École Nationale Supérieure d'Ingénieurs de Caen

6, Boulevard Maréchal Juin
F-14050 Caen Cedex, FRANCE

TP $n^o 3$

Niveau	**2ème année**
Parcours	**Informatique**
Unité d'enseignement	**2I2AC3 - Architectures parallèles**
Responsable	**Emmanuel Cagniot** **Emmanuel.Cagniot@ensicaen.fr**

1 Problème

Le problème des huit reines consiste à disposer huit reines (dames) sur un échiquier sans qu'aucune ne soit en prise avec les autres. Une reine peut se déplacer dans toutes les directions d'un nombre variable de positions (figure 1). Par conséquent, deux reines ne peuvent partager une même ligne, colonne, diagonale ou anti-diagonale. Ce problème possède 92 solutions mais seulement 12 solutions originales, c'est à dire que les $92 - 12 = 80$ autres solutions s'obtiennent par symétrie et rotation. Le problème des reines est généralisable à un échiquier de taille $n \times n$ mais ne possède pas de solution si $n \leq 3$.

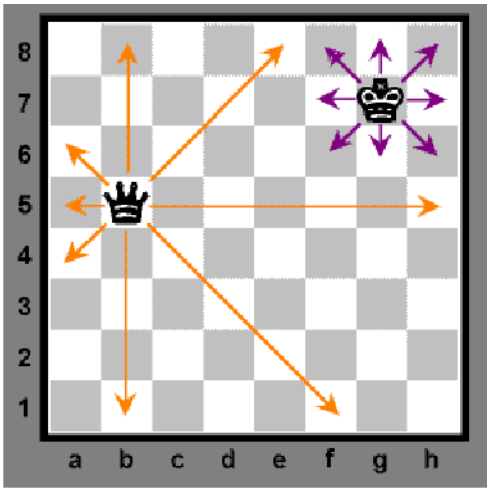

FIGURE 1 – Mouvements de la reine (à gauche) sur un échiquier.

Plusieurs stratégies de résolution peuvent être envisagées. Les stratégies exhaustives calculent toutes les solutions du problème : par conséquent un échec permet de garantir que le problème à résoudre ne possède

pas de solution. Cependant, les temps de calcul croissent exponentiellement avec la taille du problème à résoudre. Inversement, les stratégies non-exhaustives utilisent des heuristiques pour ne pas explorer tout l'ensemble des configurations du problème : de fait les temps de calculs requis par ces stratégie sont tout à fait acceptables ; cependant, comme leur convergence n'est pas assurée, un échec ne garantit pas que le problème à résoudre ne possède pas de solution.

Un algorithme exhaustif simpliste consiste a exploiter la récursivité. Dans cet algorithme un échiquier de taille $n \times n$ est représenté par un tableau d'entiers `chessboard` de taille n. Les indices de ce tableau représentent les numéros de ligne de l'échiquier tandis que ses entrées représentent les numéros de colonnes sur lesquelles sont posées les reines. Ainsi, pour un échiquier de taille 4×4 dont la numérotation des lignes et colonnes commence à partir de zéro, un tableau tel que `chessboard = [ 2, 0, 3, 1 ]` signifie que les reines occupent respectivement les cases $(0, 2)$, $(1, 0)$, $(2, 3)$ et $(3, 1)$.

Le tableau `chessboard` est rempli récursivement. Si i désigne le numéro de ligne sur laquelle nous essayons de poser une nouvelle reine alors nous cherchons un numéro de colonne j tel qu'une reine posée sur la case (i, j) ne soit pas en prise avec celles déjà posées sur les lignes $i' < i$. Savoir si la case (i, j) est menacée par d'autres reines déjà posées est relativement simple. En effet, soit (i', j') la position d'une reine déjà posée $(i' < i)$. Cette reine menace la case (i, j) si l'une des deux conditions suivantes est vérifiée :

1. $j' = j$, qui signifie que les deux reines partagent la même colonne ;
2. $| i - i' | = | j - j' |$, qui signifie que la pente de la droite définie par les deux points (i, j) et (i', j') vaut 1 ou -1. Dans le premier cas les deux reines partagent la même diagonale tandis qu'elles partagent la même anti-diagonale dans le second.

Si nous parvenons à placer une nouvelle reine dans la case (i, j) alors nous relançons l'algorithme sur la ligne $i + 1$. Ce dernier s'arrête lorsque $i = n - 1$ (nous tenons une nouvelle solution) ou lorsqu'aucune case (i, j) ne convient (nous sommes dans une impasse). Comme nous cherchons toutes les solutions du problème, nous relançons l'algorithme sur la ligne $i + 1$ pour toute case (i, j) pouvant accueillir une nouvelle reine. Par conséquent cet algorithme n'est au final qu'un parcours d'arbre n-aire en profondeur d'abord.

Les figures 2, 3 et 4 présentent les trois fonctions de cet algorithme, `solve_n_queens` étant son point d'entrée.

Nous nous intéressons à différentes techniques de parallélisation de l'algorithme ci-dessus en OpenMP. L'archive `tp3.tar.gz`, accessible via la page web décrivant l'unité d'enseignement, contient un squelette incomplet de l'application à réaliser. Sa structure est la suivante :

`src/include/ :` contient les déclarations des différentes versions de notre algorithme ;

`src/ :` contient les définitions des différentes versions de notre algorithme ainsi que le programme principal `bench.c` permettant d'évaluer leurs performances. Ce programme vérifie que le résultat produit par une version parallèle est identique à celui produit par la version séquentielle. Si tel n'est pas le cas alors il interrompt son exécution en écrivant une message sur sa sortie erreur ;

`CMakeLists.txt :` script permettant de générer le makefile de du programme principal via l'utilitaire `cmake` ;

`Lisezmoi.txt :` fichier texte décrivant la procédure de génération du makefile et celle de la configuration du fichier `CMakeCache.txt` produit par `cmake`. Une fois ce fichier modifié, le compilateur est autorisé à exploiter sa bibliothèque OpenMP (option `-fopenmp`), les caractéristiques de votre processeur (option `-march=native`) et toutes les optimisations possibles (option `-O3`).

La fonction `solve_n_queens` fait usage d'une boucle `for` pour calculer l'arbre des solutions au départ de chaque case de la première ligne. L'indice de cette boucle et de type entier et la condition de continuation exploite l'opérateur « strictement inférieur à » : par conséquent cette boucle est totalement parallélisable en OpenMP (approche SPMD du problème).

```
/**
 * Implémentation récursive et séquentielle du problème des n-reines.
 *
 * @param[in] dim la dimension du problème.
 * @return le nombre de solutions du problème.
 */
unsigned long solve_n_queens(const int dim);

unsigned long
solve_n_queens(const int dim) {

  // Nombre de solutions.
  unsigned long how_many = 0;

  // Echiquier.
  int chessboard[dim];

  // Nous comptabilisons le nombre de solutions trouvée au départ de chaque
  // case de la première ligne.
  for (int c = 0; c < dim; c ++) {
    how_many += put_queen(chessboard, dim, 0, c);
  }

  // C'est terminé.
  return how_many;

}
```

FIGURE 2 – Fonction `solve_n_queens`.

```
/**
 * Retourne le nombre de solutions au départ de la case (lin, col) d'un
 * échiquier.
 *
 * @param[in, out] chessboard l'échiquier.
 * @param[in] dim la dimension de l'échiquier.
 * @param[in] lin la ligne.
 * @param[in] la colonne.
 * @return le nombre de solutions au départ de la case.
 */
unsigned long put_queen(int chessboard[],
                        const int dim,
                        const int lin,
                        const int col);

unsigned long
put_queen(int chessboard[], const int dim, const int lin, const int col) {

  // Impossible de poser une dame à cette position : l'exploration de la
  // branche s'arrête.
  if (! can_put_queen(chessboard, lin, col)) {
    return 0;
  }

  // La position n'étant pas menacée nous la conservons.
  chessboard[lin] = col;

  // La dernière ligne venant d'être traitée nous tenons une nouvelle
  // solution.
  if (lin == dim - 1) {
    return 1;
  }

  // Dans le cas contraire il faut traiter la ligne suivante.
  unsigned long how_many = 0;
  for (int c = 0; c < dim; c ++) {
    how_many += put_queen(chessboard, dim, lin + 1, c);
  }

  // C'est terminé.
  return how_many;

}
```

FIGURE 3 – Fonction `put_queen`.

```
/**
 * Indique si une dame peut être posée à la position (lin, col) d'un
*  échiquier.
 *
 * @param[in] chessboard l'échiquier.
 * @param[in] lin la ligne.
 * @param[in] la colonne.
 * @return @c true si une dame peut peut être posée à cette position.
 */
bool can_put_queen(const int chessboard[], const int lin, const int col);

bool
can_put_queen(const int chessboard[], const int lin, const int col) {
  for (int l = 0; l < lin; l ++) {
    if (chessboard[l] == col || ABS(chessboard[l] - col) == lin - l) {
      return false;
    }
  }
  return true;
}
```

FIGURE 4 – Fonction `can_put_queen`.

1.1 Question

Complétez la définition de la fonction `omp_solve_n_queens_v1` qui met en œuvre l'approche SPMD décrite ci-dessus (vous prendrez soin de justifier la clause de répartition des itérations utilisée).

L'approche SPMD précédente est évidemment la plus simple à mettre en œuvre et présente une très bonne efficacité. Cette approche n'est toutefois pas la seule car il est également possible de travailler sur la fonction `put_queen`. Cependant la difficulté est ici beaucoup plus importante car il s'agit d'une fonction récursive.

En observant la définition de cette fonction, nous constatons qu'elle possède en structure similaire à `omp_solve_n_queens_v1` au niveau de sa boucle `for`. La tentation est alors grande de reproduire ce que nous avons déjà fait avec `omp_solve_n_queens_v1`. Cependant, la situation est ici très différente du fait de l'aspect récursif de la fonction. En effet lorsqu'une région parallèle est rencontrée lors de l'exécution d'un code, il y a création dynamique de threads dans la limite fixée par la variable d'environnement `OMP_NUM_THREADS` (la valeur par défaut est tout simplement le nombre de cœurs du processeur). Ces threads sont détruits en sortie de région. Or les opérations de création et de destructions de threads sont des appels systèmes extrêmement coûteux en termes de cycles processeur. Par conséquent, pour qu'une parallélisation soit efficace il faut que le coup de création et de destruction d'un thread soit absorbé par la charge de travail confiée à ce thread.

Dans le cas d'une fonction récursive que se passe t-il lorsqu'un thread rencontre une région parallèle ? Y'a t-il à nouveau création puis destruction de nouveaux threads ? En fait ce cas de figure est appelé « parallélisme imbriqué ». Par défaut ce type de parallélisme est inhibé c'est à dire qu'il n'y a pas de création de nouveaux threads lorsqu'un thread rencontre une région parallèle : le code de la région est tout simplement exécuté par le thread l'ayant rencontrée. Il est toutefois possible d'activer le parallélisme imbriqué via la variable d'environnement booléenne `OMP_NESTED`. Dans ce cas, il y a création et destructions de nouveaux threads à chaque fois qu'un thread rencontre une région parallèle et ce dans la limite fixée par l'implémentation. En règle générale ce type de parallélisme est à proscrire car il peut conduire à

la création d'un nombre exponentiel de threads. Il est cependant possible de limiter ce nombre au travers d'une autre variable d'environnement de type entier : `OMP_THREAD_LIMIT`.

Est-il possible de mettre en œuvre le parallélisme imbriqué dans le cas qui nous intéresse ? En observant le code de la fonction `put_queen` nous pouvons voir que l'opération la plus lourde est l'appel de la fonction `can_put_queen`. Si nous profilons notre code nous constatons que l'essentiel du temps correspond en fait au calcul de la valeur absolue (`ABS`). Or, cette opération est relativement légère comparée au coût de création d'un thread. D'autre part, tout comme pour la fonction `omp_solve_n_queens_v1`, chaque thread doit disposer de sa propre copie locale du tableau `chessboard` ce qui signifie qu'à chaque récursion il faudra allouer un nouveau tableau (un autre appel système) afin d'y recopier le contenu du paramètre `chessboard`. Là encore il s'agit d'une opération très coûteuse. En conséquence, envisager une approche SPMD couplée au parallélisme imbriqué pour la fonction `put_queen` n'est vraiment pas judicieux. Envisager cette même approche sans parallélisme imbriqué revient tout simplement à refaire ce que nous avons déjà fait dans la fonction `omp_solve_n_queens_v1` mais avec le coup création/destruction des threads à chaque récursion : là aussi l'idée est tout sauf judicieuse.

L'approche SPMD de la fonction `put_queen` étant abandonnée, nous nous tournons vers une approche de type MIMD. En OpenMP, deux mécanismes permettent de la mettre en œuvre :
— les sections parallèles (historiquement le premier mécanisme) ;
— les tâches (depuis la norme 3.0).

Commençons par les sections parallèles. L'idée est ici de confier l'exécution de travaux différents à un groupe de threads, le nombre de sections fixant le nombre de threads qui travaillent. Dans le cas d'une fonction récursive telle que `put_queen`, l'idée est répartir le traitement des colonnes de la ligne courante entre les threads disponibles : en ce sens il s'agit ni plus ni moins que de reproduire une approche SPMD mais sans boucle `for`. Cependant, le mécanisme des sections parallèles est extrêmement rigide car il nous oblige à écrire (en dur dans le code) autant de sections que de travaux à effectuer. Or, le nombre de threads disponibles peut varier d'un processeur à l'autre ou d'une valeur de la variable `OMP_NUM_THREADS` à une autre. Pour résoudre ce problème il nous faut activer le parallélisme imbriqué, l'idée étant d'exploiter progressivement l'ensemble des threads disponibles. Dans notre cas il n'est pas possible de jouer avec la variable `OMP_THREAD_LIMIT` car chaque section parallèle que nous écrirons aura besoin d'une copie locale du tableau `chessboard`. Par conséquent, les opérations d'allocation et de recopie (très coûteuses) associées ne doivent pas perdurer lorsque le nombre de threads maximum est atteint. Il y a donc deux comportements à coder en dur dans notre notre version parallèle de `put_queen` :
— créer des sections parallèles avec leur copie locale de `chessboard` à chaque récursion tant que le nombre de threads maximum n'est pas atteint ;
— travailler directement sur le paramètre `chessboard` lorsque le nombre maximum de threads est atteint.

1.2 Question

Complétez la définition de la fonction `omp_put_queen_v2`. Son paramètre `threads` indique le nombre de threads encore disponibles. Si ce nombre est supérieur à 1 alors deux sections parallèles sont écrites : la première section traite la première moitié des colonnes de la ligne courante tandis que l'autre fait de même pour la seconde moitié. Lorsqu'elles invoquent à nouveau la fonction `omp_put_queen_v2` la valeur transmise pour le paramètre `threads` est la moitié de la valeur actuelle (n'oubliez pas que la variable locale `how_many` doit faire l'objet d'une réduction). Activez ensuite le parallélisme imbriqué (`export OMP_NESTED=true`) avant de relancer l'exécutable `bench`.

Bien que le parallélisme imbriqué permette de compenser la rigidité des sections parallèles dans le cas récursif, il s'agit tout de même d'un mécanisme lourd à mettre en œuvre. Le mécanisme de tâches, introduit par la norme 3.0 permet d'adresser ce problème. Une tâche n'est rien d'autre qu'un travail à exécuter « plus tard ». Pour pouvoir l'être, les instructions associées à ce travail doivent disposer de toutes les données sur lesquelles elle vont d'appliquer (principe du behavioral design pattern `Command` en conception

par objets).

Les tâches sont attribuées aux threads qui vont les exécuter par un mécanisme appelé scheduler. Il existe plusieurs types de schedulers et si l'utilisateur n'est pas satisfait de celui proposé par son implémentation alors il est libre de le remplacer par un autre. C'est le grand avantage des tâches par rapport aux autres mécanismes proposés par OpenMP (sections parallèles, boucles parallèles, etc.) : ce n'est pas l'utilisateur qui code en dur l'ordonnancement de ses threads. La contrepartie de cette souplesse est une écriture beaucoup plus délicate.

À toute région parallèle est associée une file de tâches. En règle générale, seul un thread de la région approvisionne cette file tandis que les autres attendent. Une fois la file complétée, tous les threads de la région exécutent les tâches de la file jusqu'à épuisement de cette dernière. Lors de son exécution, une tâche peut demander la création d'autres tâches qui sont à leur tour insérées dans la file. Leur position d'insertion dépend de leur dépendance avec celle ayant demandée leur création. Par exemple supposons qu'une tâche `A` en cours d'exécution demande la création d'une tâche `B`. Si `A` ne dépend pas de l'exécution de `B` alors `B` est insérée dans la file pendant que `A` poursuit son exécution. Si, au contraire, `A` doit attendre la fin de l'exécution de `B` alors `A` est remise dans la file (dans son état actuel) derrière `B` : c'est donc `B` qui sera sortie avant `A` par le scheduler.

Dans le cas qui nous intéresse, nous allons mettre en œuvre l'approche `MIMD` adoptée précédemment avec les sections parallèles (les tâches remplacement les sections).

1.3 Question

Complétez la définition de la fonction `omp_solve_n_queens_v3` en ouvrant une région parallèle permettant à un seul thread de créer les tâches initiales de la file (appel de la fonction `omp_put_queen_v3` ; n'oubliez pas la réduction sur la variable locale `how_many`).

1.4 Question

Complétez la définition de la fonction `omp_put_queen_v3` chargée de la création récursive des tâches initiales de la file dans la limite du nombre de threads disponibles. Ces tâches doivent mettre à jour la variable locale `how_many`. Comme la clause `reduction` n'a pas de signification pour une tâche, cette variable est indiquée comme partagée et ne peut être mise à jour que via la directive `atomic`.

Bien que la fonction `omp_put_queen_v3` que vous venez d'écrire soit très satisfaisante, nous souhaitons à présent exploiter certains mécanismes avancés liés aux tâches et introduits par la dernière norme d'OpenMP : les groupes de tâches et les dépendances entre tâches (jetez un coup d'oeil sur le support écrit de TP pages $17 - 19$).

1.5 Question

La version actuelle de notre fonction fait usage de la directive de synchronisation explicite `taskwait` qui permet à une tâche mère de n'achever son exécution que lors ses filles ont fait de même. Modifiez la de telle manière que :

- l'usage de la directive `taskwait` est banni ;
- les deux tâches utilisées pour gérer les deux moitiés de la ligne courante `lin` ne mettent plus à jour la variable `how_many` ;
- une troisième tâche collecte les résultats des deux tâches précédentes pour mettre à jour la variable `how_many`.