

# **Projet OpenMP - Calcul parallèle de la somme de deux grands entiers avec utilisation d'algorithmes de type Carry Look Ahead**

## **Le Problème :**

Le problème abordé dans ce projet consiste à développer un programme parallèle capable de calculer la somme de deux grands entiers de manière efficace en utilisant OpenMP pour exploiter le parallélisme disponible sur les architectures multi-cœurs.

Le calcul de la somme de grands entiers peut être une tâche intensive en termes de calcul, surtout lorsque les entiers impliqués sont de taille considérable. Avec l'augmentation de la taille des données et l'avènement des architectures multicœurs, il est devenu essentiel de paralléliser les opérations arithmétiques pour exploiter pleinement les ressources matérielles disponibles et accélérer le processus de calcul.

L'objectif de ce projet est donc de développer un programme parallèle capable de tirer parti du parallélisme offert par les architectures multi-cœurs pour accélérer le calcul de la somme de deux grands entiers. Pour ce faire, nous explorerons l'utilisation d'algorithmes de type Carry Look Ahead (CLA) pour optimiser le processus de calcul de la somme.

L'algorithme Carry Look Ahead (CLA) est une méthode efficace pour effectuer des opérations de somme avec retenue (carry) en parallèle. En générant les retenues nécessaires de manière anticipée, l'algorithme CLA permet d'éviter des calculs redondants et d'accélérer le processus global de calcul de la somme des entiers.

Pour réaliser ce projet, nous utiliserons le framework OpenMP, une API de programmation parallèle qui simplifie le développement de programmes parallèles sur architectures multi-cœurs en utilisant des directives simples et efficaces. En utilisant OpenMP, nous créerons des régions parallèles où les calculs impliqués dans l'algorithme CLA seront répartis entre plusieurs threads, permettant ainsi une exécution plus rapide et efficace des opérations arithmétiques.

## Algorithme Carry Look Ahead :

**Carry Generate (G) :** cette fonction indique la façon dont le report est généré pour deux entrées à un seul bit, indépendamment de tout report d'entrée.

Comme nous l'avons vu dans l'additionneur complet, le report est généré en utilisant l'équation  $AB$

Par conséquent,  $G = A \cdot B$  (similaire à la façon dont le report est généré par un additionneur complet)

**Carry Propagate (P) :** Cette fonction indique le moment où le report est propagé à l'étape suivante avec un ajout chaque fois qu'il y a un report d'entrée.

Considérons un seul bit deux entrées A et B.

A	B	Carry In	Description
0	0	1	Le report n'est pas propagé (0)
0	1	1	Carry se propage (1)
1	0	1	Carry se propage (1)
1	1	1	Carry se propage (1)

Ainsi,  $P = A+B$

### Fonction de calcul de report pour l'étape suivante

$$C_{j+1} = G_j + (P_j \cdot C_j)$$

$$= A_j \cdot B_j + (A_j + B_j) \cdot C_j$$

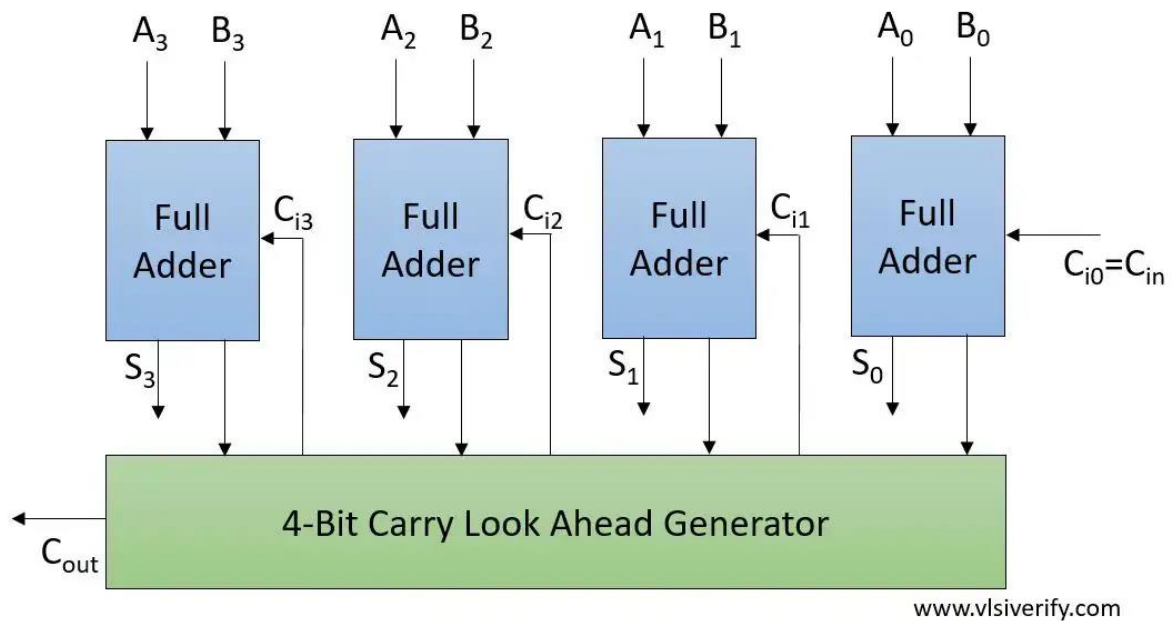
Cependant, si vous remarquez clairement chaque fois que  $A_j=1$  et  $B_j=1$

$C_{j+1} = 1$  (toujours) car le composant  $A_j \cdot B_j$  annule l'effet de la partie  $[(A_j + B_j) \cdot C_j]$ . Ainsi, cela n'a pas d'importance même si vous utilisez la fonction de propagation d'équation comme  $P = A (+) B$  comme mentionné dans d'autres publications.

$$G = AB \cdot B$$

$$P = A + B \text{ ou } A (+) B$$

## Diagramme



### 4-Bit Carry Look Ahead Adder

Calcul de toutes les retenues pour l'ajout de l'étape suivante

#### Contributions -

A<sub>3</sub> A<sub>2</sub> A<sub>1</sub> A<sub>0</sub>

B<sub>3</sub> B<sub>2</sub> B<sub>1</sub> B<sub>0</sub>

Et C<sub>in</sub>

#### Sortir

Somme = S<sub>3</sub> S<sub>2</sub> S<sub>1</sub> S<sub>0</sub>

Et C<sub>out</sub>

Somme  $S_j = P_j \wedge C_j = A_j \wedge B_j \wedge C_j$

$C_0 = C_{in}$

$C_1 = G_0 + (P_0 \cdot C_0)$

$= A_0 \cdot B_0 + (A_0 \wedge B_0) \cdot C_0$

$C_2 = G_1 + (P_1 \cdot C_1)$

$= A_1 \cdot B_1 + (A_1 \wedge B_1) \cdot (A_0 \cdot B_0 + (A_0 \wedge B_0) \cdot C_0)$

$C_3 = G_2 + (P_2 \cdot C_2)$

$= A_2 \cdot B_2 + (A_2 \wedge B_2) \cdot C_2$

$= A_2 \cdot B_2 + (A_2 \wedge B_2) \cdot (A_1 \cdot B_1 + (A_1 \wedge B_1) \cdot (A_0 \cdot B_0 + (A_0 \wedge B_0) \cdot C_0))$

$C_4 = G_3 + (P_3 \cdot C_3)$

$= A_3 \cdot B_3 + (A_3 \wedge B_3) \cdot C_3$

$= A_3 \cdot B_3 + (A_3 \wedge B_3) \cdot (A_2 \cdot B_2 + (A_2 \wedge B_2) \cdot (A_1 \cdot B_1 + (A_1 \wedge B_1) \cdot (A_0 \cdot B_0 + (A_0 \wedge B_0) \cdot C_0)))$

C<sub>out</sub> = C<sub>4</sub> ;

Notez qu'à chaque étape, nous dépendons en fin de compte des entrées A, B et Cin. Cela le rend en fait plus rapide par rapport à l'additionneur de report d'ondulation qui dépend fortement du report généré par l'étape précédente.

L'implémentation finale est ramenée au niveau des portes XOR, AND et OR.

## Approche initiale simple pour calculer deux grands entiers :

L'algorithme initial `addStrings` traite des nombres représentés en base 10 (décimale) sous forme de chaînes de caractères. Il effectue l'addition colonne par colonne, similairement à la méthode manuelle d'addition, en traitant les retenues entre les colonnes. Cet algorithme est simple à comprendre et à mettre en œuvre, mais peut-être moins efficace pour des opérations sur de grands nombres en raison de la complexité de manipulation des chiffres décimaux.

```
1  string addStrings(string num1, string num2)
2  {
3      string result;
4      int carry = 0;
5      int i = num1.size() - 1;
6      int j = num2.size() - 1;
7
8      while (i >= 0 || j >= 0 || carry)
9      {
10         int digit1 = (i >= 0) ? num1[i--] - '0' : 0;
11         int digit2 = (j >= 0) ? num2[j--] - '0' : 0;
12         int sum = digit1 + digit2 + carry;
13         result += (sum % 10) + '0';
14         carry = sum / 10;
15     }
16
17     reverse(result.begin(), result.end());
18     return result;
19 }
20
```

## Approche initiale avec l'Algorithme Carry Look Ahead :

L'autre algorithme, basé sur le principe du Carry Look Ahead (CLA), est plus avancé et est conçu spécifiquement pour des opérations arithmétiques sur de grands nombres binaires. Il exploite le parallélisme intrinsèque des opérations d'addition binaire (base 2) en générant des signaux de propagation (P) et de génération (G) pour chaque bit, puis en calculant les retenues pour chaque bit en fonction de ces signaux. Cela permet de réduire le nombre de dépendances de données et de réaliser des calculs en parallèle, ce qui peut conduire à une amélioration des performances, en particulier pour des opérations sur de grands nombres.

```

1  string addStrings(string num1, string num2)
2  {
3      // Convertir les chaînes d'entrée décimales en binaire
4      string bin1 = decimalToBinary(num1);
5      string bin2 = decimalToBinary(num2);
6
7      // Inverser les chaînes binaires pour faciliter le traitement
8      reverse(bin1.begin(), bin1.end());
9      reverse(bin2.begin(), bin2.end());
10
11     int size = max(bin1.size(), bin2.size()) + 1; // Taille du résultat, en tenant compte d'une éventuelle retenue
12
13     // Initialiser les vecteurs pour stocker les signaux
14     vector<int> G(size, 0);
15     vector<int> P(size, 0);
16     vector<int> C(size, 0);
17
18     // Générer les signaux G et P
19     generatePropagate(bin1, bin2, G, P);
20
21     // Générer les signaux de retenue
22     carryLookahead(P, C);
23
24     // Effectuer l'addition en utilisant les signaux de retenue
25     string result;
26     int carry = 0;
27     for (int i = 0; i < size - 1; i++)
28     {
29         int a = (i < bin1.size()) ? bin1[i] - '0' : 0;
30         int b = (i < bin2.size()) ? bin2[i] - '0' : 0;
31         int sum = a ^ b ^ carry; // Calcul de la somme
32         result += sum + '0'; // Ajout de la somme au résultat
33         carry = (G[i] || (P[i] && carry)) ? 1 : 0; // Calcul de la nouvelle retenue
34     }
35     if (carry)
36         result += '1'; // S'il y a une retenue à la fin
37
38     // Inverser la chaîne résultante avant de la retourner
39     reverse(result.begin(), result.end());
40     return result;
41 }

```

Dans le code que nous avons utilisé, l'algorithme Carry Look Ahead (CLA) est appliqué de la manière suivante :

### 1. decimalToBinary(const string &decimal) :

```

1  string decimalToBinary(const string &decimal)
2  {
3      string binary;
4      // Conversion de chaque chiffre décimal en binaire
5      for (char digit : decimal)
6      {
7          int num = digit - '0'; // Conversion du caractère en entier
8          string binaryDigit;
9          while (num > 0)
10         {
11             binaryDigit += to_string(num % 2); // Ajout du reste de la division par 2
12             num /= 2; // Division par 2
13         }
14         reverse(binaryDigit.begin(), binaryDigit.end()); // Inversion de la chaîne binaire
15         // Assurer que chaque chiffre binaire a une longueur fixe de 4 chiffres
16         while (binaryDigit.size() < 4)
17             binaryDigit = "0" + binaryDigit; // Ajout de zéros à gauche si nécessaire
18         binary += binaryDigit; // Concaténation des chiffres binaires
19     }
20     return binary; // Retourner la représentation binaire
21 }

```

- Cette fonction prend en entrée une chaîne de caractères représentant un nombre décimal.
- Elle parcourt chaque chiffre de la chaîne décimale, le convertit en binaire et l'ajoute à une chaîne binaire.
- Pour chaque chiffre décimal, elle divise le chiffre par 2 pour obtenir le bit binaire correspondant.
- Les chiffres binaires sont stockés dans une chaîne, avec des zéros ajoutés à gauche si nécessaire pour assurer une longueur fixe de 4 chiffres.
- Elle retourne la représentation binaire du nombre décimal en tant que chaîne de caractères.

## 2. `binaryToDecimal(const string &binary) :`

```

1  string binaryToDecimal(const string &binary)
2  {
3      string decimal;
4      // Conversion de chaque groupe de 4 chiffres binaires en décimal
5      for (size_t i = 0; i < binary.size(); i += 4)
6      {
7          string binaryDigit = binary.substr(i, 4);
8          int num = stoi(binaryDigit, nullptr, 2);
9          decimal += to_string(num);
10     }
11     return decimal;
12 }
```

- Cette fonction prend en entrée une chaîne de caractères représentant un nombre binaire.
- Elle parcourt la chaîne binaire par groupes de 4 chiffres.
- Chaque groupe de 4 chiffres binaires est converti en décimal à l'aide de la fonction **stoi()** avec une base de 2.
- Les chiffres décimaux sont concaténés pour former la représentation décimale du nombre binaire.
- Elle retourne la représentation décimale du nombre binaire en tant que chaîne de caractères.

## 3. `generatePropagate(const string &bin1, const string &bin2, vector<int> &G, vector<int> &P) :`

```

1  void generatePropagate(const string &bin1, const string &bin2, vector<int> &G, vector<int> &P)
2  {
3      int size = max(bin1.size(), bin2.size());
4      for (int i = 0; i < size; i++)
5      {
6          int a = (i < bin1.size()) ? bin1[i] - '0' : 0;
7          int b = (i < bin2.size()) ? bin2[i] - '0' : 0;
8          G[i] = a & b; // Générer le signal G
9          P[i] = a | b; // Générer le signal P
10     }
11 }
```

- Cette fonction génère les signaux G et P utilisés dans l'algorithme Carry Look Ahead (CLA) pour l'addition binaire.

- Elle prend en entrée deux chaînes binaires représentant les nombres à additionner, ainsi que deux vecteurs G et P pour stocker les signaux.
- Pour chaque position dans les deux nombres binaires, elle calcule les signaux G et P en effectuant des opérations logiques AND et OR entre les bits correspondants.
- Les signaux G et P sont stockés dans les vecteurs passés par référence.

#### 4. `carryLookahead(const vector<int> &P, vector<int> &C) :`

```

1  void carryLookahead(const vector<int> &P, vector<int> &C)
2  {
3      int size = P.size();
4      for (int i = 0; i < size; i++)
5      {
6          for (int j = i + 1; j < size; j++)
7          {
8              if (P[j])
9              {
10                 C[i] = 1; // Si le signal P est vrai, définir le signal C à 1
11                 break;
12             }
13         }
14     }
15 }

```

- Cette fonction effectue la propagation de la retenue à partir des signaux P générés précédemment.
- Elle prend en entrée un vecteur de signaux P et un vecteur de signaux de retenue C.
- Pour chaque position dans le vecteur P, elle examine les positions suivantes pour déterminer si un signal de retenue est nécessaire.
- Si un signal de retenue est nécessaire, elle définit le signal correspondant dans le vecteur C.



5. **addStrings(string num1, string num2) :**

```

1  string addStrings(string num1, string num2)
2  {
3      // Convertir les chaînes d'entrée décimales en binaire
4      string bin1 = decimalToBinary(num1);
5      string bin2 = decimalToBinary(num2);
6
7      // Inverser les chaînes binaires pour faciliter le traitement
8      reverse(bin1.begin(), bin1.end());
9      reverse(bin2.begin(), bin2.end());
10
11     int size = max(bin1.size(), bin2.size()) + 1; // Taille du résultat, en tenant compte d'une éventuelle retenue
12
13     // Initialiser les vecteurs pour stocker les signaux
14     vector<int> G(size, 0);
15     vector<int> P(size, 0);
16     vector<int> C(size, 0);
17
18     // Générer les signaux G et P
19     generatePropagate(bin1, bin2, G, P);
20
21     // Générer les signaux de retenue
22     carryLookahead(P, C);
23
24     // Effectuer l'addition en utilisant les signaux de retenue
25     string result;
26     int carry = 0;
27     for (int i = 0; i < size - 1; i++)
28     {
29         int a = (i < bin1.size()) ? bin1[i] - '0' : 0;
30         int b = (i < bin2.size()) ? bin2[i] - '0' : 0;
31         int sum = a ^ b ^ carry; // Calcul de la somme
32         result += sum + '0'; // Ajout de la somme au résultat
33         carry = (G[i] || (P[i] && carry)) ? 1 : 0; // Calcul de la nouvelle retenue
34     }
35     if (carry)
36         result += '1'; // S'il y a une retenue à la fin
37
38     // Inverser la chaîne résultante avant de la retourner
39     reverse(result.begin(), result.end());
40     return result;
41 }

```

- Cette fonction effectue l'addition binaire des deux nombres décimaux en utilisant l'algorithme Carry Look Ahead (CLA).
- Elle convertit d'abord les nombres décimaux en représentations binaires à l'aide de la fonction **decimalToBinary()**.
- Ensuite, elle génère les signaux G et P à l'aide de la fonction **generatePropagate()**.
- Elle effectue la propagation de la retenue à l'aide de la fonction **carryLookahead()**.
- Enfin, elle effectue l'addition en utilisant les signaux de retenue générés et retourne le résultat en tant que chaîne binaire.

## Parallélisation du problème utilisé avec l'Algorithme Carry Look Ahead en utilisant OpenMP:

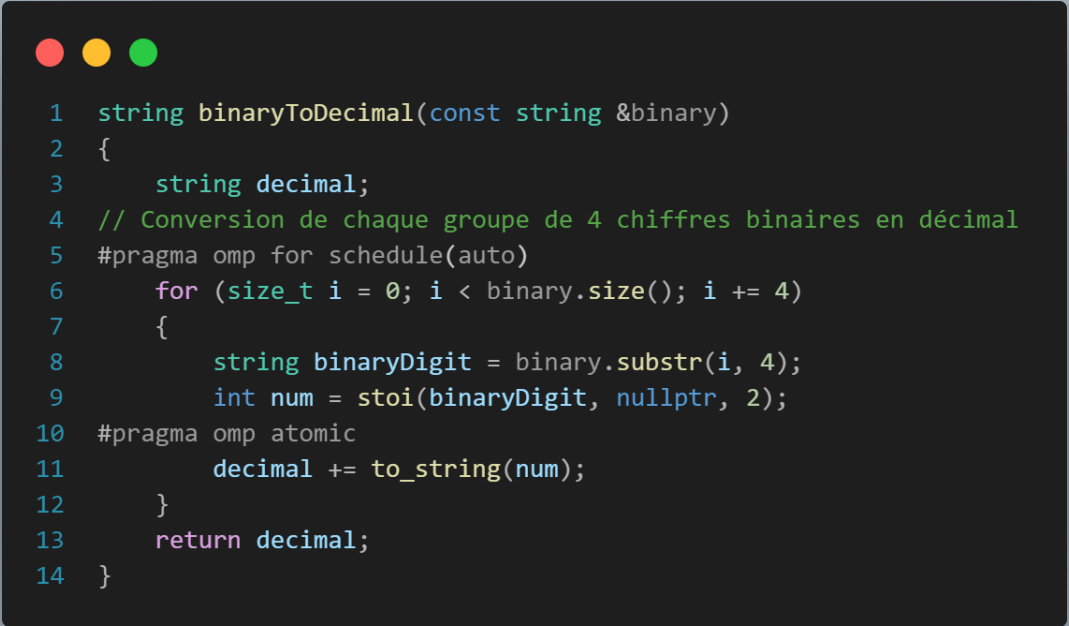
L'objectif principal de la parallélisation de l'algorithme Carry Look Ahead (CLA) avec OpenMP est d'exploiter le parallélisme offert par les architectures multi-cœurs pour accélérer le processus de calcul de la somme de grands entiers. En parallélisant les calculs impliqués dans l'algorithme CLA, nous visons à répartir la charge de calcul entre plusieurs threads, permettant ainsi une exécution plus rapide des opérations arithmétiques.

La parallélisation vise également à améliorer l'efficacité et l'utilisation des ressources disponibles en tirant parti des capacités de traitement parallèle des processeurs modernes. En divisant les tâches en sous-tâches exécutables en parallèle, nous cherchons à exploiter pleinement le potentiel de parallélisme pour obtenir des performances accrues lors du calcul de la somme de grands entiers.

### ‘decimalToBinary()’:

```

1  string decimalToBinary(const string &decimal)
2  {
3      string binary;
4      // Conversion de chaque chiffre décimal en binaire
5      for (char digit : decimal)
6      {
7          int num = digit - '0'; // Conversion du caractère en entier
8          string binaryDigit;
9          while (num > 0)
10         {
11             binaryDigit += to_string(num % 2); // Ajout du reste de la division par 2
12             num /= 2; // Division par 2
13         }
14         reverse(binaryDigit.begin(), binaryDigit.end()); // Inversion de la chaîne binaire
15         // Assurer que chaque chiffre binaire a une longueur fixe de 4 chiffres
16         while (binaryDigit.size() < 4)
17             binaryDigit = "0" + binaryDigit; // Ajout de zéros à gauche si nécessaire
18         binary += binaryDigit; // Concaténation des chiffres binaires
19     }
20     return binary; // Retourner la représentation binaire
21 }
```

**'binaryToDecimal ()':**


```

1  string binaryToDecimal(const string &binary)
2  {
3      string decimal;
4      // Conversion de chaque groupe de 4 chiffres binaires en décimal
5      #pragma omp for schedule(auto)
6      for (size_t i = 0; i < binary.size(); i += 4)
7      {
8          string binaryDigit = binary.substr(i, 4);
9          int num = stoi(binaryDigit, nullptr, 2);
10     #pragma omp atomic
11         decimal += to_string(num);
12     }
13     return decimal;
14 }

```

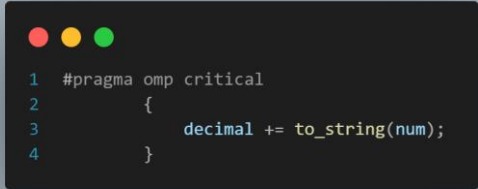


```

1  #pragma omp for schedule(auto)

```

1. **omp for:** Comme dans la fonction decimalToBinary, cette directive indique qu'une boucle suivra cette directive et que ses itérations doivent être exécutées en parallèle.
2. **schedule(auto):** L'ordonnancement est choisi automatiquement par le compilateur, généralement basé sur des heuristiques pour améliorer les performances.



```

1  #pragma omp critical
2  {
3      decimal += to_string(num);
4  }

```

3. **#pragma omp critical :** garantit que seule une thread à la fois peut exécuter le bloc de code à l'intérieur des accolades. Ainsi, chaque thread attend son tour pour accéder à la variable decimal, assurant que les modifications sont effectuées de manière ordonnée et cohérente.

## ‘generatePropagate ()’:

```

1 // Fonction pour générer les signaux G et P
2 void generatePropagate(const string &bin1, const string &bin2, vector<int> &G, vector<int> &P)
3 {
4     int size = max(bin1.size(), bin2.size());
5     #pragma omp parallel for shared(G, P) schedule(static)
6     for (int i = 0; i < size; i++)
7     {
8         int a = (i < bin1.size()) ? bin1[i] - '0' : 0;
9         int b = (i < bin2.size()) ? bin2[i] - '0' : 0;
10    #pragma omp task firstprivate(a, b)
11    {
12        G[i] = a & b; // Générer le signal G
13    }
14    #pragma omp task firstprivate(a, b)
15    {
16        P[i] = a | b; // Générer le signal P
17    }
18 }
19 }

```

```

1 #pragma omp parallel for shared(G, P) schedule(static)

```

1. **#pragma omp parallel for:** Cette directive crée une équipe de threads, chacun exécutant une copie de la boucle suivante de manière parallèle. Chaque thread traite une partie différente de l'itération de la boucle.
2. **shared(G, P):** Cette clause spécifie que les variables G et P sont partagées entre tous les threads. Cela signifie que chaque thread accèdera à la même instance de ces variables dans la mémoire.
3. **schedule(static):** Cette clause spécifie la stratégie d'ordonnancement statique pour répartir les itérations de la boucle entre les threads. Dans ce cas, chaque thread se voit attribuer un bloc d'itérations statiquement à l'avance.

```

1  #pragma omp task firstprivate(a, b)
2      {
3          G[i] = a & b; // Générer le signal G
4      }
5  #pragma omp task firstprivate(a, b)
6      {
7          P[i] = a | b; // Générer le signal P
8      }

```

1. **omp task:** Cette directive indique qu'une tâche doit être créée pour exécuter le bloc de code suivant. Chaque itération de la boucle génère deux tâches, une pour calculer le signal G et l'autre pour calculer le signal P.
2. **firstprivate(a, b):** Cette clause spécifie que les variables a et b doivent être privées à chaque tâche, mais leur valeur lors de la création de la tâche est copiée dans chaque tâche. Cela garantit que chaque tâche utilise les valeurs correctes de a et b lors de son exécution.

### ‘carryLookahead ()’:

```

1  void carryLookahead(const vector<int> &P, vector<int> &C)
2  {
3      int size = P.size();
4      #pragma omp parallel for shared(C) schedule(auto)
5          for (int i = 0; i < size; i++)
6          {
7              for (int j = i + 1; j < size; j++)
8              {
9                  if (P[j])
10                 {
11                     #pragma omp critical
12                     {
13                         C[i] = 1; // Si le signal P est vrai, définir le signal C à 1
14                     }
15                     break;
16                 }
17             }
18         }
19     }

```



```
1 #pragma omp parallel for shared(C) schedule(auto)
```

1. **omp parallel for:** Cette directive crée une région parallèle avec une boucle suivante qui doit être exécutée en parallèle. Chaque itération de la boucle sera exécutée par un thread différent.
2. **shared(C):** Indique que la variable C est partagée entre tous les threads. Cela signifie que chaque thread aura accès en lecture et en écriture à la variable C.
3. **schedule(auto):** Cette clause permet au compilateur de choisir automatiquement l'ordonnancement à utiliser pour la boucle. Le compilateur choisira généralement l'ordonnancement le plus approprié en fonction du contexte et des performances.



```
1 #pragma omp critical
2     {
3         C[i] = 1; // Si le signal P est vrai, définir le signal C à 1
4     }
```

4. **omp critical:** Cette directive crée une section critique, c'est-à-dire une section de code où l'accès à une ressource partagée est contrôlé de manière à ce qu'un seul thread puisse y accéder à la fois. Dans ce cas, la variable C est partagée entre les threads, et chaque thread qui entre dans cette section critique a le droit d'écrire dans C seulement après avoir acquis le verrou. Cela garantit que les mises à jour de C sont correctement synchronisées entre les threads pour éviter des résultats incorrects dus à des conditions de concurrence.

**‘addStrings ()’:**

```

1  string addStrings(string num1, string num2)
2  {
3      // Convertir les chaînes d'entrée décimales en binaire
4      string bin1 = decimalToBinary(num1);
5      string bin2 = decimalToBinary(num2);
6
7      // Inverser les chaînes binaires pour faciliter le traitement
8      reverse(bin1.begin(), bin1.end());
9      reverse(bin2.begin(), bin2.end());
10
11     int size = max(bin1.size(), bin2.size()) + 1; // Taille du résultat, en tenant compte d'une éventuelle retenue
12
13     // Initialiser les vecteurs pour stocker les signaux
14     vector<int> G(size, 0);
15     vector<int> P(size, 0);
16     vector<int> C(size, 0);
17
18     // Générer les signaux G et P en parallèle
19     #pragma omp parallel sections
20     {
21         #pragma omp section
22         generatePropagate(bin1, bin2, G, P);
23         #pragma omp section
24         carryLookahead(P, C);
25     }
26
27     // Effectuer l'addition en utilisant les signaux de retenue
28     string result;
29     int carry = 0;
30     #pragma omp parallel
31     {
32         #pragma omp single
33         for (int i = 0; i < size - 1; i++)
34         {
35
36             int a = (i < bin1.size()) ? bin1[i] - '0' : 0;
37             int b = (i < bin2.size()) ? bin2[i] - '0' : 0;
38             int sum = a ^ b ^ carry; // Calcul de la somme
39             result += sum + '0'; // Ajout de la somme au résultat
40             carry = (G[i] || (P[i] && carry)) ? 1 : 0; // Calcul de la nouvelle retenue
41         }
42         if (carry)
43             result += '1'; // S'il y a une retenue à la fin
44     }
45
46     // Inverser la chaîne résultante avant de la retourner
47     #pragma omp single
48     reverse(result.begin(), result.end());
49     return result;
50 }

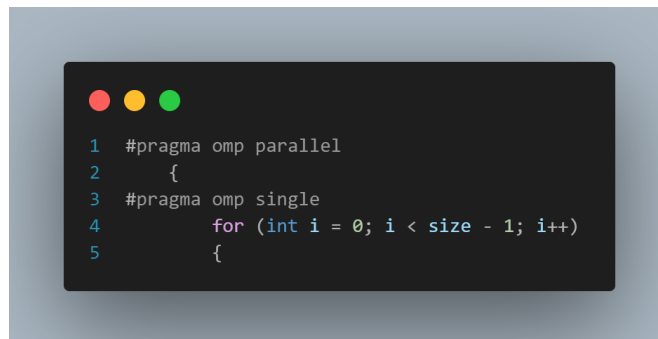
```

```

1  #pragma omp parallel sections
2  {
3      #pragma omp section
4      generatePropagate(bin1, bin2, G, P);
5      #pragma omp section
6      carryLookahead(P, C);
7  }

```

1. **omp parallel sections:** Cette directive crée une région parallèle où différentes sections peuvent être exécutées en parallèle. Les sections sont délimitées par les directives **omp section**. Dans ce cas, deux sections parallèles sont définies, l'une pour générer les signaux G et P (**generatePropagate**) et l'autre pour effectuer la propagation de la retenue (**carryLookahead**).
2. **omp section:** Cette directive indique une section de code qui sera exécutée par l'un des threads de la région parallèle. Dans ce cas, une section est utilisée pour appeler la fonction **generatePropagate** et l'autre pour appeler la fonction **carryLookahead**. Chaque section peut être exécutée par un thread différent.

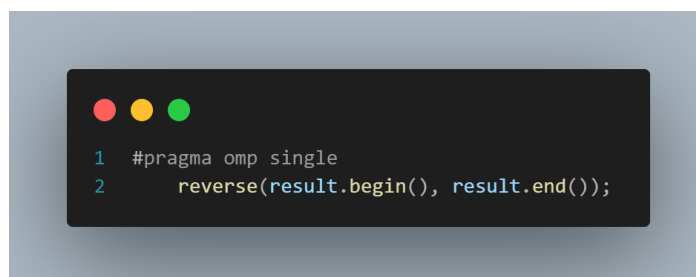


```

1  #pragma omp parallel
2  {
3  #pragma omp single
4      for (int i = 0; i < size - 1; i++)
5      {

```

3. **#pragma omp parallel** : Crée une région parallèle où plusieurs threads sont activés pour exécuter le bloc de code qui suit.
4. **#pragma omp single** : Indique qu'un seul thread exécutera le bloc de code suivant. Les autres threads resteront en attente jusqu'à ce que le thread unique ait terminé son exécution.



```

1  #pragma omp single
2      reverse(result.begin(), result.end());

```

7. **omp single:** Cette directive indique qu'un seul thread exécutera le bloc de code suivant. Dans ce cas, la fonction **reverse** est appelée une seule fois pour inverser la chaîne de résultat. Cela garantit que l'inversion de la chaîne n'est effectuée qu'une seule fois, par un seul thread, ce qui évite les conflits de données.



## **Conclusion :**

En conclusion, ce projet a réussi à paralléliser efficacement le calcul de la somme de deux grands entiers en utilisant l'algorithme Carry Look Ahead (CLA) avec OpenMP. En exploitant le parallélisme des architectures multi-cœurs, nous avons accéléré le processus de calcul tout en maintenant la cohérence des résultats. Cette approche offre une solution efficace pour des calculs intensifs, démontrant l'importance de la parallélisation dans l'optimisation des performances des applications sur des plateformes matérielles modernes.

## **Sources :**

<https://vlsiverify.com/verilog/verilog-codes/carry-look-ahead-adder/>

<https://www.youtube.com/watch?v=yj6wo5SCObY>

<https://www.geeksforgeeks.org/c-program-to-implement-full-adder/>