Exam — Parallel Programming

1ère session — 17 janvier 2024 — Durée : 2 heures.

Les documents sont autorisés. **Comme dans tous les examens, vos réponses doivent être justifiées.** Il y a trois parties plus ou moins indépendantes.

On considère le célèbre problème NP-complet qui consiste à trouver une clique maximum dans un graphe non-orienté. On s'intéresse ici à l'algorithme décrit en 1973 par deux chercheurs hollandais, Joep Kerbosch et Coenraad Bron. Leur algorithme consiste à énumérer toutes les cliques *maximales* d'un graphe, sachant que les cliques *maximum* font partie du lot [1]

Dans la suite, on considère un graphe non-orienté $G = (V, E)$, et pour un sommet $u \in V$, on note $\Gamma(u) = \{w \in V : u \leftrightarrow w \in E\}$ l'ensemble des sommets adjascents à u dans G.

L'algorithme fonctionne en maintenant trois ensembles :

- $K_{\max}$ contient la plus grande clique trouvée jusqu'à présent
- K contient la clique courante (pas forcément maximale)
- C contient l'ensemble des sommets qui peuvent être ajoutés à K pour former une clique plus grosse (ce sont les « candidats » ; donc si $C = \emptyset$, alors K est maximale)
- A contient un sous-ensemble de C qu'on accepte d'ajouter à K (les « candidats autorisés »)

L'idée derrière l'utilisation de l'ensemble A est qu'une fois qu'on a énuméré toutes les cliques qui contiennent le sommet x, alors on peut s'interdire de considérer x dans toute la suite. L'algorithme maintient une variable globale $K_{\max}$ qui contient la plus grande clique trouvée jusqu'à présent. Voici son pseudo-code :

```
1:  function MaxClique(K, C, A)
2:      if |K| + |A| ≤ |K_max| then          ▷ Optimisation : essaye de couper court aux calculs
3:          return                             ▷ On ne pourra pas avoir mieux
4:      if C = ∅ then
5:          if |K| > |K_max| then              ▷ K est maximale
6:              K_max ← K
7:      else
8:          Choisir u ∈ C qui maximise |C ∩ Γ(u)|   ▷ Choisis le meilleur « pivot » (cf. infra)
9:          while A − Γ(u) ≠ ∅ do
10:             Choisir x ∈ A − Γ(u)           ▷ Ajoute le sommet x à la clique courante
11:             MaxClique(K ∪ {x}, C ∩ Γ(x), A ∩ Γ(x))
12:             A ← A − {x}                    ▷ À partir de maintenant, ajouter x est interdit

    # Ceci stocke dans une clique maximum de G
13: K_max ← ∅
14: print MaxClique(∅, V, V).
```

Voici des précisions supplémentaires (il n'est pas nécessaire de les comprendre pour faire l'examen, mais l'auteur du sujet aime être complet).

- Le test de la ligne 2 est correct car dans le meilleur des cas, tous les sommets de A vont être ajoutés à la clique courante K. Donc la clique maximale à laquelle on peut aboutir dans l'état courant est de taille majorée par $|K| + |A|$.
- Le sommet $u \in C$ (qu'on appelle le « pivot ») joue le rôle suivant. Supposons qu'on a énuméré toutes les cliques maximales contenant $K \cup \{u\}$. Alors, chaque nouvelle clique contenant K mais pas u doit contenir un sommet qui n'est pas adjacent à u. Justification (par l'absurde) : on étend K en une clique maximale $K \cup S$, où tous les éléments de S sont adjacents à u. Alors, comme u est adjacent à tous les éléments de K, on trouve que $K \cup S \cup \{u\}$ est une clique plus grosse, ce qui contredit la maximalité de la précédente.

1. Une clique est maximale si elle n'est pas contenue dans une clique plus grande. Une clique est maximum si elle est de taille maximale.

Exercice 1 – MaxClique avec OpenMP

1. Ecrivez une version multithread simple et directe de cet algorithme en ajoutant des directives OpenMP au pseudo-code ci-dessus.

Solution :

1: **function** MaxClique(K, C, A)
2: // *accès potentiellement conflictuel à* $K_{\max}$. *Cf commentaire après le code*
3: **if** $|K| + |A| \leq |K_{\max}|$ **then** ▷ Optimisation : essaye de couper court aux calculs
4: **return** ▷ On ne pourra pas avoir mieux
5: **if** $C = \emptyset$ **then**
6: `#pragma omp critical`
7: **if** $|K| > |K_{\max}|$ **then** ▷ K est maximale
8: $K_{\max} \leftarrow K$
9: **else**
10: Choisir $u \in C$ qui maximise $|C \cap \Gamma(u)|$ ▷ Choisis le meilleur « pivot » (cf. infra)
11: **while** $A - \Gamma(u) \neq \emptyset$ **do**
12: Choisir $x \in A - \Gamma(u)$ ▷ Ajoute le sommet x à la clique courante
13: `#pragma omp task`
14: MaxClique($K \cup \{x\}, C \cap \Gamma(x), A \cap \Gamma(x)$)
15: $A \leftarrow A - \{x\}$ ▷ À partir de maintenant, ajouter x est interdit

Ceci stocke dans une clique maximum de G
16: $K_{\max} \leftarrow \emptyset$
17: `#pragma omp parallel`
18: `#pragma omp single`
19: **print** MaxClique($\emptyset, V, V$).

En fait, il y a un conflit potentiel entre la lecture de $K_{\max}$ ligne 3 et l'écriture de $K_{\max}$ ligne 8. Mais ceci est facile à éviter : on peut ajouter une variable entière "size" qui contient $|K_{\max}|$. Alors on peut d'une part remplacer :

3: // *accès potentiellement conflictuel à* $K_{\max}$. *Cf commentaire après le code*
4: **if** $|K| + |A| \leq |K_{\max}|$ **then** ▷ Optimisation : essaye de couper court aux calculs
5: **return** ▷ On ne pourra pas avoir mieux

Par

3: `#pragma omp atomic read`
4: $size' \leftarrow size$
5: **if** $|K| + |A| \leq size$ **then** ▷ Optimisation : essaye de couper court aux calculs
6: **return** ▷ On ne pourra pas avoir mieux

et d'autre part remplacer :

6: `#pragma omp critical`
7: **if** $|K| > |K_{\max}|$ **then** ▷ K est maximale
8: $K_{\max} \leftarrow K$

Par

6: `#pragma omp critical`
7: **if** $|K| > size$ **then** ▷ K est maximale
8: `#pragma omp atomic write`
9: $size \leftarrow |K|$
10: $K_{\max} \leftarrow K$

2. Résoudre de petites instance du problème peut être très rapide avec cet algorithme. Modifiez votre réponse à la question précédente pour faire en sorte que les instances dont la taille est inférieure à un seuil donné soient traitées directement, séquentiellement.

Solution :

Le bon ensemble sur lequel mettre un seuil est A, car c'est celui qui indique le mieux quelle quantité de travail est restante.

Il faut remplacer :

13: `#pragma omp task`
14: MaxClique$(K \cup \{x\}, C \cap \Gamma(x), A \cap \Gamma(x))$

Par :

13: **if** $|A \cap \Gamma(x)| \geq$ seuil **then**
14: `#pragma omp task`
15: MaxClique$(K \cup \{x\}, C \cap \Gamma(x), A \cap \Gamma(x))$
16: **else**
17: MaxClique$(K \cup \{x\}, C \cap \Gamma(x), A \cap \Gamma(x))$

3. Chaque appel à MaxClique engendre *des* appels récursifs. Doit-on s'attendre à ce que ce derniers aient tous le même temps d'exécution, ou au contraire à une forte variabilité ?

Solution :

Forte variabilité, car le premier appel récursif a un ensemble A plus gros que le dernier, donc celui-ci va terminer plus rapidement.

4. Malheureusement, n'importe quelle version parallèle de cet algorithme risque d'effectuer un nombre d'opérations total plus élevé que la version séquentielle. Pourquoi ?

Solution :

Chaque fois qu'une meilleure valeur de K_{max} est trouvée, alors cela augmente la probabilité que le test de "coupure" de la ligne 3 abrège les calculs. Et donc, il se pourrait que le premier appel récursif définisse K_{max}, puis que le deuxième (qui a lieu *après* dans le code séquentiel) soit "coupé". Mais si les deux appels récursifs ont lieu "en même temps", alors cette coupure n'est pas possible, et le deuxième appel récursif sera exploré pour rien.

Exercice 2 – MaxClique avec MPI

1. Ecrivez une version modifiée du pseudo-code dans laquelle :
   — Il n'y a plus de variable globale
   — La fonction MaxClique prend un argument $K_{\max}$ qui contient la plus grande clique trouvée jusqu'à présent
   — La fonction MaxClique renvoie la plus grande clique trouvée jusqu'à présent

Solution :

1: **function** MaxClique$(K, C, A, K_{\max})$
2: **if** $|K| + |A| \leq |K_{\max}|$ **then** ▷ Optimisation : essaye de couper court aux calculs
3: **return** $K_{\max}$ ▷ On ne pourra pas avoir mieux
4: **if** $C = \emptyset$ **then**
5: **if** $|K| > |K_{\max}|$ **then** ▷ K est maximale
6: $K_{\max} \leftarrow K$
7: **else**
8: Choisir $u \in C$ qui maximise $|C \cap \Gamma(u)|$ ▷ Choisis le meilleur « pivot » (cf. infra)
9: **while** $A - \Gamma(u) \neq \emptyset$ **do**
10: Choisir $x \in A - \Gamma(u)$ ▷ Ajoute le sommet x à la clique courante
11: $K_{\max} \leftarrow$ MaxClique$(K \cup \{x\}, C \cap \Gamma(x), A \cap \Gamma(x), K_{\max})$
12: $A \leftarrow A - \{x\}$ ▷ À partir de maintenant, ajouter x est interdit
13: **return** $K_{\max}$

Ceci stocke dans une clique maximum de G

14: $K_{\max} \leftarrow \emptyset$
15: **print** MaxClique$(\emptyset, V, V)$.

2. On peut supposer que les sommets de G sont représentés par des entiers de l'intervalle $[0; n-1]$. Comment faire pour transmettre un ensemble d'entiers à un autre processus avec des fonctions MPI ? (écrivez le code d'envoi ET de réception).

Solution :
Il faut d'abord envoyer la taille de l'ensemble, puis l'ensemble lui-même.

```
// envoi de l'ensemble A[0:n]
MPI_Send(&n, 1, MPI_INT, dest, tag, comm);
MPI_Send(A, n, MPI_INT, dest, tag, comm);

// réception de l'ensemble A[0:n]
MPI_Recv(&n, 1, MPI_INT, dest, tag, comm, MPI_STATUS_IGNORE);
MPI_Recv(A, n, MPI_INT, dest, tag, comm, MPI_STATUS_IGNORE);
```

3. ($\star\star\star$) Proposez le pseudo-code d'une version distribuée avec MPI de l'algorithme ci-dessus. On suggère la stratégie suivante. Les processus de rang $1, 2, \ldots$ sont des ouvriers. Le processus de rang 0 (« l'ingénieur ») exécute l'algorithme séquentiel. Dans la boucle **while** des lignes 9–12, si ce n'est pas la dernière itération et qu'un ouvrier est disponible, alors l'ingénieur lui *délègue* l'appel récursif. Dans le cas contraire, l'ingénieur effectue l'appel récursif lui-même.

Exercice 3 – Produit de matrice booléen par la méthode dite « des quatre russes »

Soient M une (petite) matrice 64×64 à coefficients modulo 2 et A une (grande) matrice de taille $N \times 64$ également à coefficients modulo 2. Une telle matrice de taille $k \times 64$ peut être commodément représentée en mémoire par un tableau de k entiers de 64 bits (un par ligne). On s'intéresse au calcul du produit $A \times M$.

Pour cela, on utilise un algorithme dû à V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev (*On economic construction of the transitive closure of a direct graph.* In Soviet Math (Doklady), volume 11, pages 1209–1210, 1970). Il est apparu ensuite que tous ne sont pas russes, mais le nom est resté...

Le calcul du produit vecteur-matrice $x \times A$ calcule une combinaison linéaire des lignes de A, dont les coefficients sont décrits par le vecteur x. Modulo 2, cela revient à calculer le XOR des lignes de A telles que $x_i = 1$ (x_i désigne le i-ème bit de x).

L'idée générale consiste à précalculer toutes les combinaisons linéaires de groupes de 8 lignes consécutives. Ensuite, le produit vecteur-matrice et matrice-matrice peuvent se faire avec les petites fonctions montrées figure 1.

1. La structure de données qui représente M tient-elle dans le cache L1 d'un processeur moderne ?

Solution :
Un CPU moderne a typiquement un cache d'au moins 32Ko. La matrice elle-même peut être représentée par 64 mots de 64 bits (comme l'énoncé l'indique), ce qui fait 512 octets, donc ça tient. Ceci dit, *ce n'était pas la question* !
Il était demandé si le `struct matmult_table_t` tient. Celui-ci est composé de 8×256 mots de 64 bits, ce qui fait 16384 octets. Mais ça tient aussi.

2. Quelle est l'intensité opérationnelle de la fonction `gemm()` ?

Solution :
Il faut bien noter que le `struct matmult_table_t` *tient en cache*, donc on y accède "gratuitement". La fonction `gemv()` contient 8 XORs, 7 shifts et 8 ANDs, ce qui fait 23 opérations arithmétiques ; elle ne fait pas d'accès mémoire hors-cache. On en conclut que `gemm()` a une

```c
typedef uint64_t u64;            // l'abbréviation u64 est utilisée dans le noyau Linux

struct matmul_table_t {          // contient les combinaisons linéaires précalculées
        u64 tables[8][256];
};

/* renvoie x*M */
u64 gemv(u64 x, const struct matmul_table_t *M)
{
        u64 r = 0;
        r ^= M->tables[0][x         & 0x00ff];
        r ^= M->tables[1][(x >>  8) & 0x00ff];
        r ^= M->tables[2][(x >> 16) & 0x00ff];
        r ^= M->tables[3][(x >> 24) & 0x00ff];
        r ^= M->tables[4][(x >> 32) & 0x00ff];
        r ^= M->tables[5][(x >> 40) & 0x00ff];
        r ^= M->tables[6][(x >> 48) & 0x00ff];
        r ^= M->tables[7][(x >> 56) & 0x00ff];
        return r;
}

/* calcule A <--- A*M */
u64 gemm(int N, u64 *A, const struct matmul_table_t *M)
{
        for (int i = 0; i < N; i++)
                A[i] = gemv(A[i], M);
}
```

FIGURE 1 – Calcul du produit (grande matrice) × (petite matrice) par la méthode des « quatre russes »

intensité opérationnelle de ($23N$ opérations) / ($8N$ octets). En effet, il n'y a que la lecture de $A[i]$ qui déclenche une lecture depuis la RAM. L'intensité opérationnelle est donc d'environ 3, ce qui est relativement élevé (les exemples en cours étaient sensiblement plus faibles, de l'ordre de 1/4).

3. Cette fonction est-elle plus vraissemblablement *CPU-bound* ou *memory-bound* ?

Solution :

Comme l'intensité opérationnelle est assez élevée, il est vraissemblable que cette opération sera CPU-bound (en cours, il a été dit que c'est souvent le cas du produit de matrice...).

Le fait que le tableau A est très grand n'est pas un argument suffisant pour conclure le contraire : tout le problème est "combien d'opérations arithmétiques ont lieu par mot transféreré" ?

L'argument que "les opérations binaires sont très rapides" ou que "il n'y a presque aucun calcul" est complètement faux. Un CPU Intel Xeon Moderne met rigoureusement autant de temps à faire un XOR sur des chaines de 512 bits qu'un "fused multiply-add" parallèle sur des vecteurs de 8 flottants double-precision (deux instructions par cycle, dans les deux cas). Il y a presque 24 opérations arithmétiques par mot de 64 bits de A. Un coeur d'un CPU moderne arrive à lire la RAM à (disons) ≈ 20Go / s dans de bonnes condition. Ca fait 2.5G lignes de A lues par seconde. Donc il faudrait pouvoir faire 60 milliards d'opérations par seconde pour tenir le rythme. Avec de l'AVX512, si on les fait 8 par 8, il faudrait pouvoir exécuter 7.5 milliards d'instructions SIMD par seconde. Ceci n'est pas du tout évident (il faut au minimum en faire 2 par cycle à presque 4GHz). Tout ceci milite pour le caractère CPU-bound.

On peut observer de manière expérimentale que ce produit matrice-matrice est CPU-bound sur un processeur PowerPC A2 ! Cf. Mellila Bouam, Charles Bouillaguet, Claire Delaplace, Camille Noûs : *Computational records with aging hardware : Controlling half the output of SHA-256*. Parallel Comput. 106 : 102804 (2021)

4. Ecrivez une version multithreads de la fonction `gemm()`

Solution :

```
u64 gemm(int N, u64 *A, const struct matmul_table_t *M)
{
        #pragma omp parallel for
        for (int i = 0; i < N; i++)
                A[i] = gemv(A[i], M);
}
```

5. Ecrivez une version vectorisée AVX512 de `gemm` en utilisant des *intrinsics*. On peut supposer que n est un multiple de 8. Un petit examen du « guide des intrinsics » est montré figure 2.

Solution :

```
u64 gemm_avx512(int N, u64 *A, const struct matmul_table_t *M)
{
        assert(N % 8 == 0);
        __m512i mask = _mm512_set1_epi64(0x000ff);
        for (int i = 0; i < N; i += 8) {
                __m512i x = _mm512_load_si512(&A[i]);
                __m512i r = _mm512_setzero_si512();
                for (int j = 0; j < 8; j++) {
                    __m512i idx = _mm512_and_epi64(mask, _mm512_srli_epi64(x, 8*j));
                    __m512i t = _mm512_i64gather_epi64(idx, &M->tables[j], 8);
                    r = _mm512_xor_epi64(r, t);
                }
                _mm512_store_si512(&A[i], r);
        }
}
```

6. On considère que votre code est exécuté sur un coeur d'un processeur Intel « skylake » récent. Un coeur est capable d'exécuter plusieurs instructions en parallèle, si elles ne sont pas en conflit. Pour cela, le coeur a des *ports* dédiés à des instructions différentes. Chaque port peut exéuter une instruction par cycle dans le meilleur des cas. Cf. figure 3. Donner une borne inférieure sur le nombre de cycles nécessaire à l'exécution de la fonction. Combien de « BOPS » (binary operation per second) est-ce que cela représente-t-il ?

Solution :

Il y a $N/8\times$ (loads + store) et $N\times$ (AND + XOR + SRL + gather).

Notons p_i le nombre minimal de cycles où le port i est occupé. À cause des stores, on a donc $n_4 = N/8$. Admettons pour simplifier que les stores occupent aussi tout le temps le port 7 plutôt que le 2 ou le 3 qui sont sous la pression des loads. Les loads imposent que $n_2 + n_3 = \#\text{load} + 8\#\text{gather}$. Quant aux opérations arithmétiques, elles imposent que $n_0 = \#\text{shift} + \#\text{gather} + x$ et $n_5 = \#\text{gather} + y$, où $x + y = \#textand + \#\text{xor}$ (les deux variables x et y disent comment les AND/XOR se répartissent entre le port 0 et le port 5). Pour résumer on a :

$$\begin{aligned} n_0 &= 2N + x \\ n_2 + n_3 &= N/8 + 8N \\ n_4 &= N/8 \\ n_5 &= N + y \\ x + y &= 2N \end{aligned}$$

Et le problème est de trouver $\min\{n_0, n_2, n_3, n_4, n_5\}$. Sous cette forme générale, c'est un problème de programmation linéaire. Mais en fait, quand on le regarde attentivement, on voit vite

```
/* Return vector of type __m512i with all elements set to zero. */
__m512i _mm512_setzero_si512();

/* Broadcast 64-bit integer a to all elements of dst. */
__m512i _mm512_set1_epi64(u64 a);

/* Load 512-bits of integer data from memory into dst.  */
__m512i _mm512_load_si512(void const* mem_addr);

/* Compute the bitwise XOR/AND of 512 bits in a and b, and store the result in dst. */
__m512i _mm512_xor_epi64(__m512i a, __m512i b);
__m512i _mm512_and_epi64(__m512i a, __m512i b);

/* Store 512-bits of integer data from a into memory. */
void _mm512_store_si512(void* mem_addr, __m512i a);

/*
 * Shift packed 64-bit integers in a right by imm8 while shifting in zeros,
 * and store the results in dst.
 */
__m512i _mm512_srli_epi64(__m512i a, int imm8);

/*
 * Gather 64-bit integers from memory using 64-bit indices. 64-bit elements are
 * loaded from addresses starting at base_addr and offset by each 64-bit element
 * in vindex (each index is scaled by the factor in scale). Gathered elements
 * are merged into dst. scale should be 1, 2, 4 or 8.
 *
 * FOR j := 0 to 7
 *         i := j*64
 *         m := j*64
 *         addr := base_addr + vindex[m+63:m] * scale * 8
 *         dst[i+63:i] := MEM[addr+63:addr]
 * ENDFOR
 */
__m512i _mm512_i64gather_epi64(__m512i vindex, void const* base_addr, int scale);
```

FIGURE 2 – Extrait du guide des fonctions intrinsics.

que c'est le terme en $8N$ qui vient des gather qui va tout dominer. Dans le meilleur des cas, ça va se répartir équitablement entre les ports 2 et 3, et on va avoir $n_2 = n_3 = N/16 + 4N$.
Pour s'en convaincre, on peut voir qu'avec $x = 0.5N$ et $y = 1.5N$, alors $n_0 = 2.5N$ et $n_5 = 2.5N$, ce qui est moins que $4N$.
Donc en fait, la borne inférieure vient de la lecture des "tables", et on constate que gather n'est pas très efficace. Au final, ça semble un peu... memory-bound !

Operation	port(s)
shift	0
xor/and	0 | 5
load	2 | 3
store	(2 | 3 | 7) & 4
gather	0 & 5 ; 8 × (2 | 3)

Ce tableau se lit de la façon suivante : l'instruction `shift` s'exécute sur le port 0 (il peut donc y en avoir un seul par cycle). Le XOR ou le AND peuvent s'exécuter indifféremment sur le port 0 ou sur le port 5 (donc il peut y en avoir deux par cycles). Le `store` (copie du contenu d'un registre vers la mémoire) occupe le port 4 ainsi qu'un autre port parmi $\{2, 3, 7\}$ (mais à cause du port 4 il ne peut y avoir qu'un seul `store` par cycle). `gather` occupe les port 0 et 5 pendant un cycle *puis* envoie 8 « micro-instructions » qui peuvent s'exécuter sur les ports 2 ou 3 (il s'ensuit que `gather` prend 5 cycles en tout au minimum).

FIGURE 3 – Détail pour les instructions qui nous intéressent (source : `https://uops.info`)