Solution Cours–Laboratoire #8
Programmation parallèle avec OpenMP/C

Mardi, 14 novembre 2017

1. Calcul de π avec différentes façons d'effectuer la réduction

Solution avec une “vraie” réduction

```
#pragma omp parallel for reduction(+: somme)
for ( int i = 0; i < nb_rectangles; i++ ) {
  double x = (i + 0.5) * largeur;
  double fx = 4.0 / (1.0 + x*x);
  somme += fx;
}
```

Comparaison des temps d'exécution (sur japet)

`nb_rectangles = 10000000`

	nb_threads		
	8	16	32
critical	6.570		
atomic	1.759		
reduction			

Comparaison des temps d'exécution (sur japet)

`nb_rectangles = 10000000`

	nb_threads		
	8	16	32
critical	6.570	16.930	41.907
atomic	1.759	2.712	3.349
reduction			

Comparaison des temps d'exécution (sur japet)

`nb_rectangles = 10000000`

	nb_threads		
	8	16	32
critical	6.570	16.930	41.907
atomic	1.759	2.712	3.349
reduction	0.016		

Comparaison des temps d'exécution (sur japet)

`nb_rectangles = 10000000`

	nb_threads		
	8	16	32
critical	6.570	16.930	41.907
atomic	1.759	2.712	3.349
reduction	0.016	0.009	0.005

Effet de `omp_set_dynamic`

Enable or disable the dynamic adjustment of the number of threads within a team. The function takes the language-specific equivalent of true and false, **where true enables dynamic adjustment of team sizes and false disables it**.

Donc :

- si `true` (1) $\Rightarrow$ Nombre de threads utilisés peut ne pas être le nombre demandé
- si `false` (0) $\Rightarrow$ Nombre de threads utilisé est celui demandé

Valeur par défaut = `false` (0)

2. Recherche du maximum

Solution avec une “vraie” réduction

```
int leMax = a[0];
#pragma omp parallel for reduction(max: leMax)
for ( int i = 1; i < n; i++ ) {
  if ( a[i] > leMax ) {
    leMax = a[i];
  }
}
```

b. Si on utilise un autre opérateur que “>”...

Le programme produit le mauvais résultat : la réduction avec `max` et `leMax` ne s'effectue que pour combiner les résultats intermédiaires produits par chacun des threads.

Analogie avec le `preduce` de `PRuby` :

- Instructions dans le bloc du `for`

  $\approx$ bloc (deux arguments) passé à `preduce` ;

- Opérateur indiqué dans la clause `reduction` du *pragma*

  $\approx$ opérateur indiqué par l'argument `final_reduce:`

C. Si on utilise “guided”...

Par exemple, avec 200 éléments et 4 threads, voici les items traités par certains des threads :

```
Thread 1 =>
 [89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
  99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
  109, 110, 111, 112, 113, 114, 115, 116]

Thread 3 =>
 [117, 118, 119, 120, 121, 122, 123, 124, 125, 126,
  127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137,
  182, 183, 184, 185, 186, 198]
```

3. Approche diviser-pour-régner pour calculer la somme de 1 à n : Avec parallélisme imbriquée vs. avec parallélisme de tâches

```
int somme( int i, int j, int seuil )
{
  if ( j - i < seuil ) {
    int r = 0;
    for ( int k = i; k <= j; k++ )
      r += k;
    return r;
  } else {
    int mid = (i + j) / 2;
    int r1, r2;
    {
      #pragma omp task shared(r1)
      r1 = somme( i, mid, seuil );

      #pragma omp task shared(r2)
      r2 = somme( mid+1, j, seuil );
    }

    #pragma omp taskwait
    return r1 + r2;
  }
}
```

```
...

// On fixe le nombre de threads a utiliser.
omp_set_num_threads( nb_threads );

// On calcule la somme avec la fonction recursive.
int res;
#pragma omp parallel
#pragma omp master    // Ou single
res = somme( 1, n, seuil );

...
```

Lorsque le nombre de *threads* est “grand” (plus que 3), le *thread* maitre (et d’autres *threads* si plus que 3 *threads*) ne semble traiter qu’une seule tâche — et ce même lorsque la quantité de tâches à traiter est grande.

4. Fonctions pour effectuer le produit de polynomes

```
static double coefficient( int k,
                           Polynome p1,
                           Polynome p2 )
{
  int expMinR1 = MAX( 0, k-p2.degre );
  int expMaxR1 = MIN( k, p1.degre );

  assert( expMinR1 <= expMaxR1 && "expMinR1 doit etre

  double c = 0.0;
  # pragma omp parallel for reduction(+: c)
  for( int i = expMinR1; i <= expMaxR1; i++ ) {
    c += p1.coefficients[i] * p2.coefficients[k-i];
  }

  return c;
}
```

```
Polynome fois( Polynome p1, Polynome p2 )
{
  Polynome p = allouer( p1.degre + p2.degre );

  # pragma omp parallel for schedule(runtime)
  for( int k = 0; k <= p.degre; k++ ) {
    p.coefficients[k] = coefficient( k, p1, p2 );
  }

  return p;
}
```

Comparaison des temps d'exécution (sur japet)

`Taille des polynomes à multiplier = 40000`

	nb_threads			
	4	8	16	32
static	4.06	2.38	1.30	0.75
static, 10	2.70	1.40	0.81	0.45
dynamic	2.69	1.42	0.76	0.43
dynamic, 10	2.66	1.44	0.77	0.45
guided	2.76	1.48	0.82	0.43

Note : Le mode "automatic" n'est pas disponible dans la version d'OpenMP/C installée sur japet.

La répartition par blocs d'éléments adjacents — i.e., `schedule(static)` — est la moins performante.

La fonction `fois` utilise du parallélisme de résultat.

Le travail à faire pour calculer le i^e coefficient — `coefficient(i, p1, p2)` — varie d'un i à un autre et a l'allure indiqué dans le diagramme à la page suivante.

```
0:      X
1:      XX
2:      XXX
3:      XXXX
4:      XXXXX
.
.
.
n/2-1: XXXXX ...  XXXX
n/2:   XXXXX ...  XXXXX      # Position milieu
n/2+1: XXXXX ...  XXXX
.
.
.
n-4:   XXXX
n-3:   XXX
n-2:   XX
n-1:   X
```

Temps : clock() vs. omp_get_wtime()

```
debut_ticks = clock();
debut = omp_get_wtime();

Polynome res = fois( p0, p1 );

fin_ticks = clock();
fin = omp_get_wtime();
```

```
Execution avec N = 40000 et avec 1 threads
Temps (tics) =    10.5400 s
Temps (time) =    10.5490 s
Execution avec N = 40000 et avec 2 threads
Temps (tics) =    10.5600 s
Temps (time) =     5.2823 s
Execution avec N = 40000 et avec 4 threads
Temps (tics) =    10.6900 s
Temps (time) =     2.7016 s
```

Pourquoi?

Temps : `clock()` vs. `omp_get_wtime()`

Pourquoi?

> `clock()` *measure the CPU time used by your process, not the wall-clock time. When you have multiple threads running simultaneously, you can obviously burn through CPU time much faster.*

Source :

`stackoverflow.com/questions/2962785/c-using-clock-to-measure-time-in-multi-threaded-programs`