IMT Atlantique

UVF2B304: Ingénierie du développement logiciel

TP : Parallélisme

Quan VO

Exercice 1. Min/Max

Premier essai

L'approche classique, on fait

```
static void minmax(float const* start, float const* stop, float* min, float* max) {
    // giveme giveme a man fmin
    float const* i;
    float m_min, m_max, temp;

    for (i = start; i < stop; i++){
        temp = *(i);
        if (m_min > temp){
            m_min = temp;
        }
        if (m_max < temp){
            m_max = temp;
        }
    }

    *min = m_min;
    *max = m_max;
}
```

Deuxième essai

Avec OpenMP, on utilise plusieurs threads pour calculer en même temps, chaque thread fait une partie de la réduction. On teste avec 4, 8 threads.

```
static void minmax(float const* start, float const* stop, float* min, float* max) {
    // giveme giveme a man fmin
    float const* i;
    float m_min, m_max, temp;
    omp_set_num_threads(4);
    #pragma omp parallel for private(temp) reduction(max : m_max) reduction(min: m_min)
    for (i = start; i < stop; i++){
        temp = *(i);
        /*printf("thread id = %d\n", omp_get_thread_num());*/
        if (m_min > temp){
            m_min = temp;
        }
        if (temp > m_max){
            m_max = temp;
        }
    }
```

```
    *min = m_min;
    *max = m_max;
}
```

Troisième essai

Utilise des instructions vectorielles, ici la technologie SSE. On charge chaque fois 128 bits (compose de 4 éléments à virgule flottant de 32 bits), et cherche la valeur max parmi ces éléments.

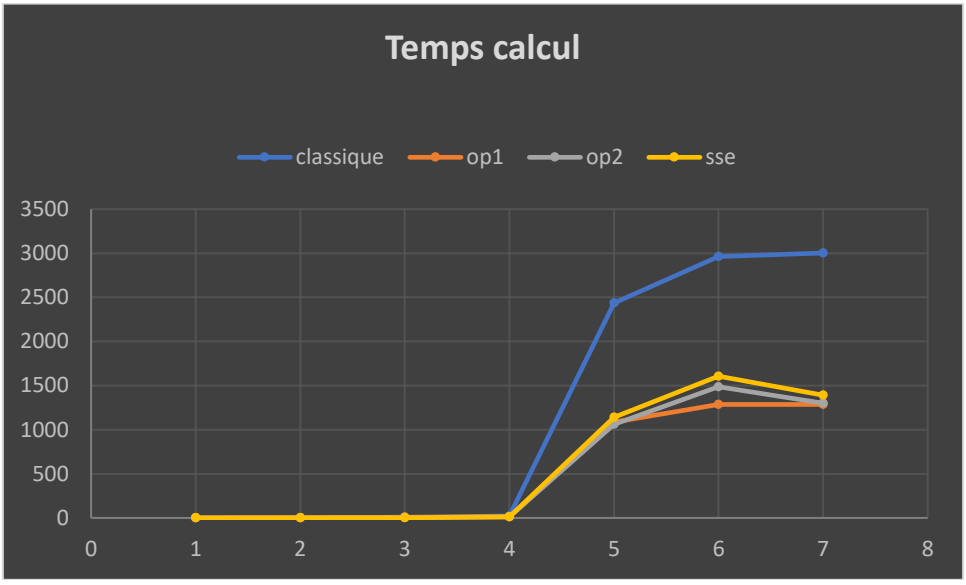
```
float horizontal_max_Vec4(__m128 x) {
    __m128 max1 = _mm_shuffle_ps(x, x, _MM_SHUFFLE(0,0,3,2));
    __m128 max2 = _mm_max_ps(x, max1);
    __m128 max3 = _mm_shuffle_ps(max2, max2, _MM_SHUFFLE(0,0,0,1));
    __m128 max4 = _mm_max_ps(max2, max3);
    float result = _mm_cvtss_f32(max4);
    return result;
}

static void minmax(float const* start, float const* stop, float* min, float* max) {
    // giveme giveme a man fmin
    float m_min;

    __m128 m_max = _mm_loadu_ps(start);

    while (start < stop){
        start = start + 4;
        __m128 cur = _mm_loadu_ps(start);
        m_max = _mm_max_ps(m_max, cur);
    }
    printf("result %.3f\n", horizontal_max_Vec4(m_max));
}
```

Comparaison le temps de calcul

Temps en ms

Nombre d'éléments	min_max classique	min_max_op1 4 threads	min_max_op2 8 threads	min_max_intrinsics SSE m_128
1000	0.011	0.245	0.188	0.059
100 000	0.7	2.947	0.676	0.482
1 000 000	3.7	9.486	4.091	1.88
10 000 000	24.5	11	11	11.411
1 000 000 000	2438	1085	1062	1140
10 000 000 000	2963	1286	1487	1607
100 000 000 000	3003	1285	1300	1393

Figure et remarque



Le calcul parallèle n'est vraiment efficace que pour les calculs avec beaucoup de valeurs. Quand il n'y a pas beaucoup d'éléments, il peut augmenter le temps de calcul car nous devons échanger entre des threads.

Exercice 2. Atomic

Dans cet exercice, on va découvrir comment on assure le « thread safe » quand il y a plusieurs threads qui accèdent un même objet.
On teste avec ces commandes :

- export OMP_NUM_THREADS=num : initialiser le nombre de threads
- gcc -fopenmp code.c -o code
- ./code add_num rm_num

Premier essai

Utilise OpenMP « parallel » pour créer des threads, on partage la taille du liste (final_size) et la liste (curr) mais chaque thread a des variables locales (add_count, rm_count). Ça veut dire, si les threads modifient en même temps l'objet « curr », il est très probable qu'on ajoute 2 fois mais la valeur de la taille augment par 1.

```
int main(int argc, char **argv)
{
  if (argc != 3)
    return 1;
  int add_count, rm_count;
  list_t curr = empty_list;
  size_t final_size;

#pragma omp parallel private(add_count, rm_count) shared(final_size, curr)
  {
    add_count = atoi(argv[1]);
    rm_count = atoi(argv[2]);
    final_size = process(&curr, add_count, rm_count);
    printf("in thread %d = %zd\n", omp_get_thread_num(), final_size);
```

```
  }
  printf("total: %zd\n", final_size);
  return 0;
}
```

Résultats : parfois vrai, parfois faux.

Deuxième essai

On crée beaucoup de threads comme le premier essai, mais avec « critical » sur les parties qui affectent la liste et la taille.

```
void list_push_front(list_t *self, int val)
{
  list_t res = make_list(val);
  res->next = *self;
  #pragma omp critical(l)
  *self = res;
}

int list_pop_front(list_t *self)
{
  assert(*self && "pop from empty list o_O");
  int val;
  #pragma omp critical(l)
  {
    val = (*self)->val;
    list_t next = (*self)->next;
    free(*self);
    *self = next;
  }
  return val;
}

int list_front(list_t self)
{
  assert(self && "front from empty list O_o");
  return self->val;
}

size_t list_size(list_t self)
{
  size_t n = 0;
  #pragma omp critical(l)
  while (self)
  {
    n += 1;
    self = self->next;
  }
  return n;
}
```

Troisième essai

On fait la même chose mais avec le mot-clé « atomic » de OpenMP. Pour cela, il nous faut distinguer des expressions différentes comme : « read », « write », « update »

```
void list_push_front(list_t *self, int val)
{
  list_t res = make_list(val);
  res->next = *self;
#pragma omp atomic write
  *self = res;
}

int list_pop_front(list_t *self)
{
  assert(*self && "pop from empty list o_O");
  int val;
#pragma omp atomic write
  val = (*self)->val;
  list_t next = (*self)->next;
  free(*self);
#pragma omp atomic write
  *self = next;

  return val;
}

int list_front(list_t self)
{
  assert(self && "front from empty list O_o");
  return self->val;
}

size_t list_size(list_t self)
{
  size_t n = 0;

  while (self)
  {
#pragma omp atomic update
    n += 1;
#pragma omp atomic write
    self = self->next;
  }
  return n;
}
```

Résultats : Dans le 2e et 3e cas, il fonctionne correctement, mais parfois avec le nombre de threads est grand et on exécute plusieurs fois de suite, il rend faux. Je suppose que le processeur de la machine est occupé donc il ne peut pas s'occuper et partager de manière correcte.