAY_SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE;
    for (int blk = id; blk < nb_blocks; blk += nb_threads) {
        int start = blk * BLOCK_SIZE;
        int end = start + BLOCK_SIZE;
        if (end > ARRAY_SIZE)
            end = ARRAY_SIZE;
        for (int i = start; i < end; i++) {
            C[i] = (A[i] < B[i]) ? A[i] : B[i];
        }
    }
    return NULL;
}
```

Chaque thread prend des blocs entiers (2048 éléments consécutifs) de manière cyclique.

3. Méthode farming (distribution dynamique)

```
void *thread_farming(void *arg) {
    thread_arg_t *targ = (thread_arg_t*)arg;
    int id = targ->id;
    set_thread_affinity(id);
    int local_count = 0;
    while (1) {
        int blk;
        pthread_mutex_lock(&mutex);
```

```
        blk = current_block;
        current_block++;
        pthread_mutex_unlock(&mutex);
        if (blk * BLOCK_SIZE >= ARRAY_SIZE)
            break;
        int start = blk * BLOCK_SIZE;
        int end = start + BLOCK_SIZE;
        if (end > ARRAY_SIZE)
            end = ARRAY_SIZE;
        for (int i = start; i < end; i++) {
            C[i] = (A[i] < B[i]) ? A[i] : B[i];
        }
        local_count++;
    }
    blocks_processed[id] = local_count;
    return NULL;
}
```

Les threads demandent dynamiquement des blocs à traiter grâce à un compteur partagé protégé par mutex.

Initialisation et libération des tableaux

```
void init_arrays() {
    A = (double*)malloc(sizeof(double) * ARRAY_SIZE);
    B = (double*)malloc(sizeof(double) * ARRAY_SIZE);
    C = (double*)malloc(sizeof(double) * ARRAY_SIZE);
    if (!A || !B || !C) {
        fprintf(stderr, "Erreur d'allocation mémoire.\n");
        exit(EXIT_FAILURE);
    }
    // Remplissage des tableaux (exemple : A[i] = i, B[i] = ARRAY_SIZE - i)
    for (long i = 0; i < ARRAY_SIZE; i++) {
        A[i] = (double)i;
        B[i] = (double)(ARRAY_SIZE - i);
    }
}
```

```
void free_arrays() {
    free(A);
    free(B);
    free(C);
}
```

Ces fonctions allouent la mémoire pour les tableaux, les initialisent et les libèrent à la fin.

Fonction principale

```
int main(int argc, char *argv[]) {
    if (argc < 4) {
        fprintf(stderr, "Usage: %s <method: cyclic|block|farming> <nb_thread
s> <migration: 0|1>\n", argv[0]);
        return EXIT_FAILURE;
    }
    strncpy(method, argv[1], sizeof(method) - 1);
    nb_threads = atoi(argv[2]);
    migration_allowed = atoi(argv[3]);
    if (nb_threads < 1) {
        fprintf(stderr, "Le nombre de threads doit être >= 1.\n");
        return EXIT_FAILURE;
    }
```

Récupère et vérifie les arguments de la ligne de commande.

```
    init_arrays();

    if (strcmp(method, "farming") == 0) {
        blocks_processed = (int*)calloc(nb_threads, sizeof(int));
        total_blocks = (ARRAY_SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE;
    }
```

Initialise les tableaux et, pour la méthode farming, alloue de la mémoire pour compter les blocs traités.

```
    double total_time = 0.0;
    for (int measure = 0; measure < NB_MEASURE; measure++) {
        if (strcmp(method, "farming") == 0) {
            current_block = 0;
            memset(blocks_processed, 0, nb_threads * sizeof(int));
        }
```

Pour chaque mesure, réinitialise les compteurs pour la méthode farming.

```
        pthread_t *threads = (pthread_t*)malloc(nb_threads * sizeof(pthread_
t));
        thread_arg_t *targs = (thread_arg_t*)malloc(nb_threads * sizeof(thread
_arg_t));
        double start_time = get_time_in_seconds();
        for (int i = 0; i < nb_threads; i++) {
            targs[i].id = i;
            int ret;
            if (strcmp(method, "cyclic") == 0) {
                ret = pthread_create(&threads[i], NULL, thread_cyclic, &targs[i]);
            } else if (strcmp(method, "block") == 0) {
                ret = pthread_create(&threads[i], NULL, thread_block, &targs[i]);
            } else if (strcmp(method, "farming") == 0) {
                ret = pthread_create(&threads[i], NULL, thread_farming, &targs
[i]);
            } else {
                fprintf(stderr, "Méthode inconnue: %s\n", method);
                exit(EXIT_FAILURE);
            }
            if (ret != 0) {
                fprintf(stderr, "Erreur lors de la création du thread %d\n", i);
                exit(EXIT_FAILURE);
            }
        }
```

Crée les threads selon la méthode choisie et démarre le chronomètre.

```
        for (int i = 0; i < nb_threads; i++) {
            pthread_join(threads[i], NULL);
```

```
        }
        double end_time = get_time_in_seconds();
        total_time += (end_time - start_time);
        free(threads);
        free(targs);
    }
    double average_time = total_time / NB_MEASURE;
```

Attend que tous les threads terminent, arrête le chronomètre et calcule le temps moyen.

```
    int min_blocks = -1, max_blocks = -1;
    if (strcmp(method, "farming") == 0) {
        for (int i = 0; i < nb_threads; i++) {
            int count = blocks_processed[i];
            if (min_blocks == -1 || count < min_blocks)
                min_blocks = count;
            if (max_blocks == -1 || count > max_blocks)
                max_blocks = count;
        }
    }
```

Pour la méthode farming, calcule le nombre minimal et maximal de blocs traités par thread.

```
    // Affichage des résultats en format CSV
    if (strcmp(method, "farming") == 0) {
        printf("%s,%d,%d,%.6f,%d,%d\n", method, nb_threads, migration_allo
wed, average_time, min_blocks, max_blocks);
    } else {
        printf("%s,%d,%d,%.6f\n", method, nb_threads, migration_allowed, av
erage_time);
    }

    if (strcmp(method, "farming") == 0) {
        free(blocks_processed);
    }
    free_arrays();
```

```
    return 0;
}
```

Affiche les résultats au format CSV et libère la mémoire.

Script shell (run_tests.sh)

```
#!/bin/bash
# run_tests.sh
# Ce script lance le programme avec différentes configurations.
```

Shebang et description du script.

```
# Nom du binaire compilé (assurez-vous de compiler avec: gcc -pthread -
O2 min_array.c -o min_array)
BINARY=./min_array
OUTPUT=results.csv
```

Définit le nom du programme compilé et le fichier de sortie.

```
# En-tête du fichier CSV
echo "method,nb_threads,migration,average_time,min_blocks,max_blocks"
> $OUTPUT
```

Crée le fichier CSV et écrit l'en-tête.

```
# Méthodes à tester
methods=("cyclic" "block" "farming")
# Nombre de threads à tester
threads=(1 2 4 8 16 32 64 128 256 512 1024)
# Migration : 0 = non autorisée, 1 = autorisée
migrations=(0 1)
```

Définit les différentes configurations à tester:

- 3 méthodes différentes
- 11 nombres de threads différents
- 2 modes de migration (avec/sans)

```
for m in "${methods[@]}"; do
  for mig in "${migrations[@]}"; do
    for t in "${threads[@]}"; do
      echo "Running method=$m, threads=$t, migration=$mig"
      # Exécution du programme et récupération de la sortie CSV
      result=$($BINARY $m $t $mig)
```

Boucles imbriquées pour tester toutes les combinaisons possibles.

```
      # Pour les méthodes cyclic et block qui ne fournissent pas min_blocks e
t max_blocks, ajouter des champs vides
      if [[ "$m" == "cyclic" || "$m" == "block" ]]; then
        echo "$result,," >> $OUTPUT
      else
        echo "$result" >> $OUTPUT
      fi
    done
  done
done
```

Pour les méthodes "cyclic" et "block", ajoute deux champs vides pour min_blocks et max_blocks afin de maintenir le format CSV cohérent.

```
echo "Les résultats ont été sauvegardés dans $OUTPUT"
```

Affiche un message de confirmation à la fin de l'exécution.