Efficacité d'un programme séquentiel

L'objectif est de mettre en œuvre des techniques d'optimisation pour les programmes séquentiels.

1 Fusion de boucle

Observez le programme `boucles.c`. Quelle optimisation auriez-vous naturellement tendance à faire ? Le gain obtenu est-il décevant, correct, ou plus que satisfaisant ? Comment l'expliquer ?

Reproduire l'expérience en ajoutant une troisième puis une quatrième boucle, obtenez-vous une accélération encore meilleure ? Pourquoi ?

2 Déroulement de boucle

Tout d'abord, observons le coût d'une boucle : le programme `deroulement.c` effectue un calcul tout bête au sein d'une boucle qui a un nombre d'itérations connu. Tel quel, chaque mesure prend quelques secondes.

Déroulez la boucle, c'est-à-dire par exemple définir `D` à 2 pour faire 2 fois moins d'itérations, mais en répliquant le contenu de la boucle pour lui faire faire deux fois le calcul par itération. Attention aux indices de tableau (*vérifiez* que le résultat obtenu est bien le même). Essayez en déroulant 2 itérations, 4 et 8.

Essayez d'optimiser le cœur de la boucle en utilisant des variables supplémentaires ou ne faisant qu'une seule affectation par tour.

Est-il intéressant de dérouler indéfiniment ? Quelle combinaison d'options de gcc permet de dérouler les boucles ?

3 Multiplication et somme de matrices

Les programmes `mul_mat.c` et `som_mat.c` effectuent sommes et multiplications de matrices de façon très basique. Cette façon de faire est loin d'être optimale.

1. Lancer plusieurs fois le programme `som_mat` pour vérifier que les procédures `somMat` et `somMat2` ont un temps d'exécution similaire.
2. Modifier la procédure `somMat2` en permutant l'ordre des boucles sur i et sur j. Mesurer l'accélération obtenue.

   Cette accélération est due à une meilleure exploitation de la mémoire cache qui, grossièrement, charge dans sa mémoire, non seulement la valeur de la case mémoire demandée par le processeur, mais aussi celles de ses cases voisines.
3. A votre avis, quelle case de `t[i][j+1]` ou de `t[i+1][j]` est rangée en mémoire à coté de `t[i][j]` ? Est-ce vraiment toujours le cas ?

4. Recommencer l'expérience en faisant varier la taille de la matrice (prendre N = 1024, N = 256, N = 64). Expliquer les résultats obtenus en vous aidant de la commande lstopo pour connaitre la taille des caches.

On s'intéresse maintenant au produit de matrices.

1. Modifier le code de `prodMat2` afin d'utiliser plus efficacement le cache du processeur. Le gain obtenu est-il décevant, correct ou plus que satisfaisant ?
2. Il est probable que quelques défauts de cache évitables subsistent dans votre code. Les repérez-vous ? Quelle permutation des boucles sur i, j, k induit le plus petit nombre de défauts de cache ? Modifier votre code en conséquence.
3. Lorsque N est assez grand il est probable quelques défauts de cache évitables subsistent dans votre code. Supposons que le cache fasse 8 Mo pour quelle valeur de N apparaissent ces défauts de cache ?

Transposée d'une matrice

L'objectif est de calculer la transposée d'une matrice (dans notre cas, ce sera une image) le plus rapidement possible. Une version séquentielle simple vous est fournie dans le fichier transpose.c : transpose_compute_seq.

Pour l'essayer :

./prog -k transpose -l images/shibuya.png -v seq -d p

(l'option "-d p" permet d'apprécier le résultat à chaque itération)

Étape 1 : parallélisation OpenMP simple

Recopiez transpose_compute_seq dans une fonction transpose_compute_omp et ajoutez des directives de parallélisation de boucles OpenMP. Mesurez les accélérations obtenues.

Étape 2 : parallélisation OpenMP tuilée

Examinez la version « tuilée » de la transposée (fonction transpose_compute_tiled). Proposez une version OpenMP parallèle (appelée omp_tiled) en distribuant les tuiles sur les processeurs disponibles (chaque tuile reste exécutée séquentiellement).

Étape 3 : parallélisation OpenMP avec des tâches

Utilisez maintenant les tâches OpenMP pour paralléliser la version tuilée. L'idée est de créer une tuile par tâche. Comparez cette version (omp_task) avec la version précédente (omp_tiled).

Faites varier la constante GRAIN pour chercher à obtenir la meilleure accélération.

Recherche des composantes connexes dans une image en OpenMP

Introduction au problème

On souhaite identifier les objets blancs présents sur une image en noir et blanc. On travaille ici en 4-connexité : deux pixels sont connexes s'ils sont adjacents par un bord (nord, sud, est ou ouest). Deux pixels blancs appartiennent au même objet s'ils sont connexes ou s'il existe une chaine de pixels connexes blancs les reliant. Dans le tableau ci-contre les pixels contenant 1 et ceux contenant 2 forment deux objets 4-connexes distincts.

i \ J	0	1	2	3	4
0	0	1	0	0	2
1	0	1	0	2	2
2	0	1	1	0	0
3	0	0	1	0	0
4	0	0	0	0	0

Pour identifier les objets on commence par attribuer l'entier 0 aux pixels noirs et un entier distinct à chaque pixel blanc. Ensuite, on propage l'identité maximale dans le voisinage des pixels blancs en itérant le calcul.

Au bout d'un nombre d'itérations (borné par le nombre de pixels) les pixels appartenant au même objet ont tous acquis la même identité, celle du pixel de plus grande identité. La propagation stagne, le calcul est terminé.

	0	1	2	3	4
0		8			5
1		8		5	5
2		8	8		
3			8		
4					

Pour réaliser cette propagation, il est naturel d'itérer jusqu'à stagnation un balayage de l'ensemble des pixels en attribuant à chaque pixel blanc (non nul) l'identité maximale entre celle du pixel considéré et celles des quatre pixels voisins. L'efficacité d'un tel algorithme basé sur une succession de parcours classiques (`for (i=0 ; i < DIM; i++) for (j=0 ; j < DIM; j++)` *...) de l'ensemble des pixels peut être sensiblement amélioré en remplaçant le balayage ordinaire par deux balayages :*

- *le premier balaye les pixels dans le sens habituel :*

```
for (i=0; i < DIM; i++)
    for (j=0; j < DIM; j++) …
```

- *le second dans le sens inverse :*

```
for (i=DIM-1; i >=0 ;i--)
    for (j=DIM-1; j >=0 ;j--)…
```

Il apparaît alors peu utile de calculer le maximum sur l'ensemble des 4 pixels voisins mais simplement sur ceux favorisant le plus la propagation du max.

Par convention posons que le sens descendant corresponde au parcours du tableau de gauche à droite puis de haut en bas ; pour faire descendre le max il suffit de comparer les identités de la cellule considérée à celles des cellules ouest et nord. Le sens montant correspond au parcours du tableau de droite à gauche puis de bas en haut, on fait remonter le max en consultant les cellules sud et est. Cet algorithme est implémenté par la fonction max_compute_seq *(dans* max.c*).*

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

Sens descendant

7	6	5	4
6	5	4	3
5	4	3	2
4	3	2	1

Sens montant

Notons que la propagation peut être effectuée en parallèle sans grande précaution car l'ordre des calculs importe peu pourvu qu'on arrive à la stagnation.

Étape 1 : OpenMP FOR

Parallélisez le premier code donné à l'aide de boucles OpenMP sans forcément respecter les dépendances de données induites par le code séquentiel : l'ordre des calculs importe peu pourvu qu'on arrive à la stagnation. Ici, il s'agit de paralléliser les fonctions descendre_max et monter_max de la façon la plus simple possible (ajout de deux directives).

- Vérifiez que le code le bon fonctionnement du code avec l'option –l images/spirale.png
- Mesurez les performances obtenues pour DIM = 2048 avec des spirales de 100 tours (options -s 2048 -p 100)

Expérimentalement on s'aperçoit que si l'on se contente de paralléliser les boucles for, la version parallèle effectue significativement plus d'itérations que la version séquentielle en présence de « gros » objets. En effet, admettons que tous les pixels soient blancs alors la version parallèle nécessitera autant d'itérations qu'il y a de threads pour remonter le max au lieu d'une seule pour la version séquentielle.

Une piste pour améliorer l'efficacité de la parallélisation est de sacrifier un peu de parallélisme pour conserver la bonne transmission du max. Sur le schéma ci-contre chaque case représente une « tuile » pixels. Le contenu de chaque tuile est traité de façon séquentielle. À la première étape on traite la tuile marquée 1, une fois traitée on peut traiter celles marquées 2 et ainsi de suite. De façon générale on peut traiter une tuile dans le sens descendant (resp. montant) après que ses voisines nord et ouest (resp. sud et est) ont été traitées.

	N	
O	*C*	

Descendant

	C	*E*
	S	

Montant

La version « tuilée » de cet algorithme est implémentée par la fonction max_compute_tiled *(dans* compute.c*), le nombre de tuiles étant fixé par la constante GRAIN (il y a GRAIN² tuiles).*

Étape 2 : Tâches OpenMP

Parallélisez le code de la version tuilée en faisant en sorte que chaque macro cellule soit réalisée par une tâche OpenMP. On utilisera alors la clause depend pour contraindre l'ordre d'exécution des tâches afin de

conserver l'ordre séquentiel de la propagation du max. Pour simplifier l'expression des dépendances on utilisera les tableaux cellulem[GRAIN][GRAIN] et celluled[GRAIN][GRAIN].

- À nouveau, vérifiez que le code le bon fonctionnement du code avec l'option –l images/spirale.png
- Mesurez les performances obtenues pour DIM = 2048 avec des spirales de 100 tours.
- Calculez une borne théorique maximale en fonction du GRAIN et en supposant que l'on dispose d'un nombre non borné de processeurs.
- Mesurez les accélérations obtenues par rapport à la version séquentielle en faisant varier la taille du grain.

Étape 3 : Placement des tâches sur machine NUMA

Les machines du CREMI (salle 203 par exemple) possèdent une architecture à accès mémoire non uniforme (NUMA). Il est possible de vérifier le nombre de bancs mémoires à l'aide de la commande lstopo. On s'intéresse désormais à l'influence du placement des tâches dans notre programme sur le temps d'exécution. Il s'agit de comparer les performances obtenues par la parallélisation à l'aide des tâches OpenMP à celles obtenues par un ordonnanceur ad hoc (max_compute_sched) qui permet de placer explicitement les tâches sur les différents cœurs. En particulier, le cœur choisi pour la tâche en charge de la macro-cellule (i, j) est choisi dans la fonction « cpu ». Dans l'état actuel, le cœur est choisi de la façon suivante :

cœur = i % P ;

À quoi correspond cette politique de distribution ? Est-elle stupide ?

Influence du cache L3

Relevez les différents temps d'exécution des différentes versions pour une taille d'image 2048x2048, un grain de 32, et un twist assez grand (500). Pour les versions parallèles, utilisez OMP_NUM_THREADS=12.

Comparer les versions parallèles avec et sans first-touch (option –ft). Est-ce que cela a une influence sur les temps d'exécution ?

Pour mettre en valeur l'influence du cache L3, on pourra modifier le choix du cœur en adoptant un placement différent à la montée et à la descente :

cœur = ((sens == 1) ? i + 6: i) % P;

Influence sur les bancs mémoire

Reprendre le placement initial pour les taches de calcul (cœur = i % P). Mesurer le temps d'exécution pour la taille d'image 4096x4096, un grain de 32, et un twist assez grand (500). Pour les versions parallèles, utilisez OMP_NUM_THREADS=12.

Utilisez le script shell et le script R pour produire une courbe de speed up fonction du nombre de threads.

Rscript speedUp.R <fichier de données> <temps séquentiel>

Reprendre les mesures uniquement pour la version pthreads avec le paramètre -ft après avoir modifier la stratégie de placement du first-touch (ajouter la chaine anti-ft dans le nom du fichier de sortie) :

cœur = ((sens == 2) ? i + 6 : i) % P;