Programmation parallèle et distribuée

Travaux dirigés

Marek Felšöci

2025/2026

1. Introduction

1.1. Documents associés

Spécifications d'OpenMP 6.0	https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf

1.2. Pré-requis

Les exercices portant sur la parallélisation en mémoire partagée avec OpenMP (voir Section 2) nécessitent un ordinateur équipé d'un **processeur multi-cœur** ainsi qu'un **compilateur** du langage C **avec le support d'OpenMP** tel que le GNU C Compiler (gcc) ou Clang.

1.3. Configuration d'OpenMP

Les instructions dans cette section permettent de vérifier la capacité d'un ordinateur et de sa configuration logicielle à compiler et à exécuter en parallèle des programmes écrits à l'aide d'OpenMP.

Placez le code source suivant dans un fichier nommé `openmp-test.c`,

```c
#include <stdio.h>
#include <omp.h>

int main(void) {
  printf(
    "Nombre de fils d'exécution disponibles : %d\n",
    omp_get_max_threads()
  );

  #pragma omp parallel
  printf("Bonjour le monde !\n");

  return 0;
}
```

compilez le fichier avec le compilateur compatible de votre choix tel que le GNU C Compiler (gcc)

```
gcc -o openmp-test openmp-test.c -fopenmp
```

et exécutez-le afin de vérifier la capacité de votre ordinateur à exécuter des programmes à base d'OpenMP.

```
OMP_NUM_THREADS=4 ./openmp-test
```

Vous devriez obtenir la sortie suivante.

```
Nombre de fils d'exécution disponibles : 4
Bonjour le monde !
Bonjour le monde !
Bonjour le monde !
Bonjour le monde !
```

2. Directives OpenMP

2.1. Construction de régions parallèles

2.1.1. Exercice 1

Prenons le code source ci-dessous. Il s'agit d'un programme plus simple que celui que nous avons utilisé pour vérifier notre configuration logicielle. Il ne comporte qu'une seule directive OpenMP et ne fait pas d'appel aux fonctions de la bibliothèque de cette interface de programmation.

```c
#include <stdio.h>

int main(void) {
  #pragma omp parallel
  printf("Bonjour\n");
  printf("le monde !\n");
  return 0;
}
```

Compilez ce code source sans et avec l'option `-fopenmp` du compilateur et exécutez le programme résultant dans les deux cas. Qu'observez-vous ?

Combien de fils d'exécution le programme utilise-t-il par défaut ?

En vous appuyant sur le support du cours, changez le nombre de fils d'exécution utilisé par défaut à l'aide de :

- la clause `num_threads`,
- la fonction `omp_set_num_threads` de la bibliothèque OpenMP,
- la variable d'environnement `OMP_NUM_THREADS`.

Quels sont les avantages et les inconvénients de ces différentes méthodes ?

2.1.2. Exercice 2

Dans cet exercice, nous reprenons le code source de la Section 2.1.1 en modifiant la signature de la fonction principale afin de pouvoir accéder aux éventuels arguments passés au programme depuis la ligne de commandes. Cette implémentation nous permet de vérifier si au moins un argument a été passé au programme sur la ligne de commandes et si cet argument est un « 0 » (zéro).

```c
#include <stdio.h>

int main(int argc, char ** argv) {
  // Testons si au moins un argument a été passé au programme sur la ligne de
  // commandes et, si c'est le cas, vérifions si c'est « 0 ».
  if(argc > 1 && argv[1][0] == '0') {

  }

  #pragma omp parallel
  printf("Bonjour\n");
  printf("le monde !\n");

  return 0;
}
```

Le but de l'exercice est d'adapter le code source de façon à empêcher la parallélisation du programme si le premier argument de la ligne de commande est « 0 » (zéro). Pour ce faire, aidez-vous de la clause `if` applicable à la directive `#pragma omp parallel`.

2.2. Partage de travail

2.2.1. Exercice 1

Écrivez un programme en C qui :

1. déclare et initialise un scalaire `val` de type `double`,
2. déclare deux tableaux de valeurs de type `double` de la même taille `N`, `tab1` et `tab2`,
3. initialise `tab1` avec des valeurs arbitraires, par exemple `tab1[i] = 0.5 * i`,
4. effectue la somme de chaque élément de `tab1` et de `val` et enregistre le résultat dans `tab2`.

Dans ce programme, `N` sera une macro définie au début du fichier `.c` correspondant, par exemple `#define N 20`.

Parallélisez le calcul de la somme dans ce programme avec OpenMP. Selon vous, quelle directive de partage de travail est la mieux adaptée dans ce cas et pourquoi ?

2.2.2. Exercice 2

Étudiez, compilez et exécuter le code source ci-dessous, puis répondez aux questions qui le suivent.

```c
#include <stdio.h>
#include <omp.h>
#define N  100
#define PORTION  10

int main() {
```

```
  int tid;
  double a[N], b[N], c[N];

  for(size_t i = 0; i < N; i++) {
    a[i] = b[i] = i;
  }

  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    if(tid == 0) {
      printf("Nombre de fil. exécution = %d\n", omp_get_num_threads());
    }

    printf("Fil d'exécution %d: démarrage...\n", tid);

    #pragma omp for schedule(dynamic, PORTION)
    for(size_t i = 0; i < N; i++) {
      c[i] = a[i] + b[i];
      printf("Fil d'exécution %d: c[%2zu] = %g\n", tid, i, c[i]);
    }
  }
  return 0;
}
```

- Quelles instructions sont exécutées par tous les fils d'exécution ? Par un seul fil d'exécution ?
- Exécutez le même programme plusieurs fois de suite et commentez l'ordre d'exécution des instructions.
- Exécutez le programme en rédirigeant sa sortie standard vers l'outil `sort` de tri lexicographique, puis observez et commentez la répartition des itérations par fil d'exécution.
- Vérifiez si la répartition reste stable à travers plusieurs exécutions. Pourquoi ?
- À présent, utilisez la politique d'ordonnancement `static`. Exécutez le programme plusieurs fois et commentez sur la stabilité de la répartition des itérations entre les fils d'exécution.
- Quels sont les effets potentiels de la politique de l'ordonnancement choisie sur les performances ?

2.2.3. Exercice 3

Étudiez, compilez et exécuter le code source ci-dessous, puis répondez aux questions qui le suivent.

```
#include <stdio.h>
#define N 100

int main() {
  double a[N], b[N], c[N], d[N];

  for(size_t i = 0; i < N; i++)
    a[i] = b[i] = i;
```

```
  #pragma omp parallel
  {
    #pragma omp for schedule(static) nowait
    for(size_t i = 0; i < N; i++) {
      c[i] = a[i] + b[i];
    }

    #pragma omp for schedule(static)
    for(size_t i = 0; i < N; i++) {
      d[i] = a[i] + c[i];
    }
  }

  for(size_t i = 0; i < N; i++) {
    printf("%g ", d[i]);
  }

  printf("\n");

  return 0;
}
```

- Répétez l'exécution plusieurs fois. Les résultats sont-ils cohérents ?
- Quelles itérations vont être exécutées par quels fils d'exécution ?
- Est-ce raisonnable d'utiliser la clause `nowait` dans ce cas-ci ?
- Ordonnancez l'exécution de la seconde boucle suivant la politique `guided`, exécutez le programme plusieurs fois et commentez sur la cohérence des résultats.

2.2.4. Exercice 4

Dans cet exercice, nous nous basons sur une implémentation de l'algorithme de tri par énumération. Elle comporte deux fonctions de calcul `enumeration_sort_reference` et `enumeration_sort_kernel`. La première représente une implémentation séquentielle de référence de l'algorithme de tri. La dernière représente une implémentation séquentielle à paralléliser.

Étudiez le code source et parallélisez le calcul dans la fonction `enumeration_sort_kernel` à l'aide des directives et des clauses OpenMP que vous jugerez adaptées à ce cas de figure.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#define MAX 5
#define N 10240

// Implémentation séquentielle de référence (ne pas modifier)
void enumeration_sort_reference(double tab[N]) {
  size_t i, j;
  size_t * position = malloc(N * sizeof(size_t));
  double * copy     = malloc(N * sizeof(size_t));
```

```
  for(i = 0; i < N; i++) {
    position[i] = 0;
    copy[i] = tab[i];
  }

  for(j = 0; j < N; j++) {
    for(i = 0; i < N; i++) {
      if((tab[j] < tab[i]) || ((tab[i] == tab[j]) && (i < j))) {
        position[i]++;
      }
    }
  }

  for(i = 0; i < N; i++) {
    tab[position[i]] = copy[i];
  }

  free(position);
  free(copy);
}
// Implémentation à paralléliser
void enumeration_sort_kernel(double tab[N]) {
  size_t i, j;
  size_t * position = malloc(N * sizeof(size_t));
  double * copy     = malloc(N * sizeof(size_t));

  for(i = 0; i < N; i++) {
    position[i] = 0;
    copy[i] = tab[i];
  }

  for(j = 0; j < N; j++) {
    for(i = 0; i < N; i++) {
      if((tab[j] < tab[i]) || ((tab[i] == tab[j]) && (i < j))) {
        position[i]++;
      }
    }
  }

  for(i = 0; i < N; i++) {
    tab[position[i]] = copy[i];
  }

  free(position);
  free(copy);
}
// Fonction d'affichage
void print_sample(double tab[], size_t size, size_t sample_length) {
  if (size <= 2 * sample_length) {
    for(size_t i = 0; i < size; i++) {
      printf("%g ", tab[i]);
    }
  } else {
    for(size_t i = 0; (i < size) && (i < sample_length); i++) {
```

```
      printf("%g ", tab[i]);
    }

    printf("... ");

    for(size_t i = size - sample_length; i < size; i++) {
      printf("%g ", tab[i]);
    }
  }

  printf("\n");
}

int main(void) {
  double * a   = malloc(N * sizeof(double));
  double * ref = malloc(N * sizeof(double));
  double time_reference, time_kernel;

  // Initialisation des tableaux 'a' et 'ref' avec des valeurs aléatoires.
  srand((unsigned int) time(NULL));
  for(size_t i = 0; i < N; i++) {
    a[i] = (float) rand() / (float) (RAND_MAX / MAX);
    ref[i] = a[i];
  }

  // Exécution de l'implémentation séquentielle de référence.
  time_reference = omp_get_wtime();
  enumeration_sort_reference(ref);
  time_reference = omp_get_wtime() - time_reference;
  printf("Tri séquentiel : %3.5lf s\n", time_reference);
  // Exécution de l'implémentation parallélisée.
  time_kernel = omp_get_wtime();
  enumeration_sort_kernel(a);
  time_kernel = omp_get_wtime() - time_kernel;
  printf("Tri parallèle : %3.5lf s\n", time_kernel);

  // Affichage des extraits des tableaux triés.
  print_sample(ref, N, 5);
  print_sample(a, N, 5);

  // Comparaison des résultats.
  for(size_t i = 0; i < N; i++) {
    if(ref[i] != a[i]) {
      printf("Mauvais résultats !\n");
      exit(1);
    }
  }

  printf("Ça roule !\n");

  free(a);
  free(ref);
  return 0;
}
```

2.3. Status des variables

2.3.1. Exercice 1

Compilez le code source suivant avec et sans la clause `private(val)`. Exécutez les deux versions du programme. Qu'observez-vous et pourquoi ?

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(void) {
  int val;
  #pragma omp parallel
  {
    val = rand();
    sleep(1);
    printf("val : %d\n", val);
  }
  return 0;
}
```

2.3.2. Exercice 2

Écrivez un programme en C qui :

- déclare un tableau `tab` de valeurs de type `double` et de taille `N`,
- initialise `tab` avec des valeurs arbitraires,
- calcule et affiche la somme de tous les éléments du tableau.

Dans ce programme, `N` sera une macro définie au début du fichier `.c` correspondant, par exemple `#define N 1024`.

Parallélisez le calcul de la somme dans ce programme avec la directive de partage de travail `for` d'OpenMP. Quelle clause faudra-t-il utiliser avec cette directive pour obtenir une somme correcte ?

```c
#include <stdio.h>
#include <omp.h>
#define N 1024

int main(void) {
  double tab[N], somme = 0.0;

  for(int i = 0; i < N; i++) {
    tab[i] = 1.0;
  }

  #pragma omp parallel for reduction(+:somme)
  for(int i = 0; i < N; i++) {
    somme += tab[i];
  }

  printf("Somme : %.2f\n", somme);
```

```
  return 0;
}
```

2.3.3. Exercice 3

Étudiez, compilez et exécuter le code source ci-dessous. Ensuite, expliquez les valeurs des variables `tid`, `tprivate` et `rprivate` affichées lors de l'exécution.

```
#include <stdio.h>
#include <omp.h>

int tid, tprivate, rprivate;
#pragma omp threadprivate(tprivate)

int main() {
  printf("Région parallèle 1\n");
  #pragma omp parallel private(tid, rprivate)
  {
    tid = omp_get_thread_num();
    tprivate = tid;
    rprivate = tid;
    printf(
      "Fil d'exécution %d: tprivate=%d rprivate=%d\n", tid, tprivate, rprivate
    );
  }

  printf("Région parallèle 2\n");
  #pragma omp parallel private(tid, rprivate)
  {
    tid = omp_get_thread_num();
    printf(
      "Fil d'exécution %d: tprivate=%d rprivate=%d\n",
      tid, tprivate, rprivate
    );
  }
  return 0;
}
```

2.3.4. Exercice 4

Le code source ci-dessous permet de calculer une approximation de la valeur de π. Il y a deux fonctions de calcul `pi_reference` et `pi_kernel`. La première représente une implémentation séquentielle de référence du calcul de π. La dernière représente une implémentation séquentielle à paralléliser.

Étudiez le code source et parallélisez le calcul dans la fonction `prime_kernel` à l'aide des directives et des clauses OpenMP que vous jugerez adaptées à ce cas de figure.

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

```c
#include <math.h>
#include <omp.h>
#define PI "3.141592653589793238462"
#define PRECISION 1.e-10
#define MAX 5
#define N 51200000

// Implémentation séquentielle de référence (ne pas modifier)
void pi_reference(size_t nb_steps, double * pi) {
  double term;
  double sum = 0.;
  double step = 1. / (double) nb_steps;

  for(size_t i = 0; i < nb_steps; i++) {
    term = (i + 0.5) * step;
    sum += 4. / (1. + term * term);
  }

  (*pi) = step * sum;
}

// Implémentation à paralléliser
void pi_kernel(size_t nb_steps, double * pi) {
  double term;
  double sum = 0.;
  double step = 1. / (double) nb_steps;

  for(size_t i = 0; i < nb_steps; i++) {
    term = (i + 0.5) * step;
    sum += 4. / (1. + term * term);
  }

  (*pi) = step * sum;
}
int main(void) {
  double pi, pi_ref;
  double time_reference, time_kernel;

  // Exécution de l'implémentation séquentielle de référence.
  time_reference = omp_get_wtime();
  pi_reference(N, &pi_ref);
  time_reference = omp_get_wtime() - time_reference;
  printf("Calcul séquentiel : %3.5lf s\n", time_reference);
  // Exécution de l'implémentation parallélisée.
  time_kernel = omp_get_wtime();
  pi_kernel(N, &pi);
  time_kernel = omp_get_wtime() - time_kernel;
  printf("Calcul parallèle : %3.5lf s\n", time_kernel);

  printf("Pi : %s\n", PI);
  printf("Pi (résultat du calcul de référence) : %.22g\n", pi_ref);
  printf("Pi (résultat du calcul parallèle) : %.22g\n", pi);

  // Comparaison des résultats.
```

```
  if (fabs(pi_ref - pi) > PRECISION) {
    printf("Mauvais résultats !\n");
    exit(1);
  }

  printf("Ça roule !\n");

  return 0;
}
```

2.3.5. Exercice 5

Le code source ci-dessous permet de calculer le produit d'une matrice par un vecteur. Il y a deux fonctions de calcul `matvec_reference` et `matvec_kernel`. La première représente une implémentation séquentielle de référence du produit matrice-vecteur. La dernière représente une implémentation séquentielle à paralléliser.

Étudiez le code source et parallélisez le calcul dans la fonction `matvec_kernel` à l'aide des directives et des clauses OpenMP que vous jugerez adaptées à ce cas de figure.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#define PRECISION 1.e-20
#define MAX 5
#define N 5120

// Implémentation séquentielle de référence (ne pas modifier)
void matvec_reference(double c[N], double A[N][N], double b[N]) {
  size_t i, j;

  for(i = 0; i < N; i++) {
    c[i] = 0.;
    for(j = 0; j < N; j++) {
      c[i] += A[i][j] * b[j];
    }
  }
}

// Implémentation à paralléliser
void matvec_kernel(double c[N], double A[N][N], double b[N]) {
  size_t i, j;

  for(i = 0; i < N; i++) {
    c[i] = 0.;
    for(j = 0; j < N; j++) {
      c[i] += A[i][j] * b[j];
    }
  }
}

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

```
  double * A   = malloc(N * N * sizeof(double));
  double * b   = malloc(N * sizeof(double));
  double * c   = malloc(N * sizeof(double));
  double * ref = malloc(N * sizeof(double));
  double time_reference, time_kernel;

  // Initialisation des tableaux 'b' et 'A' avec des valeurs aléatoires.
  srand((unsigned int) time(NULL));
  for(size_t i = 0; i < N; i++)
    b[i] = (float) rand() / (float) (RAND_MAX / MAX);
  for(size_t i = 0; i < N * N; i++)
    A[i] = (float) rand() / (float) (RAND_MAX / MAX);

  // Exécution de l'implémentation séquentielle de référence.
  time_reference = omp_get_wtime();
  matvec_reference(ref, (double (*)[N])A, b);
  time_reference = omp_get_wtime() - time_reference;
  printf("Calcul séquentiel : %3.5lf s\n", time_reference);
  // Exécution de l'implémentation parallélisée.
  time_kernel = omp_get_wtime();
  matvec_kernel(c, (double (*)[N])A, b);
  time_kernel = omp_get_wtime() - time_kernel;
  printf("Calcul parallèle : %3.5lf s\n", time_kernel);

  // Affichage et comparaison des résultats.
  for (size_t i = 0; i < N; i++) {
    printf("ref[%2d] = %f, c[%d] = %f\n", i, ref[i], i, c[i]);
    if (abs(ref[i] - c[i]) > PRECISION) {
      printf("Mauvais résultats !\n");
      exit(1);
    }
  }
  printf("Ça roule !\n");

  free(A);
  free(b);
  free(c);
  free(ref);
  return 0;
}
```

2.4. Synchronisation

2.4.1. Exercice 1

Dans le code source suivant, reprenant un exemple vu en cours, deux fils d'exécution peuvent se retrouver en situation de compétition lors de l'incrémentation de `n`. Dans son état actuel, le programme est donc susceptible de produire un résultat incorrect.

De quelle façon pourrions-nous éviter la situation de compétition dans ce programme ? Implémentez votre correctif.

```c
#include <stdio.h>
#define MAX 10000

int main() {
  size_t i;
  int n = 0;

  #pragma omp parallel for
  for(i = 0; i < MAX; i++) {
    n++;
  }

  printf("n = %d\n", n);
  return 0;
}
```

2.4.2. Exercice 2

Le code source suivant, reprenant un exemple vu en cours, comporte un risque de situation de compétition et un défaut de cohérence. Dans son état actuel, le programme est donc susceptible d'afficher le message « Fini » avant le message « Pas fini ».

De quelle façon pourrions-nous éviter la situation de compétition et le défaut de cohérence dans ce programme ? Implémentez votre correctif.

```c
#include <stdio.h>

int main() {
  int fin = 0;

  #pragma omp parallel sections
  {
    #pragma omp section
    {
      while(!fin) {
        printf("Pas fini\n");
      }
    }
    #pragma omp section
    {
      fin = 1;
      printf("Fini\n");
    }
  }
  return 0;
}
```

2.4.3. Exercice 3

Le code source suivant, reprenant un exemple vu en cours, comporte un défaut de synchronisation. Dans son état actuel, le programme est donc susceptible de produire un résultat incorrect.

De quelle façon pourrions-nous éviter le défaut de synchronisation dans ce programme ? Implémentez votre correctif.

```c
#include <stdio.h>
#include <omp.h>
int main() {
  double total, part1, part2;

  #pragma omp parallel num_threads(2)
  {
    int tid;
    tid = omp_get_thread_num();

    if(tid == 0) {
      part1 = 25;
    }

    if(tid == 1) {
      part2 = 17;
    }

    if(tid == 0) {
      total = part1 + part2;
      printf("%g\n", total);
    }
  }

  return 0;
}
```

2.4.4. Exercice 4

Le code source ci-dessous permet de calculer tous les nombres premiers entre `PRIME_MIN` et `PRIME_MAX`. Il y a deux fonctions de calcul `prime_reference` et `prime_kernel`. La première représente une implémentation séquentielle de référence du calcul des nombres premiers. La dernière représente une implémentation séquentielle à paralléliser.

Étudiez le code source et parallélisez le calcul des nombres premiers dans la fonction `prime_kernel` à l'aide des directives et des clauses OpenMP que vous jugerez adaptées à ce cas de figure.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <omp.h>
#define PRIME_MIN  3
#define PRIME_MAX  100000

// Implémentation séquentielle de référence (ne pas modifier)
void prime_reference(size_t primes[], size_t * ptr_nb_primes) {
  size_t nb_primes = 0;
  size_t divisor;
  bool is_prime;
```

```
  for(size_t i = PRIME_MIN; i < PRIME_MAX; i += 2) {
    is_prime = true;
    divisor = PRIME_MIN;

    while((divisor < i) && is_prime) {
      if((i % divisor) == 0) {
        is_prime = false;
      }
      divisor += 2;
    }

    if(is_prime) {
      primes[nb_primes] = i;
      nb_primes++;
    }
  }

  (*ptr_nb_primes) = nb_primes;
}

// Implémentation à paralléliser
void prime_kernel(size_t primes[], size_t * ptr_nb_primes) {
  size_t nb_primes = 0;
  size_t divisor;
  bool is_prime;

  for (size_t i = PRIME_MIN; i < PRIME_MAX; i += 2) {
    is_prime = true;
    divisor = PRIME_MIN;

    while((divisor < i) && is_prime) {
      if((i % divisor) == 0) {
        is_prime = false;
      }
      divisor += 2;
    }

    if (is_prime) {
      primes[nb_primes] = i;
      nb_primes++;
    }
  }

  (*ptr_nb_primes) = nb_primes;
}

// Fonction d'affichage
void print_sample(size_t tab[], size_t size, size_t sample_length) {
  if (size <= 2 * sample_length) {
    for(size_t i = 0; i < size; i++) {
      printf("%zu ", tab[i]);
    }
  } else {
```

```
    for(size_t i = 0; (i < size) && (i < sample_length); i++) {
      printf("%zu ", tab[i]);
    }

    printf("... ");

    for(size_t i = size - sample_length; i < size; i++) {
      printf("%zu ", tab[i]);
    }
  }

  printf("\n");
}

int main() {
  size_t * primes_ref = malloc(PRIME_MAX / 2 * sizeof(size_t));
  size_t * primes     = malloc(PRIME_MAX / 2 * sizeof(size_t));
  size_t nb_primes_ref, nb_primes;
  double time_reference, time_kernel;

  // Exécution de l'implémentation séquentielle de référence.
  time_reference = omp_get_wtime();
  prime_reference(primes_ref, &nb_primes_ref);
  time_reference = omp_get_wtime() - time_reference;
  printf("Calcul séquentiel : %3.5lf s\n", time_reference);
  // Exécution de l'implémentation parallélisée.
  time_kernel = omp_get_wtime();
  prime_kernel(primes, &nb_primes);
  time_kernel = omp_get_wtime() - time_kernel;
  printf("Calcul parallèle : %3.5lf s\n", time_kernel);

  // Affichage des extraits des résultats.
  print_sample(primes_ref, nb_primes_ref, 5);
  print_sample(primes, nb_primes, 5);

  // Comparaison des résultats.
  if(nb_primes_ref != nb_primes) {
    printf(
      "Mauvais résultat ! "
      "Le nombre de nombres premiers calculés n'est pas correct.\n"
    );
    exit(1);
  }

  int faux = 0;
  for(size_t i = 0; i < nb_primes; i++) {
    if(primes_ref[i] != primes[i]) {
      faux++;
    }
  }

  if(faux) {
    printf("Mauvais résultat ! Les nombres premiers ne correspondent pas.\n");
    exit(1);
```

```
    }

    printf("Ça roule !\n");

    free(primes_ref);
    free(primes);
    return 0;
}
```