PC40 : Compte rendu UPC

André Russo

Introduction

Ce TP a pour but de découvrir UPC au cours d'une première partie Getting Started, puis de comprendre le fonctionnement de la parallélisation avec UPC au cours de la seconde partie sur le solveur laplace 1D. Enfin, la troisième partie sur le simulateur de conduction de chaleur 2D vise à comprendre et évaluer les différentes optimisations possibles pour améliorer la performance du code UPC.

Code

Le code source est sur Github à l'adresse https://github.com/andrerusso02/PC40_UPC.

Table des matières

Getting started

UPC Hello World

Lorsque l'on compile et exécute le programme getting_started/hello_world.upc sur 4 threads, on constate que le message Hello World s'affiche quatre fois et que les indices MYTHREAD de chaque message ne sont pas ordonnés. Ce comportement est attendu étant 4 threads sont exécutés simultanément, l'indice de chaque thread n'indique pas l'ordre de fin des threads.

Vector Addition

Le code getting_started/vector addition ajoute deux vecteurs v1 et v2 pour calculer le vecteur v1plusv2. Ces trois vecteurs sont déclarés shared et leur blocksize n'est pas spécifié : il est donc de 1 par défaut ce qui signifie que chaque case de chacun de ces vecteurs sont distribués physiquement dans l'espace local partagé des différents threads selon l'algorithme Round Robin. Ainsi, la case i d'un vecteur est placée dans l'espace partagé du thread i%THREADS. Or lors de l'addition, la condition "if(MYTHREAD == i%THREADS)" affecte justement le calcul impliquant les cases d'indices i des trois vecteurs au thread i%THREADS. Chaque thread n'utilise donc que des données partagées se situant dans son propre espace local partagé, et exploite donc le concept d'affinité d'UPC qui permet de réduire le temps d'accès au données. L'implémentation de cet additionneur est donc efficace.

Matrix-vector multiplication

Dans le code getting_started/matric_vector_mult.upc tiré du livre sur UPC, on multiplie la matrice carrée a[THREADS][THREADS] avec le vecteur b[THREADS] pour obtenir le vecteur c[THREADS].
Soit l'exécution du code sur 2 threads. Avec le blocksize par défaut de 1, la répartition de la mémoire partagée est la suivante :

Thread 0	Thread 1
a[0][0]	a[0][1]
a[1][0]	a[1][1]
b[0]	b[1]
c[0]	c[1]

Figure 1 : Répartition dans la mémoire partagée avec un blocksize de 1

On constate que chaque thread a alors une affinité avec une colonne d'indice MYTHREAD de la matrice a. Pour une multiplication de matrice, cela n'est pas très intéressant car pour

calculer la valeur d'une case d'indice MYTHDREAD de c, il faut parcourir l'ensemble de la ligne d'indice MYTHDREAD et non la colonne. Ainsi, il serait plus efficace d'affecter à chaque thread une ligne et non une colonne de a. C'est ce qui est fait dans le code getting_started/matric_vector_mult_efficient.upc, dont le blocksize de a est fixé à THREADS. Dans le cas de deux threads, on aura alors la répartition suivante :

Thread 0	Thread 1
a[0][0]	a[1][0]
a[0][1]	a[1][1]
b[0]	b[1]
c[0]	c[1]

Figure 1 : Répartition dans la mémoire partagée avec un blocksize de THREADS

1D Laplace solver

Code UPC à partir du code C (laplace_solver/ex_3.upc)

On déclare les trois tableaux comme shared afin qu'ils soient stockés dans l'espace partagé des threads. Sans cela, chaque thread aurait sa propre copie des trois tableaux dans son espace privé. Cela permet que lors de lors de l'initialisation réalisée entièrement par le thread 0, les valeurs donnés à x et b se répercutent pour l'ensemble des threads, et que lors de l'affichage, le thread 0 ait accès aux éléments de x_new calculés par les autres threads.

```
static shared double x_new[TOTALSIZE];
static shared double x[TOTALSIZE];
static shared double b[TOTALSIZE];
```

Pour éviter la race-condition, il faut placer l'instruction upc_barrier; juste avant la boucle d'affichage. Cette instruction bloque tant que tous les threads ne l'ont pas atteint. Ainsi, cela permet d'assurer que tous les threads terminent le calcul de x_new avant que le thead 0 passe à l'affichage.

Utilisation d'une simple boucle for (laplace_solver/ex_4.upc)

Lors du calcul de x_new, la boucle qui itère le calcul sur chaque élément de x_new est la suivante :

```
    for( j=1; j<TOTALSIZE; j++ )
```

On constate que dans un thread donné, le header de la boucle est évalué TOTALSIZE et uniquement certains tours de boucle conduisent effectivement au calcul d'un élément de x_new. Pour optimiser cela, on constate que pour un thread donné, celui-ci calcule les éléments de x_new d'indices MY_THREAD, MY_THREAD+THREADS, MY_THREAD+2*THREADS… On peut donc remplacer la boucle précédente par :

```
for( j=MYTHREAD; j<THREADS*TOTALSIZE-1; j+=THREADS)
```

Ainsi, le header de la boucle est évalué au plus MYTHREAD*TOTALSIZE/THREADS+1 fois.

Utilisation de blocs et d'upc_forall (laplace_solver/ex_5.upc)

Nous commençons par répartir différemment les 3 vecteurs dans la mémoire partagée. On spécifie une taille de blocs de BLOCKSIZE et on donne une taille de tableau de BLOCKSIZE*THREADS. Ainsi, chaque thread a dans son espace partagé local une portion de chaque vecteur de longueur BLOCKSIZE.

```
shared [BLOCKSIZE] double x[BLOCKSIZE*THREADS];
shared [BLOCKSIZE] double x_new[BLOCKSIZE*THREADS];
shared [BLOCKSIZE] double b[BLOCKSIZE*THREADS];
```

La boucle utilisée dans le code ex_3.upc affecte les threads aux calculs des différents éléments selon l'algorithme de Round Robin. UPC fournit la boucle upc_forall afin d'affecter plus simplement les ressources aux calculs à effectuer selon round robin. On peut alors remplacer la boucle précédente par :

```
upc_forall( j =1; j<BLOCKSIZE*THREADS-1; j++; &x_new[j]){
```

Ici, le dernier argument signifie qu'upc_forall va affecter chaque calcul au thread qui a une affinité avec x_new[j]. Cela permet de tirer au mieux parti de l'affinité malgré la nouvelle organisation en blocs des vecteurs.

Synchronisation (laplace_solver/ex_6.upc)

Nous cherchons maintenant à synchroniser la version du code ex_6.upc réalisant 10000 itérations. Pour cela, il faut ajouter un "upc_barrier;" après chaque upc_forall. En effet, upc_forall de garantit pas que tous les threads terminent en même temps la boucle. Il faut donc ajouter les upc_barrier pour garantir que le travail soit terminé avant de passer à l'inversion des pointeurs pour le premier upc_barrier, et à l'affichage pour le second.

Réduction

A l'aide de la librairie upc_collective et la fonction upc_all_reduceD, il devrait être possible de réduire un tableau de valeurs calculées localement par chaque thread en une seule variable partagée. Dans notre cas, cela devrait être :

```
upc_all_reduceD(diff, &diffmax, UPC_MAX, THREADS, 1, NULL,
UPC_IN_NOSYNC | UPC_OUT_NOSYNC);
```

Je n'ai cependant pas réussi à faire fonctionner cette fonction : après son appel diffmax reste à 0.0. J'ai donc simplement utilisé une boucle for :

```
if( MYTHREAD == 0 ){
    diffmax = diff[0];
    for( j = 1; j < THREADS; j++ ){
        if( diffmax < diff[j] )
            diffmax = diff[j];
    }
    printf("diff max = %f \n", diffmax);
    }
upc_barrier;
```

4 - Heat

Dans cette partie, nous allons dans un premier temps couvrir le fonctionnement des améliorations successives du programme de simulation de conduction de température 2D, puis nous allons analyser la performance des différentes itérations du programme.

Partie 1: Fonctionnement des codes

Le principe de la simulation est le suivant : on part d'une grille 2D d'une certaine dimension; au départ, la ligne supérieure est remplie de 1 pour modéliser une élévation de température d'une unité en haut de la grille.
Une double boucle parcourt alors l'ensemble de la grille (excepté les bordures) pour calculer la moyenne de température des 4 cases voisines de la case ciblée, ce qui permet de simuler la propagation de la chaleur.
Plusieurs itérations sont réalisées : jusqu'à ce que la variation de température à chaque itération passe en dessous d'un certain seuil.

Dans la version séquentielle fournie, deux tableaux 2D sont utilisés : à chaque itération, les valeurs du tableau "grid" sont utilisées pour calculer les nouvelles valeurs alors stockées dans le second tableau "new_grid". A la fin de chaque itération, il faut alors copier le contenu de "grid" dans "new_grid" et inversement.

Première version UPC (heat_1.upc)

L'objectif est dans un premier temps d'adapter le programme en C pour paralléliser son exécution avec UPC.

La section du code intéressante à paralléliser est la double boucle "for" réalisant une itération du calcul de la propagation de la température. D'une part, lorsque N est grand, le temps d'exécution de cette section devient important. D'autre part, étant donné que la grille "grid" ne change pas au cours d'une itération, le calcul de chaque case de "new_grid" est indépendant du calcul des autres cases de la grille. On peut donc répartir ce calcul sur différents threads pour optimiser le calcul de "new_grid" à chaque itération.

Pour réaliser la parallélisation, on affecte aux différents threads le calcul de lignes entières, en affectant chaque ligne selon l'algorithme Round Robin. L'une des boucles for imbriquées est alors remplacé par la construction "upc_forall" :

```
        upc_forall( i=1; i<N+1; i++; i)
        {
            // the thread calculates the new values for the row
            for( j=1; j<N+1; j++ )
            {
```

Les 2 grilles 2D doivent alors être accessibles par tous les threads, on les déclare donc shared:

```
shared [N+2] double grid[N+2][N+2], new_grid[N+2][N+2];
```

Enfin, pour calculer la variation de température maximum, il faut un moyen de centraliser la variation de température maximale trouvée par chaque thread. Pour cela, on utilise un tableau partagé dTmax_local[THREADS] dans lequel chaque thread partage la valeur qu'il a calculé.

```
shared double dTmax_local[THREADS];
……
dTmax_local[MYTHREAD] = dTmax;
upc barrier;
dTmax = dTmax_local[0];
for( i=1; i<THREADS; i++ )
      if( dTmax < dTmax_local[i] )
            dTmax = dTmax_local[i];
```

Cette première version parallèle n'est pas optimisée : copies de la grille, affinités non respectées…

Seconde version UPC: copies, affinités (heat/heat_3.upc)

Pour cette seconde version, nous cherchons à améliorer le temps d'exécution du programme, d'une part en évitant la copie de grid dans new_grid et inversement, et d'autre part en privilégiant l'accès par les threads à l'espace partagé ayant une affinité avec eux. En effet, tirer parti de l'affinité permet de diminuer le temps nécessaire pour accéder à une variable partagée.

Pour commencer, nous déclarons grid et new_grid avec un blocksize de N+2, pour distribuer chaque ligne des grilles dans l'espace partagé de façon à ce qu'une ligne d'indice i ait une affinité avec le thread i%THREADS.

```
shared [N+2] double grid[N+2][N+2], new_grid[N+2][N+2];
```

Cela est intéressant car le calcul d'une ligne donnée i de new_grid est ensuite réalisée par le thread i%THREADS. De plus, pour le calcul d'une case donnée, la moyenne des températures est calculée sur les valeurs des 4 cases voisines dont 2 appartiennent à la même ligne, donc on tire parti de l'affinité pour la lecture de ces 2 valeurs.
Finalement, on réduit donc de 5 accès à 2 accès sans affinité pour le calcul de chaque nouvelle valeur de la grille.

Ensuite, on souhaite éviter l'inversion de grid et new_grid qui s'effectue par copie de chaque valeur d'une grille dans l'autre.
Pour cela, nous créons deux tableaux de pointeurs privés vers shared :

```
shared [N+2] double *ptr[N+2], *new_ptr[N+2];
```

Chaque pointeur du tableau pointe est initialisé de sorte à pointer vers une ligne du tableau :

```
    for( i=0; i<N+2; i++ )
    {
        ptr[i] = &grid[i][0];
        new_ptr[i] = &new_grid[i][0];
    }
```

Finalement, il suffit alors de travailler uniquement avec ces deux tableaux de pointeurs dans le reste du code, et à simplement inverser les pointeurs à la fin de chaque itération :

```
for( k=0; k<N+2; k++ )
{
      tmp_ptr    = ptr[k];
      ptr[k]     = new_ptr[k];
      new_ptr[k] = tmp_ptr;
}
```

Dans cette seconde version, nous avons amélioré d'une part l'utilisation des affinités et supprimé la copie non nécessaire des grilles à la fin de chaque itération.

Troisième version UPC : privatisation (heat/heat_4.upc)

Pour la seconde version du code UPC, l'accès à l'espace partagé est réalisé à l'aide des pointeurs ptr[] et new_ptr[], pointeurs privés vers shared. Cependant, le coût d'utilisation de pointeurs privés vers shared est plus élevé que l'utilisation de pointeurs classiques.

Le modèle utilisé par UPC étant le PGAS (partitioned global adress space), chaque thread a son propre espace mémoire local, et la mémoire partagée est répartie physiquement dans les espaces locaux des threads. A l'aide de la syntaxe UPC, il est alors possible d'utiliser des pointeurs privé vers privé pour accéder à la mémoire partagée contenue dans l'espace local de chaque thread.

Dans le code, nous avons réparti les grilles par lignes entre les threads; cela signifie que chaque thread possède (N+2)/THREADS lignes dans son espace partagé local.

```
#define N_PRIVATE_ROWS (N+2)/THREADS
```

Nous créons deux tableaux de pointeurs ptr_priv et new_ptr priv, que l'on fait pointer vers le début de chaque ligne des grilles stockées dans l'espace local du thread en question.

```
double *ptr_priv[N_PRIVATE_ROWS], *new_ptr_priv[N_PRIVATE_ROWS];
…
for( i=0; i<N_PRIVATE_ROWS; i++ )
{
    ptr_priv[i] = (double *) ptr[i*THREADS+MYTHREAD];
    new_ptr_priv[i] = (double *) new_ptr[i*THREADS+MYTHREAD];
}
```

L’objectif est alors d’utiliser ces nouveaux pointeurs à la place des anciens lorsque qu’un thread accède aux données partagées avec lesquelles il a une affinité. Par exemple, pour le calcul de de T:

```
            T = (ptr_priv[i/THREADS][j-1]
                + ptr_priv[i/THREADS][j+1]
                + ptr[i-1][j]
                + ptr[i+1][j])
```

Ainsi, nous avons ici optimisé les temps d’accès à l’espace partagé en utilisant des pointeurs privés lorsque cela était possible.

Remarque : Pour cette partie, il était également demandé d’ajouter deux boucles for pour calculer les valeurs de température pour la première et la dernière ligne. Cependant, comme cette variation n’est pas présente dans les autres versions du programme, j’ai pensé qu'effectuer cette modification fausserait la comparaison de performance lors du benchmark, étant donné que cela reviendrait à rajouter des calculs qui ne sont pas réalisés dans les autres versions.

Quatrième version UPC (heat/heat_5.upc)

Nous désirons maintenant rendre le paramètre N dynamique, en donnant la possibilité de le donner en argument au moment de l'exécution.

Cela implique d’allouer dynamiquement l‘espace partagé utilisé par les deux grilles, étant donné d’une part que leur dimension est dépendante de N et d’autre part que leur blocksize de N+2 n’est plus connue à la compilation.

Pour allouer l’espace partagé à l'exécution, chaque thread va allouer l’espace correspondant aux lignes stockées dans son espace partagé local, à l’aide de la fonction upc_alloc, qui retourne un pointeur vers shared pointant sur la région partagée allouée. Nous utilisons également un tableau de pointeurs local_chunks[THREADS], dont chaque pointeur pointe vers les lignes partagées locales du thread correspondant. Le tableau local_chunks est partagé et permet donc l’accès de toutes les lignes de la grille par tous les threads.

```
shared[] double *shared local_chunks[THREADS];
shared[] double *shared new_local_chunks[THREADS];
…
local_chunks[MYTHREAD]
  = (shared[] double *shared)
upc_alloc(N_LOCAL_ROWS*(N+2)*sizeof(double));
new_local_chunks[MYTHREAD]
  = (shared[] double *shared)
upc_alloc(N_LOCAL_ROWS*(N+2)*sizeof(double));
```

Bien que l'on puisse alors travailler uniquement avec les tableaux local_chunks et new_local_chunks, il est plus intéressant d'utiliser lorsque cela est possible des pointeurs privés comme dans la version précédente.

```
ptr_priv = (double*) local_chunks[MYTHREAD];
    new_ptr_priv = (double*) new_local_chunks[MYTHREAD];
```

On peut alors réaliser le calcul de T; le calcul des indices des cases deviennent plus complexes étant donné que dans cette approche, la représentation des données sous la forme de tableaux de N+2 x N+2 cases est perdue.

```
T = 0.25 * (
      ptr_priv[(i/THREADS)*(N+2)+j-1] +
      ptr_priv[(i/THREADS)*(N+2)+j+1] +
      local_chunks[(i-1)%THREADS][(i-1)/THREADS*(N+2)+j] +
      local_chunks[(i+1)%THREADS][(i+1)/THREADS*(N+2)+j]
);
```

Enfin, il faut libérer l'espace alloué dynamiquement à la fin de l'exécution.

```
upc_free(local_chunks[MYTHREAD]);
upc_free(new_local_chunks[MYTHREAD]);
```

Partie 2 : Benchmark

Dans cette seconde partie, nous allons tester la performance temporelle des différentes versions réalisées.

Réalisation des benchmark

Les temps d'exécutions utilisés dans la suite de cette partie représentent uniquement le temps d'exécution de la partie parallélisée du code, c'est-à-dire du contenu de la boucle do…while.
Chaque version du code est compilée et exécutée pour différentes valeurs de N et nombres de threads.

Les tests ont été réalisés sur un ordinateur personnel. Le processeur Ryzen 7 4800H ayant 16 threads (8 coeurs), les codes ont été testés avec 2, 4, 8 et 16 threads. N+2 devant être multiple du nombre de threads, des multiples de 16 espacés de d'environ 150 ont été choisis pour N+2 : 16, 144, 304 et 448. Pour chaque nombre de threads et valeurs de N, la moyenne du temps d'exécution de chaque version a été calculée sur 10 exécutions.

Dans le dossier "benchmark" du rendu, les différents dossiers contiennent des versions d'un même code où seule la valeur de N varie; les scripts contenus dans le dossier "benchmark/scripts" permettent de compiler automatiquement ces différentes versions pour différents nombres de threads, de les exécuter un certain nombre de fois chacune et de faire

la moyenne du temps d'exécution. Le fonctionnement de ces scripts est détaillé dans README.md.

Résultats

Version séquentielle

Dans un premier temps, on mesure le temps d'exécution de heat_c.c, version séquentielle du programme, en fonction de N.

N	temps exec (s)
14	0,0003576
142	0,3500681
302	1,9661323
446	3,7819748

Figure 3 : Tableau des temps d'exécution heat_c.c en fonction de N

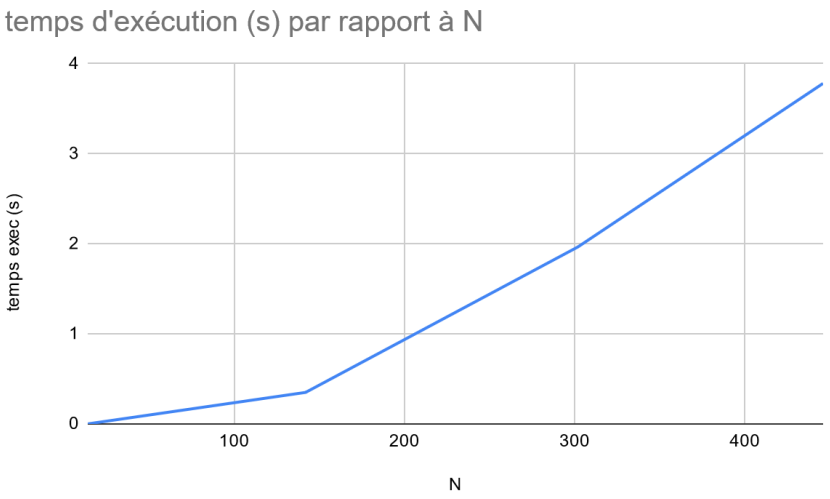

Figure 4 : Tableau des temps d'exécution heat_c.c en fonction de N

En observant le graphique, on constate que le temps d'exécution croît de façon non linéaire avec le nombre de lignes de grille. Etant donné que le code effectue une opération sur chaque case d'une grille 2D, on peut supposer que la complexité du problème est de l'ordre $O(n^2)$.

heat_1.upc

Nous testons maintenant le temps d'exécution de heat_1.upc, première version UPC de l'algorithme. Nou obtenons les résultats suivants :

temps exec (s)	*nb threads*			
N	2	4	8	16
14	0,0014696	0,0015192	0,0019595	0,0090528
142	1,2190106	1,1285672	1,1892603	2,0238671
302	6,4636618	5,8206168	6,1497922	8,3183864
446	13,7431278	12,5852369	12,8672014	16,0647774

Figure 5 : Tableau du temps d'exécution de heat_1.upc en fonction de N et du nombre de threads

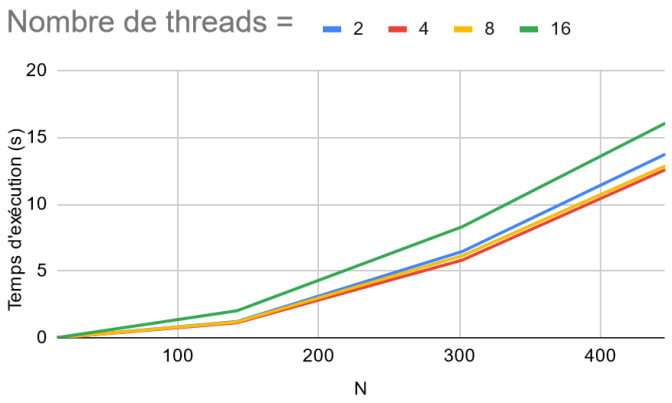

Figure 6 : Variation du temps d'exécution en fonction de N pour différents nombres des threads

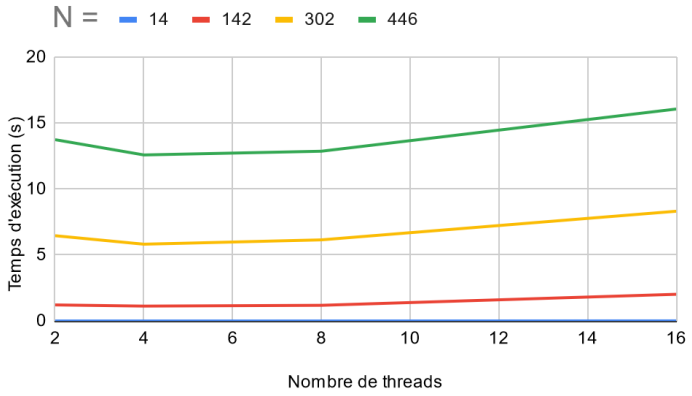

Figure 7 : Variation du temps d'exécution en fonction du nombre de threads pour différentes valeurs de N.

En observant les figures 5 et 6, on remarque que les performances de cette version sont dans l'ensemble beaucoup moins bonnes que pour la version séquentielle. Ainsi, pour N = 446, la meilleure performance atteinte avec 4 threads est un temps d'exécution de 12,5 secondes, alors que la version séquentielle calculait pour N = 446 en 3.7 secondes.
De plus, on remarque avec la figure 7 que le temps d'exécution augmente avec le nombre de threads, alors que l'on attend le contraire d'un code parallélisé.
Ce manque de performance montre que les optimisations réalisées par la suite sont réellement nécessaires.

heat_3.upc

Les résultats de benchmark pour heat_3.upc sont les suivants :

temps exec (s)	*nb threads*			
N	2	4	8	16
14	0,0007382	0,0007212	0,0008412	0,005963
142	0,3794406	0,2254574	0,1707774	0,2295112
302	2,030486	1,3447762	0,8957407	0,772493
446	4,3443709	2,8201066	1,9675799	1,6283634

Figure 8 : Tableau du temps d'exécution de heat_3.upc en fonction de N et du nombre de threads

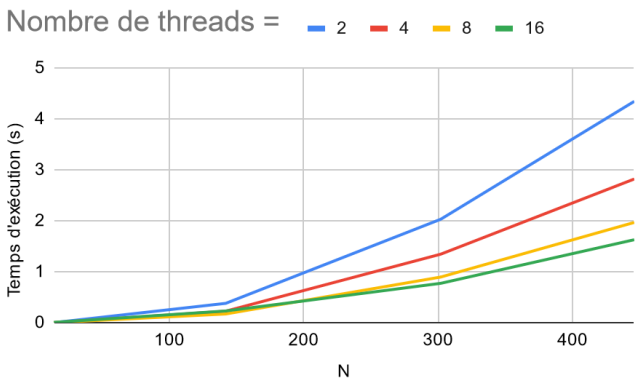

Figure 9 : Variation du temps d'exécution en fonction de N pour différents nombres des threads

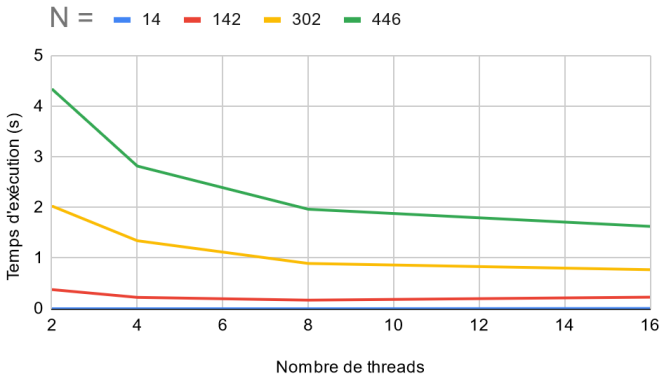

Figure 10 : Variation du temps d'exécution en fonction du nombre de threads pour différentes valeurs de N.

On constate d'après la figure 10 que l'augmentation du nombre de threads diminue bien cette fois le temps d'exécution, et ce d'autant plus que N est grand. Ainsi, cette seconde

version heat_3.upc pour laquelle on a enlevé la copie des grilles présente le comportement que l'on attend d'un programme parallélisé.
De plus, on remarque que lorsque l'on utilise 16 threads, le temps d'exécution avec N=302 et N=446 est plus court que la version séquentielle. On cependant nuancer, en effet la version séquentielle pourrait bénéficier d'optimisations similaires pour éviter la copie des grilles qui doit sans doute ralentir aussi son exécution.

heat_4.upc

Les résultats de benchmark pour heat_3.upc sont les suivants :

temp exec (s)	*nb threads*			
N	2	4	8	16
14	0,0004079	0,0004774	0,0005456	0,0281842
142	0,1616579	0,098608	0,0780122	0,1429766
302	0,8492993	0,5553526	0,3791389	0,3332603
446	1,8923698	1,2083607	0,8549406	0,9738217

Figure 11 : Tableau du temps d'exécution de heat_4.upc en fonction de N et du nombre de threads

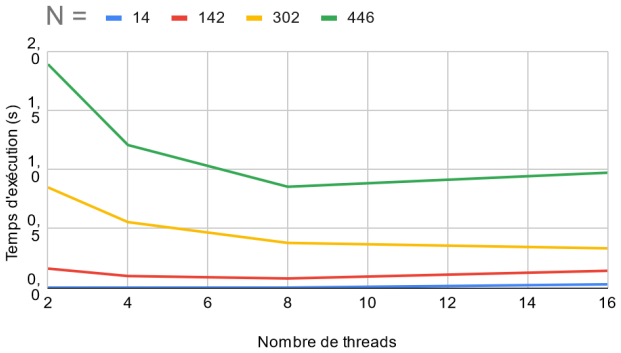

Figure 12 : Variation du temps d'exécution en fonction du nombre de threads pour différentes valeurs de N.

A la vue des courbes, on constate que les optimisations apportées par heat_4.upc permettent de réduire encore le temps d'exécution. Ainsi par exemple, entre heat_3.upc et heat_4.upc, on constate un temps d'exécution divisé par 2,3, et divisé par 5.9 par rapport à la version séquentielle. L'utilisation de pointeurs privés est donc un ressort d'optimisation important.

heat_5.upc

Les résultats de benchmark pour heat_3.upc sont les suivants :

temp exec (s)	*nb threads*			
N	2	4	8	16
14	0,0004793	0,0004995	0,0006525	0,0291115
142	0,139452	0,0873519	0,0607538	0,2068942
302	0,6604801	0,456638	0,3092844	0,4054903
446	1,4421945	0,9688506	0,7128609	0,6046529

Figure 13 : Tableau du temps d'exécution de heat_5.upc en fonction de N et du nombre de threads

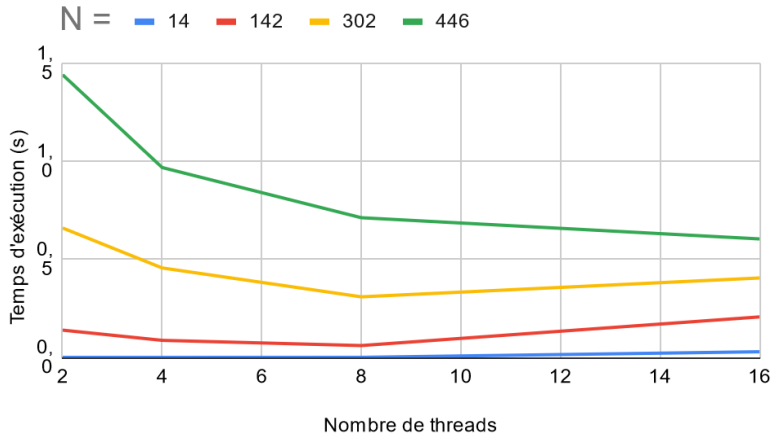

Figure 14 : Variation du temps d'exécution en fonction du nombre de threads pour différentes valeurs de N.

Pour cette version qui a la particularité d'allouer dynamiquement l'espace partagé, on constate peu de différence de vitesse d'exécution par rapport à la version précédente.

Conclusion

Au cours de ce TP, nous avons découvert UPC et évalué sa performance vis-à-vis d'un programme séquentiel. Nous avons également mesuré l'impact de différentes optimisations réalisables avec le modèle de programmation PGAS utilisé par UPC.