Département d'informatique
Faculté des sciences
UNIVERSITÉ DE SHERBROOKE

Projet

par
Mathieu Canuel-Ross, 10 080 217
Charles Coulombe, 11 043 186

travail présenté à
Gabriel Girard

dans le cadre du cours
IFT630 - Processus concurrents et parallèlisme

15 avril 2013

Table des matières

1 Introduction 3

2 But 3

3 Mise en contexte 3

4 Tests 3

5 Spécifications 4

6 Technologies 4

7 Hypothèses 4

8 Algorithmes 4
- 8.1 UpdateRange . . . 5
  - 8.1.1 Séquentiel . . . 5
  - 8.1.2 Fils d'exécution . . . 5
  - 8.1.3 OpenMP . . . 5
- 8.2 Output . . . 5
  - 8.2.1 Séquentiel . . . 5
  - 8.2.2 Fils d'exécution . . . 5
  - 8.2.3 OpenMP . . . 6

9 Résultats 6
- 9.1 Test 1 . . . 6
- 9.2 Test 2 . . . 7
- 9.3 Test 3 . . . 8
- 9.4 Test 4 . . . 9

10 Conclusion 10

11 Annexes 11

Références 26

Table des figures

1 Résultats du test 1 7
2 Résultats du test 2 8
3 Résultats du test 3 8
4 Résultats du test 4 9

Table des algorithmes

1 UpdateRange 11
2 UpdateRangeThreads 12
3 UpdateRangeThreadsTask 13
4 UpdateRangeOpenMP 14
5 Output 16
6 OutputThreads 19
7 OutputThreadsTask1 20
8 OutputThreadsTask2 22
9 OutputOpenMP 23

1 Introduction

De nos jours, la génération de données en grande quantité se fait de plus en plus fréquente dans le domaine scientifique, par exemple, en biologie, dans le cas des techniques de séquençage à haut débit ou encore lors de l'analyse de phénomènes physiques. De plus, les technologies, qui évoluent très rapidement, nous permettent de traiter de plus grands bassins de données. La programmation concurrente et parallèle s'intègre bien dans ces domaines, permettant de traiter plus efficacement de grands lots de données. Bien que de plus en plus répandues, les architectures parallèles sont limitées par certaines contraintes matérielles telles que la quantité de processeurs et la quantité de cœurs d'un processeur qui sont nettement inférieures à la quantité de données que ces architectures sont appelées à traiter.

C'est pourquoi, dans le cadre du cours IFT630, nous avons décidé de mettre à profit les notions de programmation parallèle et les outils vus à ce jour pour optimiser un outil d'analyse de données génomiques.

2 But

Le présent projet a pour but d'optimiser, en y intégrant les fils d'exécution, l'outil d'analyse de données génomiques et d'ensuite comparer les algorithmes avec les différentes implémentations suivantes :

- Séquentielle ;
- Fils d'exécution, Tinythread++[1] ;
- OpenMP[2].

3 Mise en contexte

L'outil d'analyse de données génomiques a été à la base conçu par Pierre-Étienne Jacques, Ph.D, chercheur/professeur au département de biologie de l'Université de Sherbrooke. L'outil a ensuite été réécrit pour y intégrer de nouvelles fonctionnalités et permettre son optimisation.

L'outil développé a pour but de calculer de façon précise le profil moyen de données de localisation génomiques. Voici un bref résumé de son fonctionnement : chaque groupe de gènes est mis à jour à partir des séquences contenues dans le fichier de données génomiques. Typiquement, sept fichiers sont créés par groupe : un pour chacune des sept orientations graphiques. Il y a aussi la possibilité de créer un fichier par fichier de données et d'écrire tous les gènes.

4 Tests

Nous avons effectué quatre séries de tests et ceux-ci sont présentés ci-dessous :

1. Six groupes de 6460 gènes et cinq fichiers de données (~5 * 40 000) ;
2. Six groupes de 6460 gènes et un fichier de données (~1 * 40 000) ;
3. Un groupe de 312 gènes et cinq fichiers de données (~5 * 40 000) ;
4. Un groupe de 7052 gènes et un fichier de données (1 247 764).

Chaque test comprend cinq essais évaluant trois aspects : l'algorithme de mise à jour, l'écriture sur disque et le temps d'exécution total.

5 Spécifications

Les tests ont été effectués sur un ordinateur ayant les spécifications suivantes :

- Système : Windows 7 (64 bits) ;
- Processeur : I5 (4 cœurs) ;
- Mémoire vive : 4 Go ;
- Disque : SSD 128 Go.

À noter que le processus détenait une priorité normale sur le système d'exploitation et possédait une affinité avec tous les cœurs.

6 Technologies

Nous avons implémenté les fils d'exécution en utilisant la librairie TinyThread++. Cette librairie multiplateforme est écrite en C++ et fournie une interface comprenant les outils nécessaires pour la concurrence selon les normes du standard C++11. L'interface de l'API est basée sur les fonctionnalités fournies par le système d'exploitation.

Nous avons aussi mis à l'essai, à des fins de comparaisons, l'API OpenMP. Celui-ci étant déjà intégré à C++, il suffit d'ajouter une instruction spéciale au compilateur afin de pouvoir utiliser ses fonctions.

7 Hypothèses

Il est bel et bien entendu que le temps de calcul varie selon l'implémentation des algorithmes et la quantité de données à traiter. Dans notre cas, les algorithmes sont de l'ordre O(N). Bien que la quantité de données varie, l'outil possède tout de même un temps d'exécution minimal, qui varie entre 5 et 7 secondes.

Sachant cela, nous pouvons estimer le temps d'exécution d'un petit test (moins de 10 000 gènes et un seul fichier de données) comme étant négligeable et donc possédant un temps d'exécution d'environ six secondes.

De plus, il est à noter que l'implémentation des fils d'exécution peut nuire au gain de performance lors d'un scénario où l'exécution est de courte durée(en termes de quelques secondes). Cela s'explique par le traitement requis pour la gestion des fils d'exécution. Ainsi, nous supposons que le gain obtenu par l'ajout des fils d'exécution sera très minime voir négligeable ou négatif pour le plus petits tests, mais qu'il sera de l'ordre de 30% pour le test 1 et davantage pour le test 4. Ce dernier, se veut, à notre avis, le plus important des tests effectués étant donné la quantité de données.

Nous estimons qu'OpenMP sera moins performant que TinyThread++ lors d'une petite charge de travail étant donné la gestion sous-jacente des fils d'exécution, qui à notre avis est plus lourde que celle de TinyThread++.

8 Algorithmes

Dans ce projet, deux sections de l'outil ont été modifiées pour y implémenter les fils d'exécution. En voici une brève description.

8.1 UpdateRange

L'algorithme met à jour les gènes pour chaque fichier de données.

8.1.1 Séquentiel

L'implémentation séquentielle de l'algorithme est relativement simple. Il a d'abord été conçu et optimisé pour un fonctionnement séquentiel. De plus, que la charge de travail soit relativement petite ou moyenne, le temps de traitement est équivalent.

L'algorithme est en annexe : UpdateRange.

8.1.2 Fils d'exécution

Il n'a pas été difficile d'optimiser cet algorithme, car la structure se prêtait déjà très bien à l'ajout des fils d'exécution. Pour ce faire, nous avons ajouté une couche supplémentaire à la librairie TinyThread++ pour complémenter et adapter ses fonctionnalités au code de l'outil. Nous avons ajouté deux classes, soit l'objet *thread* et *thread_list*, nous permettant de démarrer et d'attendre la fin de plusieurs fils contenus dans la liste. L'objet *thread* quant à lui nous permet de séparer la création de l'exécution du fil et d'utiliser un pointeur intelligent pour manipuler le fil sous-jacent.

L'algorithme est en annexe : UpdateRangeThreads et UpdateRangeThreadsTask.

8.1.3 OpenMP

Même si l'utilisation d'OpenMP est relativement simple et que la documentation est abondante sur le sujet, nous avons eu quelques difficultés à l'intégrer à l'algorithme. Nous avons dû adapter une petite partie du code pour satisfaire aux spécifications d'OpenMP. Notre plan était d'utiliser le « parallel for » sur notre structure de données sans modifier le code séquentiel. Notre structure de données étant un dictionnaire, nous parcourons les éléments à l'aide d'un itérateur. Malheureusement, selon les spécifications d'OpenMP pour la boucle « parallel for » l'utilisation des valeurs entières et de certains opérateurs spécifiques ($<$, $>$, $=$) est obligatoire[4]. Pour contourner ce problème, nous avons créé un tableau statique de pointeurs qui est itéré à l'aide d'un entier.

L'algorithme est en annexe : UpdateRangeOpenMP.

8.2 Output

L'algorithme permet l'écriture des résultats sur le disque dur.

8.2.1 Séquentiel

Encore plus simple que le précédent, cet algorithme écrit sur disque les résultats de chaque orientation pour tous les groupes de gènes. Par la suite, l'écriture de tous les gènes avec leurs données respectives est faite selon la configuration de l'outil.

L'algorithme est en annexe : Output.

8.2.2 Fils d'exécution

Il n'a pas été difficile d'ajouter les fils d'exécution pour cet algorithme qui est très simple. La boucle principale (orientation) correspond à la tâche 1 et l'écriture de tous les gènes à la tâche 2.

Ces deux tâches peuvent être exécutées en parallèle.

L'algorithme est en annexe : OutputThreads et OutputThreadsTask1 et OutputThreadsTask2.

8.2.3 OpenMP

La modification de cet algorithme pour OpenMP a nécessité un peu plus de recherche. Nous avons dû utiliser les énoncés « sections, section, nowait » pour identifier une section à être traitée en parallèle, mais indépendante de la boucle de la tâche 1.

L'algorithme est en annexe : OutputOpenMP.

9 Résultats

Les résultats des trois séries de tests effectués sont présentés sous forme de tableau. Chaque entrée correspond au temps en millisecondes que la section évaluée a nécessité pour être complétée. La section bleue représente le total du banc d'essai.

Les différences de temps d'exécution avec la version séquentielle sont notées sous la colonne « Différence » et le gain de performance est noté en pourcentage sous la colonne « Amélioration ».

9.1 Test 1

Le premier test met clairement en évidence le gain de performance acquis par l'ajout des fils d'exécution. On peut remarquer qu'à chaque essai, le temps d'exécution de l'algorithme est diminué de presque 50%. C'est un gain notoire et significatif qui s'explique par la charge de travail considérable que requiert la mise à jour des gènes. C'est d'ailleurs le goulot d'étranglement de l'outil et c'est pourquoi on peut y constater un gain important.

Pour ce qui est de la section d'écriture des résultats, il n'est pas surprenant de constater que le temps est diminué d'un peu plus de 50%. L'importante quantité de données devant être itérées séquentiellement explique d'elle-même l'écart avec la version parallèle.

Bien que le temps global ne présente pas un gain aussi important que les deux sections composites, le gain de 18% est significatif puisqu'il est influencé par les deux algorithmes et la quantité de données. Ce gain augmentera avec une charge de travail plus élevée.

Pour OpenMP, les résultats sont très similaires à l'utilisation des fils d'exécution. On note une différence de 2% en faveur des fils d'exécution pour le temps global d'exécution.

Levure : 6 groupes & 5 fichiers de données							
	Séquentiel	Fils	OpenMP				
Essai 1							
Algo	3038	1743	1654				
Output	2517	615	658				
Total	17667	14136	14044				
Essai 2							
Algo	3192	1745	1733				
Output	1084	597	702				
Total	17109	13812	14146				
Essai 3							
Algo	3050	1700	1743				
Output	1108	664	779				
Total	16254	13356	14339				
Essai 4							
Algo	3160	1709	1756				
Output	1099	597	641				
Total	16661	13449	13915				
Essai 5							
Algo	3068	1751	1715				
Output	1155	593	707	Fils d'exécutions		OpenMp	
Total	16362	13763	14106	Différence	Amélioration	Différence	Amélioration
Algo	3101,6	1729,6	1720,2	1372	44%	1381,4	45%
Output	1392,6	613,2	697,4	779,4	56%	695,2	50%
Total	16810,6	13703,2	14110	3107,4	18%	2700,6	16%

FIGURE 1 – Résultats du test 1

9.2 Test 2

Dans le présent test, la charge de travail est minime. Par contre, nous pouvons observer un gain presqu'aussi important que pour le test 1 pour les deux algorithmes évalués.

Cependant, le gain total de 8% est minimisé par la charge de travail. Il est donc négligeable dans l'ensemble du temps d'exécution. Nous nous attendions à n'avoir aucun gain, voir même à observer une diminution. Nous sommes donc agréablement surpris de voir, que malgré tout, il y ait un petit gain.

Sur les cinq essais, les différences de temps se sont situées dans le même intervalle, ce qui s'explique encore par la charge de travail minime.

Entre OpenMP et l'algorithme séquentiel, on note une amélioration de 15% et de 7% avec les fils d'exécution. Il est difficile d'expliquer le gain de 7% étant donné que la gestion des fils d'exécution par OpenMP semblait un peu plus lourde que celle faite par la librairie TinyThread++. Nous sommes quelques peu étonné, étant donné que ce résultat est contraire à notre hypothèse de départ.

Levure : 6 groupes & 1 fichier de données							
	Séquentiel	Fils	OpenMP				
Essai 1							
Algo	620	359	321				
Output	254	159	123				
Total	5481	5105	4635				
Essai 2							
Algo	633	343	358				
Output	236	129	280				
Total	5722	5020	4847				
Essai 3							
Algo	621	368	333				
Output	283	139	138				
Total	5602	5212	4679				
Essai 4							
Algo	602	356	331				
Output	262	141	122				
Total	5499	5101	4783				
Essai 5							
Algo	617	369	335				
Output	226	144	125	Fils d'exécutions		OpenMp	
Total	5523	5252	4751	Différence	Amélioration	Différence	Amélioration
Algo	618,6	359	335,6	259,6	42%	283	46%
Output	252,2	142,4	157,6	109,8	44%	94,6	38%
Total	5565,4	5138	4739	427,4	8%	826,4	15%

FIGURE 2 – Résultats du test 2

9.3 Test 3

Le test 3 se veut plus élevé en importance au niveau de la charge de travail que le test 2 puisqu'il requiert l'itération sur les six fichiers de données et l'écriture des résultats pour chacun d'entre eux. Ce test sert principalement à soutenir le test 2 en démontrant que le gain reste négligeable lors d'une charge moyennement plus élevée.

Le traitement des fichiers de données étant itératif, il n'est pas étonnant de constater un gain de 4% seulement pour une différence de temps de 5 secondes environ. Malgré tout, les algorithmes ont encore une fois bénéficiés des fils d'exécution puisqu'ils affichent un gain de 44% et 42% respectivement.

OpenMP démontre que ce test n'est pas si négligeable puisqu'il affiche un gain de 13% comparativement à 4% pour TinyThread++ pour l'exécution globale. De plus, on remarque une constance des résultats d'OpenMP pour chacun des essais.

Levure : 1 groupe & 5 fichiers de données							
	Séquentiel	Fils	OpenMP				
Essai 1							
Algo	979	558	533				
Output	223	138	148				
Total	11480	10785	9888				
Essai 2							
Algo	960	555	483				
Output	208	129	122				
Total	11293	10935	9883				
Essai 3							
Algo	988	559	506				
Output	242	114	122				
Total	11463	10946	9941				
Essai 4							
Algo	980	533	546				
Output	219	120	144				
Total	11328	10843	9988				
Essai 5							
Algo	989	526	506				
Output	207	134	140	Fils d'exécutions		OpenMp	
Total	11400	10930	9837	Différence	Amélioration	Différence	Amélioration
Algo	979,2	546,2	514,8	433	44%	464,4	47%
Output	219,8	127	135,2	92,8	42%	84,6	38%
Total	11392,8	10887,8	9907,4	505	4%	1485,4	13%

FIGURE 3 – Résultats du test 3

9.4 Test 4

Nous remarquons que les deux algorithmes parallèles obtiennent des améliorations similaires aux tests précédents. La différence de 7% de l'algorithme Output s'explique par le fait qu'OpenMP doit gérer les énoncés « sections, section, nowait ». On note une amélioration de 4% sur l'exécution totale.

Contrairement à notre hypothèse, les différences ne sont pas aussi significatives que nous l'aurions cru. Ceci est principalement dû à la lecture et au tri séquentiel du fichier qui est beaucoup plus gros que les précédents.

Levure : 1 groupe & 1 fichier de données							
	Séquentiel	Fils	OpenMP				
Essai 1							
Algo	12515	7680	7037				
Output	41	21	26				
Total	222666	204775	196852				
Essai 2							
Algo	12714	7359	6952				
Output	46	22	40				
Total	212919	204684	198608				
Essai 3							
Algo	12430	7712	6909				
Output	49	29	23				
Total	220331	206321	196370				
Essai 4							
Algo	12253	7428	6968				
Output	49	22	21				
Total	208316	205189	196512				
Essai 5							
Algo	13503	7407	7038				
Output	43	25	25	Fils d'exécutions		OpenMp	
Total	214636	206756	196866	Différence	Amélioration	Différence	Amélioration
Algo	12683	7517,2	6980,8	5165,8	41%	5702,2	45%
Output	45,6	23,8	27	21,8	48%	18,6	41%
Total	215773,6	205545	197041,6	10228,6	5%	18732	9%

FIGURE 4 – Résultats du test 4

10 Conclusion

Pour conclure, les bancs d'essai nous ont amenés à constater qu'un gain significatif est présent lors d'une charge de travail moyenne. Selon les analyses, lors d'une charge de travail minime, l'ajout des fils d'exécution permet d'obtenir un gain de temps très petit ou négligeable. Il va sans dire que cela contredit notre hypothèse de départ. Par contre, ce gain négligeable semble être proportionnel à la charge de travail. Nous croyons donc qu'avec une charge élevée, par exemple un traitement sur le génome humain, nous pouvons obtenir un gain d'au moins 30%. De plus, OpenMP semble obtenir de meilleurs résultats tout au long des bancs d'essais et nous en favorisons donc l'utilisation.

Étant donné la variation dans la charge de travail il aurait été très acceptable de trouver une diminution du gain lors de petits traitements, qui serait facilement compensée par l'importance du gain obtenu par une charge de travail élevée.

De meilleurs résultats pourraient être obtenus en modifiant un peu le contexte des bancs d'essais. La priorité du processus devrait être élevée pour que le système lui alloue plus souvent du temps CPU. Par ailleurs, l'affinité du processus pourrait aussi être modifiée pour s'exécuter avec un ou plusieurs cœurs dédiés.

Étant donné l'utilisation de données de grande taille, les tests auraient dû être effectués sur le super-ordinateur Mammouth[6] mais pour des raisons administratives, ce fût impossible. Par exemple, il aurait été intéressant de voir l'amélioration sur le traitement d'un génome humain (1.4 Go de données) qui fût impossible à tester sur notre machine (manque de mémoire).

La lecture des fichiers de données influence grandement le temps d'exécution de l'outil, de par le fait qu'il s'exécute séquentiellement. Il serait donc intéressant d'ajouter les fils d'exécution à la lecture, ce qui pourrait diminuer de moitié le temps d'exécution total du test 4 par exemple. De plus, en utilisant OpenMP, tous les tris de l'application pourraient être optimisés grâce à la fonction « ___gnu_parallel : :sort ». Cependant, cette amélioration n'est disponible que pour les compilateurs GNU G++ et MingW.

Considérant le fait qu'OpenMP est implémenté par le compilateur, les fonctions disponibles peuvent être différentes entre les compilateurs. Une autre approche intéressante du point de vue des standards C++11 et qui offre un support pour plusieurs compilateurs est la librairie Threading Building Blocks[5] de Intel®. Cette librairie C++ très portable est destinée pour les systèmes : Windows* ; Linux*, OS X*.

11 Annexes

Algorithme 1– UpdateRange

```
template <class ENTRY_TYPE>
void updateRange(const abstract_dataset<ENTRY_TYPE> *dataset,
                 set_referencefeature &features)
{
    // ASSUMPTION : entry type defines a member "value" which should be
    // part of all dataset type

    typedef typename abstract_dataset<ENTRY_TYPE>::const_iterator chrIt_t;
    typedef typename deque<ENTRY_TYPE>::const_iterator entryIt_t;
    typedef deque<reference_feature>::iterator featureIt_t;

    featureIt_t featurePtr = features.begin();
    bool flag = false;

    // loop for all chromosome node
    for(chrIt_t chrIt = dataset->begin(); chrIt != dataset->end(); ++chrIt)
    {
        // loop for all entries associated to the current chromosome and see
        // if we can find a match
        for(entryIt_t entryIt = chrIt->second.begin();
            entryIt != chrIt->second.end(); entryIt++)
        {
            while(featurePtr != features.end() &&
                  str_insensitive_less_compare(featurePtr->chromosome(),
                                               chrIt->first/*chromosome*/))
                featurePtr++;

            featureIt_t featureIt = featurePtr;
            while(featureIt != features.end() &&
                str_insensitive_equal(featureIt->chromosome(),
                                      chrIt->first/*chromosome*/))
            {
                if(entryIt->end < featureIt->begin())
                    break;

                if(featureIt->isInRange(entryIt->begin, entryIt->end))
                {
                    if(!flag)
                    {
                        flag = true;
                        featurePtr = featureIt;
                    }

                    featureIt->updateRange(entryIt->begin,
                                           entryIt->end,
                                           entryIt->value);
                }

                ++featureIt;
            };

            // reset flag
            flag = false;
        }
    }
}
```

Algorithme 2– UpdateRangeThreads

```
template <class ENTRY_TYPE>
void updateRange(const abstract_dataset<ENTRY_TYPE> *dataset,
                 set_referencefeature &features)
{
    // ASSUMPTION : entry type defines a member "value" which should be part of
    // all dataset type

    /*
     * NOTE from Scott Myer's Effective STL Item 12:
     * Multiple readers are safe. Multiple threads may simultaneously read the
     * contents of a single container, and this will work correctly. Naturally,
     * there must not be any writers acting on the container during the reads.
     *
     * Multiple writers to different containers are safe. Multiple threads may
     * simultaneously write to different containers.
     *
     * When it comes to thread safely and STL containers, you can hope for a
     * library implementation that allows multiple readers on one container and
     * multiple writers on separate containers. You can't hope for the library
     * to eliminate the need for manual concurrency control, and you can't rely
     * on any thread support at all.
     */

    typedef typename abstract_dataset<ENTRY_TYPE>::const_iterator chrIt_t;

    // creates thread args for this entry type
    thread_argument<ENTRY_TYPE> threadArgs[dataset->size()];

    // list of threads(one for each chromosome node(26 max for humans))
    threading::thread_list threadList;
    int argIdx = 0;

    // loop for all chromosome node
    for(chrIt_t chrIt = dataset->begin(); chrIt != dataset->end(); ++chrIt, ++argIdx)
    {
        // set thread arguments
        threadArgs[argIdx].chromosome = chrIt->first;
        threadArgs[argIdx].entries = &chrIt->second;
        threadArgs[argIdx].features = &features;

        // creates and add new thread but do not start it yet.
        // set the function pointer with the proper type as the thread task
        // and set the arguments
        threadList.push_back(threading::thread(updateRangeTask<ENTRY_TYPE>,
                                               (void *) &threadArgs[argIdx]));
    }

    // starts all threads, each thread access a different data structure
    // hence allowing us to run this task concurrently without mutexes.
    threadList.startAll();

    // wait for all thread to terminate. all threads will terminate because
    // of the fixed size of entries for each chromosomes.
    threadList.joinAll();
}
```

Algorithme 3– UpdateRangeThreadsTask

```
template <class ENTRY_TYPE>
void updateRangeTask(void *argument)
{
    // ASSUMPTION : argument is a pointer to "thread_argument"

    // retrieve arguments
    thread_argument<ENTRY_TYPE> *args = (thread_argument<ENTRY_TYPE> *) argument;

    typedef typename deque<ENTRY_TYPE>::const_iterator entryIt_t;
    typedef deque<reference_feature>::iterator featureIt_t;

    featureIt_t featurePtr = args->features->begin();
    featureIt_t featureIt = featurePtr;

    bool flag = false;

    // loop for all entries associated to the current chromosome and see if we
    // can find a match
    for(entryIt_t entryIt = args->entries->begin();
        entryIt != args->entries->end(); entryIt++)
    {
        while(featurePtr != args->features->end() &&
            str_insensitive_less_compare(featurePtr->chromosome(), args->chromosome))
            featurePtr++;

        featureIt = featurePtr;
        while(featureIt != args->features->end() &&
            str_insensitive_equal(featureIt->chromosome(), args->chromosome))
        {
            if(entryIt->end < featureIt->begin())
                break;

            if(featureIt->isInRange(entryIt->begin, entryIt->end))
            {
                if(!flag)
                {
                    flag = true;
                    featurePtr = featureIt;
                }

                featureIt->updateRange(entryIt->begin, entryIt->end, entryIt->value);
            }

            ++featureIt;
        };

        // reset flag
        flag = false;
    }
}
```

Algorithme 4– UpdateRangeOpenMP

```
template <class ENTRY_TYPE>
void updateRange(const abstract_dataset<ENTRY_TYPE> *dataset,
                 set_referencefeature &features)
{
    // ASSUMPTION : entry type defines a member "value" which should be part of
    //                 all dataset type

    /*
     * NOTE from Scott Myer's Effective STL Item 12:
     * Multiple readers are safe. Multiple threads may simultaneously read the
     * contents of a single container, and this will work correctly. Naturally,
     * there must not be any writers acting on the container during the reads.
     *
     * Multiple writers to different containers are safe. Multiple threads may
     * simultaneously write to different containers.
     *
     * When it comes to thread safely and STL containers, you can hope for a
     * library implementation that allows multiple readers on one container and
     * multiple writers on separate containers. You can't hope for the library
     * to eliminate the need for manual concurrency control, and you can't rely
     * on any thread support at all.
     */

    typedef typename abstract_dataset<ENTRY_TYPE>::const_iterator chrIt_t;
    typedef typename deque<ENTRY_TYPE>::const_iterator entryIt_t;
    typedef deque<reference_feature>::iterator featureIt_t;

    /*
     * @see http://www.openmp.org/mp-documents/spec25.pdf P.34
     *
     * Since OpenMP requires the used of relationnal operator such as :
     * > < = <= >= and specifies the for construct to be defined as
     * the following :
     *      for (init-expr; var relational-op b; incr-expr) statement
     * for which the "init-expr" and "var" is a signed integer variable. Hence
     * the usage of iterators is impossible. The following structure, an array
     * of iterators, let us use iterators which are accessed via the array
     * without modifying the algorithm logic and structure.
     */
    int chromosomeCount = dataset->size();
    chrIt_t iterators[chromosomeCount];

    chrIt_t allocateIt = dataset->begin();
    for(int i = 0; i < chromosomeCount; ++i)
        iterators[i] = allocateIt++;

    /*
     * Loops for all chromosomes.
     * Each iteration is done in its own thread. The numbers of threads is set
     * by the number of chromosomes. Also, each thread is assign a static amount
     * of work, more specifically one iteration of the for-loop.
     */
    #pragma omp parallel for num_threads(chromosomeCount)
    #         shared(dataset, features, iterators) schedule(static, 1)
    for(int chrIt = 0; chrIt < chromosomeCount; ++chrIt)
    {
        // makes the following variable private to each thread
        featureIt_t featurePtr = features.begin();
        featureIt_t featureIt = featurePtr;
        bool flag = false;

        // loop for all entries associated to the current chromosome and see
```

```
63          // if we can find a match
64          for(entryIt_t entryIt = iterators[chrIt]->second.begin();
65              entryIt != iterators[chrIt]->second.end(); entryIt++)
66          {
67              while(featurePtr != features.end() &&
68                      str_insensitive_less_compare(featurePtr->chromosome(),
69                                                   iterators[chrIt]->first/*chromosome*/))
70                  featurePtr++;
71
72              featureIt = featurePtr;
73              while(featureIt != features.end() &&
74                  str_insensitive_equal(featureIt->chromosome(),
75                                        iterators[chrIt]->first/*chromosome*/))
76              {
77                  if(entryIt->end < featureIt->begin())
78                      break;
79
80                  if(featureIt->isInRange(entryIt->begin, entryIt->end))
81                  {
82                      if(!flag)
83                      {
84                          flag = true;
85                          featurePtr = featureIt;
86                      }
87
88                      featureIt->updateRange(entryIt->begin,
89                                             entryIt->end,
90                                             entryIt->value);
91                  }
92
93                  ++featureIt;
94              };
95
96              // reset flag
97              flag = false;
98          }
99      }
100 }
```

Algorithme 5– Output

```
void output_builder::outputResults(const std::string &datasetName,
                                   const set_referencefeature &features,
                                   const set_referencefeature &aggregateFeatures,
                                   const parameters_definition &param)
{
    // open output stream for metadatas
    _graphMetaData.open(GRAPH_METADATA_FILENAME, std::ios::out);
    ASSERT(_graphMetaData.is_open(), "File is not open :" << GRAPH_METADATA_FILENAME);

    std::string outGraphFilename,
        outSkrFilename,
        outSkrCurvesFilename,
        outDirectory;

    // files directory
    outDirectory = param.outputDirectory;

    // sorGraph file
    outGraphFilename = string(GRAPH_PREFIX_FILENAME) + "_" + datasetName + "_";
    outSkrFilename = string(OUT_AGG_PREFIX_FILENAME) + "_" + datasetName + "_";

    // loop for all mean
    for(uint i = 0; i < 7; i++)
    {
        // check if current mean must be outputed
        if(param.orientationSubsets[i])
        {
            if(param.oneGraphPerGroup)
            {
                for(uint grpIdx = 0; grpIdx < aggregateFeatures.groupCount(); grpIdx++)
                {
                    // open graph file
                    _outGraph.open((param.outputDirectory + outGraphFilename +
                            aggregateFeatures.groupName(grpIdx) + "_" +
                            graph_orientation::toString(i) + ".txt").c_str(),
                            std::ios::out | std::ios::trunc);
                    ASSERT(_outGraph.is_open(), "File is not open :"
                        << (param.outputDirectory + outGraphFilename +
                        aggregateFeatures.groupName(grpIdx) + "_" +
                        graph_orientation::toString(i) + ".txt").c_str());

                    // open score file
                    _outSkr.open((param.outputDirectory + outSkrFilename +
                        aggregateFeatures.groupName(grpIdx) + "_" +
                        graph_orientation::toString(i) + ".txt").c_str(),
                        std::ios::out | std::ios::trunc);
                    ASSERT(_outSkr.is_open(), "File is not open :"
                        << (param.outputDirectory + outSkrFilename +
                        aggregateFeatures.groupName(grpIdx) + "_" +
                        graph_orientation::toString(i) + ".txt").c_str());

                    _buildOutputGraphInstructions((graph_orientation::mean) i,
                                                  aggregateFeatures,
                                                  grpIdx,
                                                  datasetName,
                                                  outGraphFilename +
                                                    graph_orientation::toString(i),
                                                  outSkrFilename +
                                                    graph_orientation::toString(i),
                                                  param);

                    _outputSmoothing((graph_orientation::mean) i,
```

```
                                aggregateFeatures, 1, param);

                    _outGraph.close();
                    _outSkr.close();
                }
            }
            else
            {
                // open graph file
                _outGraph.open((param.outputDirectory + outGraphFilename +
                    graph_orientation::toString(i) + ".txt").c_str(),
                    std::ios::out | std::ios::trunc);
                ASSERT(_outGraph.is_open(), "File is not open : "
                        << (param.outputDirectory + outGraphFilename +
                        graph_orientation::toString(i) + ".txt").c_str());

                // open score file
                _outSkr.open((param.outputDirectory + outSkrFilename +
                        graph_orientation::toString(i) + ".txt").c_str(),
                        std::ios::out | std::ios::trunc);
                ASSERT(_outSkr.is_open(), "File is not open : "
                        << (param.outputDirectory + outSkrFilename +
                        graph_orientation::toString(i) + ".txt").c_str());

                _buildOutputGraphInstructions((graph_orientation::mean) i,
                                              aggregateFeatures,
                                              -1,
                                              datasetName,
                                              outGraphFilename +
                                                graph_orientation::toString(i),
                                              outSkrFilename +
                                                graph_orientation::toString(i),
                                              param);

                _outputSmoothing((graph_orientation::mean) i,
                        aggregateFeatures, aggregateFeatures.groupCount(), param);

                _outGraph.close();
                _outSkr.close();
            }
        }
    }

    // if an output of each reference feature is required...
    if(param.writeAllReferences)
    {
        std::map<uint, std::vector<const reference_feature *> > featuresAddr;
        for(deque<reference_feature>::const_iterator featureIt = features.begin();
            featureIt != features.end(); ++featureIt)
            featuresAddr[featureIt->groupIndex()].push_back(&(*featureIt));

        //output a file for each group of reference feature
        for(uint grpIdx = 0; grpIdx < aggregateFeatures.groupCount(); grpIdx++)
        {
            outSkrCurvesFilename = string(OUT_ALL_PREFIX_FILENAME) + "_" +
                    datasetName + "_" + aggregateFeatures.groupName(grpIdx);

            _outputSkrCurves(param.orientationSubsets,
                             aggregateFeatures.groupName(grpIdx),
                             grpIdx,
                             aggregateFeatures,
                             featuresAddr[grpIdx],
                             outSkrCurvesFilename,
```

```
126                                     param);
127             }
128         }
129
130         _graphMetaData.close();
131 }
```

Algorithme 6– OutputThreads

```
void output_builder::outputResults(const std::string &datasetName,
                                   const set_referencefeature &features,
                                   const set_referencefeature &aggregateFeatures,
                                   const parameters_definition &param)
{
    // open output stream for metadatas
    _graphMetaData.open(GRAPH_METADATA_FILENAME, std::ios::out);
    ASSERT(_graphMetaData.is_open(), "File is not open : " << GRAPH_METADATA_FILENAME);

    std::string outGraphFilename,
        outSkrFilename,
        outDirectory;

    outDirectory = param.outputDirectory; // files directory

    outGraphFilename = std::string(GRAPH_PREFIX_FILENAME) + "_" + datasetName + "_";
    outSkrFilename = std::string(OUT_AGG_PREFIX_FILENAME) + "_" + datasetName + "_";

    threading::thread_list threadList;
    thread_argument1 threadArgs[7];

    for(uint i = 0; i < 7; i++) // loop for all mean
    {
        if(param.orientationSubsets[i]) // check if current mean must be outputed
        {
            threadArgs[i].mean = (graph_orientation::mean) i;
            threadArgs[i].datasetName = datasetName;
            threadArgs[i].outGraphFilename = outGraphFilename;
            threadArgs[i].outSkrFilename = outSkrFilename;
            threadArgs[i].features = &features;
            threadArgs[i].aggregateFeatures = &aggregateFeatures;
            threadArgs[i].param = &param;

            threadList.push_back(threading::thread(_outputResultsTask1,
                                                   (void *) &threadArgs[i]));
        }
    }

    // if an output of each reference feature is required...
    if(param.writeAllReferences)
    {
        thread_argument2 threadArg;
        threadArg.datasetName = datasetName;
        threadArg.features = &features;
        threadArg.aggregateFeatures = &aggregateFeatures;
        threadArg.param = &param;

        threadList.push_back(threading::thread(_outputResultsTask2,
                                                   (void *) &threadArg));
    }

    // starts all task and wait for all task to finish
    threadList.startAll();
    threadList.joinAll();
    _graphMetaData.close();
}
```

Algorithme 7– OutputThreadsTask1

```
void output_builder::_outputResultsTask1(void *argument)
{
    thread_argument1 *args = (thread_argument1 *) argument;

    filesystem::file outSkr;
    filesystem::file outGraph;

    if(args->param->oneGraphPerGroup)
    {
        for(uint grpIdx = 0; grpIdx < args->aggregateFeatures->groupCount(); grpIdx++)
        {
            // open graph file
            outGraph.open((args->param->outputDirectory +
                args->outGraphFilename +
                args->aggregateFeatures->groupName(grpIdx) + "_" +
                graph_orientation::toString(args->mean) + ".txt").c_str(),
                std::ios::out | std::ios::trunc);
            ASSERT(outGraph.is_open(), "File is not open: "
                << (param->outputDirectory + outGraphFilename +
                aggregateFeatures->groupName(grpIdx) + "_" +
                graph_orientation::toString(i) + ".txt").c_str());

            // open score file
            outSkr.open((args->param->outputDirectory +
                args->outSkrFilename +
                args->aggregateFeatures->groupName(grpIdx) + "_" +
                graph_orientation::toString(args->mean) + ".txt").c_str(),
                std::ios::out | std::ios::trunc);
            ASSERT(outSkr.is_open(), "File is not open : "
                << (param->outputDirectory + outSkrFilename +
                aggregateFeatures->groupName(grpIdx) + "_" +
                graph_orientation::toString(i) + ".txt").c_str());

            _buildOutputGraphInstructions(args->mean,
                                          *args->aggregateFeatures,
                                          grpIdx,
                                          args->datasetName,
                                          args->outGraphFilename +
                                            graph_orientation::toString(args->mean),
                                          args->outSkrFilename +
                                            graph_orientation::toString(args->mean),
                                          *args->param,
                                          outGraph,
                                          outSkr);

            _outputSmoothing(args->mean, *args->aggregateFeatures,
                             1, *args->param, outSkr);

            outGraph.close();
            outSkr.close();
        }
    }
    else
    {
        // open graph file and open score file
        outGraph.open((args->param->outputDirectory + args->outGraphFilename +
            graph_orientation::toString(args->mean) + ".txt").c_str(),
            std::ios::out | std::ios::trunc);
        ASSERT(outGraph.is_open(), "File is not open : "
            << (args->param->outputDirectory + args->outGraphFilename +
            graph_orientation::toString(args->mean) + ".txt").c_str());
        outSkr.open((args->param->outputDirectory + args->outSkrFilename +
```

```
63              graph_orientation::toString(args->mean) + ".txt").c_str(),
64              std::ios::out | std::ios::trunc);
65          ASSERT(outSkr.is_open(), "File is not open: "
66              << (args->param.outputDirectory + args->outSkrFilename +
67              graph_orientation::toString(args->mean) + ".txt").c_str());
68
69          _buildOutputGraphInstructions(args->mean,
70                                        *args->aggregateFeatures,
71                                        -1,
72                                        args->datasetName,
73                                        args->outGraphFilename +
74                                          graph_orientation::toString(args->mean),
75                                        args->outSkrFilename +
76                                          graph_orientation::toString(args->mean),
77                                        *args->param,
78                                        outGraph,
79                                        outSkr);
80
81          _outputSmoothing(args->mean, *args->aggregateFeatures,
82              args->aggregateFeatures->groupCount(), *args->param, outSkr);
83
84          outGraph.close();
85          outSkr.close();
86      }
87  }
```

Algorithme 8– OutputThreadsTask2

```
void output_builder::_outputResultsTask2(void *argument)
{
    thread_argument2 *args = (thread_argument2 *) argument;

    std::string outSkrCurvesFilename;

    std::map<uint, std::vector<const reference_feature *> > featuresAddr;
    for(deque<reference_feature>::const_iterator featureIt = args->features->begin();
        featureIt != args->features->end(); ++featureIt)
        featuresAddr[featureIt->groupIndex()].push_back(&(*featureIt));

    //output a file for each group of reference feature
    for(uint grpIdx = 0; grpIdx < args->aggregateFeatures->groupCount(); grpIdx++)
    {
        outSkrCurvesFilename = std::string(OUT_ALL_PREFIX_FILENAME) + "_" +
            args->datasetName + "_" + args->aggregateFeatures->groupName(grpIdx);

        _outputSkrCurves(args->param->orientationSubsets,
                         args->aggregateFeatures->groupName(grpIdx),
                         grpIdx,
                         *args->aggregateFeatures,
                         featuresAddr[grpIdx],
                         outSkrCurvesFilename,
                         *args->param);
    }
}
```

Algorithme 9– OutputOpenMP

```
void output_builder::outputResults(const std::string &datasetName,
                                   const set_referencefeature &features,
                                   const set_referencefeature &aggregateFeatures,
                                   const parameters_definition &param)
{
    // open output stream for metadatas
    _graphMetaData.open(GRAPH_METADATA_FILENAME, std::ios::out);
    ASSERT(_graphMetaData.is_open(), "File is not open : "
                                                    << GRAPH_METADATA_FILENAME);

    std::string outGraphFilename,
        outSkrFilename,
        outSkrCurvesFilename,
        outDirectory;

    // files directory
    outDirectory = param.outputDirectory;

    // sorGraph file
    outGraphFilename = string(GRAPH_PREFIX_FILENAME) + "_" + datasetName + "_";
    outSkrFilename = string(OUT_AGG_PREFIX_FILENAME) + "_" + datasetName + "_";

    // loop for all mean
    #pragma omp parallel num_threads(8)
    #        shared(datasetName, outGraphFilename, outSkrFilename, features,
    #               aggregateFeatures, param)
    {
        // do not wait until all threads from the team are done
        #pragma omp for nowait schedule(static, 1)
        for(uint i = 0; i < 7; i++)
        {

            filesystem::file outGraph;
            filesystem::file outSkr;

            // check if current mean must be outputed
            if(param.orientationSubsets[i])
            {
                if(param.oneGraphPerGroup)
                {
                    for(uint grpIdx = 0;
                        grpIdx < aggregateFeatures.groupCount(); grpIdx++)
                    {
                        // open graph file
                        outGraph.open((param.outputDirectory +
                            outGraphFilename +
                            aggregateFeatures.groupName(grpIdx) + "_" +
                            graph_orientation::toString(i) + ".txt").c_str(),
                            std::ios::out | std::ios::trunc);
                        ASSERT(outGraph.is_open(), "File is not open :"
                            << (param.outputDirectory + outGraphFilename +
                            aggregateFeatures.groupName(grpIdx) + "_" +
                            graph_orientation::toString(i) + ".txt").c_str());

                        // open score file
                        outSkr.open((param.outputDirectory + outSkrFilename +
                                    aggregateFeatures.groupName(grpIdx) + "_" +
                            graph_orientation::toString(i) + ".txt").c_str(),
                            std::ios::out | std::ios::trunc);
                        ASSERT(outSkr.is_open(), "File is not open : "
                                << (param.outputDirectory + outSkrFilename +
                                aggregateFeatures.groupName(grpIdx) + "_" +
```

```
63                              graph_orientation::toString(i) + ".txt").c_str());
64
65                      _buildOutputGraphInstructions((graph_orientation::mean)i,
66                                                    aggregateFeatures,
67                                                    grpIdx,
68                                                    datasetName,
69                                                    outGraphFilename +
70                                                      graph_orientation::toString(i),
71                                                    outSkrFilename +
72                                                      graph_orientation::toString(i),
73                                                    param,
74                                                    outGraph,
75                                                    outSkr);
76
77                      _outputSmoothing((graph_orientation::mean) i,
78                                  aggregateFeatures, 1, param, outSkr);
79
80                      outGraph.close();
81                      outSkr.close();
82                  }
83              }
84              else
85              {
86                  // open graph file
87                  outGraph.open((param.outputDirectory + outGraphFilename +
88                      graph_orientation::toString(i) + ".txt").c_str(),
89                      std::ios::out | std::ios::trunc);
90                  ASSERT(outGraph.is_open(), "File is not open: "
91                          << (param.outputDirectory + outGraphFilename +
92                          graph_orientation::toString(i) + ".txt").c_str());
93
94                  // open score file
95                  outSkr.open((param.outputDirectory + outSkrFilename +
96                          graph_orientation::toString(i) + ".txt").c_str(),
97                          std::ios::out | std::ios::trunc);
98                  ASSERT(outSkr.is_open(), "File is not open: "
99                          << (param.outputDirectory + outSkrFilename +
100                         graph_orientation::toString(i) + ".txt").c_str());
101
102                 _buildOutputGraphInstructions((graph_orientation::mean) i,
103                                               aggregateFeatures,
104                                               -1,
105                                               datasetName,
106                                               outGraphFilename +
107                                                 graph_orientation::toString(i),
108                                               outSkrFilename +
109                                                 graph_orientation::toString(i),
110                                               param,
111                                               outGraph,
112                                               outSkr);
113
114                 _outputSmoothing((graph_orientation::mean) i,
115                     aggregateFeatures, aggregateFeatures.groupCount(),
116                     param, outSkr);
117
118                 outGraph.close();
119                 outSkr.close();
120             }
121         }
122     }
123
124     // do not wait until all threads from the team are done
125     #pragma omp sections nowait
```

```
126         {
127             // if an output of each reference feature is required...
128             #pragma omp section
129             if(param.writeAllReferences)
130             {
131                 std::map<uint, std::vector<const reference_feature *> > featuresAddr;
132                 for(deque<reference_feature>::const_iterator featureIt = features.begin();
133                     featureIt != features.end(); ++featureIt)
134                     featuresAddr[featureIt->groupIndex()].push_back(&(*featureIt));
135 
136                 //output a file for each group of reference feature
137                 for(uint grpIdx = 0;
138                     grpIdx < aggregateFeatures.groupCount();
139                     grpIdx++)
140                 {
141                     outSkrCurvesFilename = string(OUT_ALL_PREFIX_FILENAME) +
142                                 "_" + datasetName +
143                                 "_" + aggregateFeatures.groupName(grpIdx);
144 
145                     _outputSkrCurves(param.orientationSubsets,
146                                 aggregateFeatures.groupName(grpIdx),
147                                 grpIdx,
148                                 aggregateFeatures,
149                                 featuresAddr[grpIdx],
150                                 outSkrCurvesFilename,
151                                 param);
152                 }
153             }
154         }
155 
156         // implicit barrier, all thread waits here
157     }
158 
159     _graphMetaData.close();
160 }
```

Références

[1] Marcus Geelnard, *TinyThread++ API*, http://tinythreadpp.bitsnbites.eu, page visitée le 04-2013.

[2] *OpenMP API*, http://OpenMP.org/wp, page visitée le 04-2013.

[3] Blaise Barney, Lawrence Livermore National Laboratory, *OpenMP API*, https://computing.llnl.gov/tutorials/openMP/, page visitée le 04-2013.

[4] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, http://www.openmp.org/mp-documents/spec25.pdf, Version 2.5, P. 34, page visitée le 04-2013.

[5] Intel®, *Threading Building Blocks (Intel® TBB) 4.1 Update 3*, http://threadingbuildingblocks.org/, page visitée le 04-2013.

[6] https://rqchp.ca/?pageId=1388, page visité le 04-2013