1 Objectif

Comparer les performances des « langages » de programmation pour la réalisation d'applications de Data Mining.

Lancer un débat à propos du « meilleur langage de programmation » est une excellente manière de plomber une soirée entre informaticiens. La question sous-jacente est « quel est le langage qui permet de développer l'application la plus performante, la plus rapide… ».

De très bon enfant, l'atmosphère devient très vite orageuse, voire délétère. Des personnes, fort charmantes la plupart du temps, adoptent un comportement passionné, voire passionnel, montent sur leurs grands chevaux (tagada, tagada) en assénant des arguments parfois complètement irrationnels. Je sais de quoi je parle, j'en fais partie quand je me laisse aller. Pourtant, finalement, trancher dans ce genre de débat serait assez facile. Il suffit de caractériser les problèmes que l'on cherche à résoudre, écrire un code équivalent dans les différents langages, et étudier le comportement de l'exécutable généré. C'est ce que nous allons faire dans ce didacticiel en nous plaçant dans deux situations couramment rencontrées lors de la programmation d'algorithmes d'exploration de données. On verra que le résultat n'est pas du tout celui qu'on attendait (si on en attendait un, ouh là là je vois déjà certains bondir), loin de là.

Tout d'abord, corrigeons un abus de langage (si je puis dire), la performance n'est pas une affaire de langage, mais plutôt une affaire de technologie et de compilateur. Nous le verrons, le même code source, compilé avec des outils différents, peut aboutir à des exécutables avec des comportements très différents. Nous étudierons dans ce document : C# avec Visual C# Express de Microsoft ; Pascal avec Borland Delphi 6.0 ; Pascal avec le compilateur Free Pascal 2.2.4 de Lazarus 0.9.28 ; C++ avec Borland C++ Builder 4 ; C++ avec Dev C++ (compilateur G++) ; Java exécuté via la JRE1.6.0_19 sous Windows (Eclipse est l'outil de développement que j'ai utilisé). Tous ces outils, excepté Borland C++ Builer 4, sont accessibles gratuitement sur le net. Pour tous, j'ai sélectionné les options de compilations qui optimisent la rapidité d'exécution.

Les performances sont évaluées en mesurant les temps de calculs des exécutables lancés via le shell, en dehors de l'EDI (Environnement de Développement Intégré) pour éviter les interférences. Ma machine étant multi-cœur, temps utilisateur et temps CPU sont quasiment les mêmes. Nous nous contenterons du premier. Chaque programme est lancé 10 fois. Nous calculons la moyenne.

2 Expérimentations

2.1 Tri à bulles

Notre première expérimentation concerne le tri à bulles. L'idée est d'évaluer la capacité à réaliser rapidement de nombreux échanges à l'intérieur d'un vecteur, dans une zone mémoire contigüe. Pour rendre les implémentations comparables, nous avons généré le même vecteur de données de taille **n** dans chaque configuration, à savoir

```
Pour i = 0 to n-1 Faire
     V[i] := i modulo 10
Fin Pour
```

Pour chaque outil, nous avons créé une application console que nous lançons à partir du shell DOS. Le temps de calcul de chaque exécutable est mesuré 10 fois, nous utilisons la commande suivante

```
FOR %%i IN (1 2 3 4 5 6 7 8 9 10) DO CALL exécutable
```

Le chiffre donné dans le récapitulatif est la moyenne des 10 exécutions.

Pour Delphi, nous avons écrit le code suivant :

```
program TriBulles;

{$APPTYPE CONSOLE}

uses
  SysUtils, Windows;

const n : integer = 40000;

var
    t: Cardinal;
    vec: array of double;
    i,j: integer;
    echange: boolean;
    tmp: double;
    nbEchange: integer;

begin
 //remplir le tableau
 setLength(vec,n);
 for i:= 0 to n-1 do vec[i]:= i mod 10;
 writeln('debut des calculs...');

 //mesure
 t:= GetTickCount;

 //tri à bulles, version en Pascal
 echange:= true;
 nbEchange:= 0;
 while (echange = true) do
  begin
   echange:= false;
   for j:= 0 to n-2 do
    begin
     if (vec[j] > vec[j+1])
      then
       begin
        tmp:= vec[j];
        vec[j]:= vec[j+1];
        vec[j+1]:= tmp;
        echange:= true;
        nbEchange:= nbEchange + 1;
       end;
    end;
```

```
  end;
 //mesurer la durée d'exécution
 t:= GetTickCount - t;
 //affichage
 Writeln(format('time = %d ms.',[t]));
 Writeln(format('Nombre echanges = %d',[nbEchange]));
 //libérer la mémoire
 finalize(vec);
end.
```

Comme nous pouvons le noter dans ce code source, nous calculons l'écart entre 2 appels à GetTickCount de la librairie Windows pour mesurer la durée d'exécution. Nous affichons le nombre d'opérations réalisées pour être sûr que les applications effectuent bien les mêmes tâches.

Les projets pour chaque langage de programmation sont disponibles dans l'archive qui accompagne ce document. Le plus important est d'écrire le même code, en respectant les spécificités de chaque langage bien sûr. Pour C# par exemple, nous avons le code suivant :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TriBulles
{
    class Program
    {
        static void Main(string[] args)
        {
            //nombre d'éléments
            int n = 40000;

            //initialiser
            double[] vec = new double[n];
            //remplir
            for (int i = 0; i < n; i++) vec[i] = i % 10;

            //
            Console.WriteLine("Debut des calculs");

            //mesure
            int t = System.Environment.TickCount;

            //tri à bulles
            int j;
            bool echange = true;
            double tmp;

            //compteur d'échanges
            int nbEchange = 0;

            while (echange == true)
            {
                echange = false;
                for (j = 0; j <= n - 2; j++)
                {
                    if (vec[j] > vec[j + 1])
                    {
                        tmp = vec[j];
```

```
                    vec[j] = vec[j + 1];
                    vec[j + 1] = tmp;
                    echange = true;
                    nbEchange = nbEchange + 1;
                }

            }
        }

        //écart
        t = System.Environment.TickCount - t;

        //affichage
        Console.Write("Duree des calcul = "); Console.Write(t);
        Console.WriteLine(" ms.");
        Console.Write("Nombre echanges = "); Console.WriteLine(nbEchange);

    }
  }
}
```

Les résultats à l'issue de l'expérimentation sont recensés dans un tableau.

Compilateur	Temps Exec. (ms.)
Visual C#	4487
Java	4633
C++ Builder	4925
FPC Lazarus	7256
Delphi	7748
DEV C++ (G++)	10299

Ouh là là. Voilà des chiffres qui vont réjouir les uns et énerver les autres. Notons simplement que le code source est accessible en ligne, les outils de développement itou. Tout le monde peut vérifier ces résultats. On ne peut pas dire que ce soit toujours le cas pour ce qui est des publications dans le domaine du data mining. Vous n'obtiendrez pas les mêmes chiffres bien sûr, les machines étant différentes, mais le classement et l'ampleur des écarts (le ratio) seront identiques.

Pour le tri à bulles, C# est le meilleur, suivi de près par Java, C++ Builder n'est pas loin. Viennent ensuite les pascaliens Lazarus (compilateur free pascal) et Delphi. L'écart est conséquent. Manifestement, pour ce type de tâche, les compilateurs FPC et Delphi sont moins efficaces. Enfin, la situation de Dev C++ (compilateur G++) me laisse dubitatif. J'ai bien essayé de paramétrer le compilateur pour optimiser la rapidité d'exécution, mais rien n'y a fait. Deux fois plus lent que Visual C# ou Java, ça me paraît beaucoup quand même.

2.2 Produit croisé – Anti-structure

L'idée est de construire une matrice de produits croisés entre des variables. Ces dernières sont stockées dans une matrice X de taille n x p. Nous souhaitons former une matrice M de taille p x p telle que :

$$m_{j_1,j_2} = \sum_{i=1}^{n} x_{i,j_1} \times x_{i,j_2}$$

Nous avons généré la matrice X de la même manière pour tous les outils, à savoir

```
Pour i = 1 à n Faire
       X [i,j] = (i * j) modulo 2
Fin Pour
```

Une implémentation naturelle consiste à reprendre mot à mot la définition de la matrice M, nous avons le code suivant en C++.

```
#include <cstdlib>
#include <iostream>
#include <windows.h>

using namespace std;

int main(int argc, char *argv[])
{
    int n = 10000;
    int p = 200;

    double **x = new double*[n];
    for (int i = 0; i < n; i++){
     x[i] = new double[p];
     for (int j = 0; j < p ; j++)
        x[i][j] = (i*j) % 2;
    }

    double **mat = new double*[p];
    for (int j = 0; j < p; j++) mat[j] = new double[p];

    cout << "debut des calculs..." << endl;

    long t = GetTickCount();

    int j1, j2;
    int nbOp = 0;

    for (j1 = 0; j1 < p ; j1++)
        for (j2 = 0; j2 < p ; j2++)
            for (int i = 0; i < n ; i++)
            {
                mat[j1][j2] = mat[j1][j2] + x[i][j1] * x[i][j2];
                nbOp = nbOp + 1;
            }

    t = GetTickCount() - t;

    cout << "... fin des calculs" << endl;
    cout << "Temps de calcul = " << t << " ms." << endl;
    cout << "Nombre operations = " << nbOp << endl;

    for (int j = 0; j < p ; j++) delete [] mat[j];
    delete [] mat;
```

```
    for (int i = 0; i < n ; i++) delete [] x[i];
    delete [] x;

    return 0;

}
```

Notre expérimentation donne les résultats suivants :

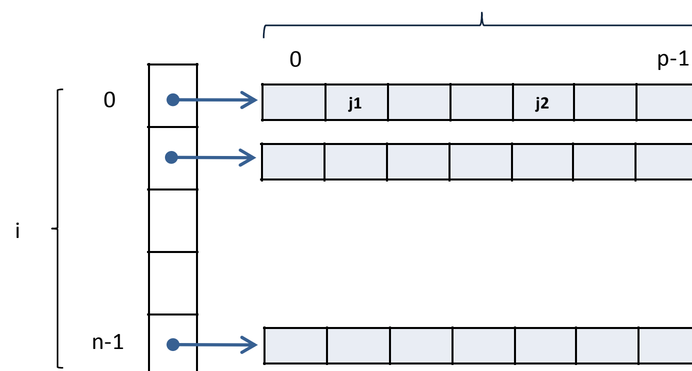
Compilateur	Temps Exec. (ms.)
Delphi	7293
C++ Builder	7408
Visual C#	7936
FPC Lazarus	12133
C++ G++	13432
Java	16271

Mal positionné pour le tri à bulles, Delphi passe en tête maintenant en compagnie de C++ Builder qui a toujours une bonne tenue. Visual C# est juste derrière. Lazarus et G++ sont un peu loin. Le grand perdant cette fois-ci semble être Java qui est deux fois plus lent que les meilleurs. Java souffrirait-il dès qu'il s'agit de manipulation de matrices ? Nous relativisons tout cela dans la section suivante.

2.3 Produit croisé – Pro-structure

La triple boucle imbriquée (en bleu dans le code ci-dessus) est le cœur de l'algorithme lors du calcul de la matrice M. Pour chaque cellule (j1, j2) de la matrice M à compléter, nous traversons totalement la matrice X (i allant de 1 à n) en accédant aux deux colonnes j1 et j2.Il s'agit d'une transcription directe de la définition. Mais est-ce vraiment l'implémentation la plus judicieuse compte tenu de nos structures de données ?

En effet, revenons à la structure de la matrice X, en vert dans le code source ci-dessus. Les données sont organisées selon le schéma suivant :



Dans la boucle intérieure (**FOR i = 0 … n-1**), le passage d'un item à l'autre pour le calcul du produit croisé entre j1 et j2 implique un saut en mémoire qu'il faut calculer. Cette traversée de la matrice X, que je qualifierai de contre nature (d'où le terme « anti-structure » du titre de la section précédente), est répétée 200 x 200 = 40.000 fois. Elle pénalise fortement l'algorithme.

Essayons de réorganiser notre code en tirant parti de la structure ci-dessus. Nous passons la boucle (FOR i) à l'extérieur. De fait, le parcours de la matrice X n'est réalisée qu'une seule fois. Le code (en Java) serait maintenant le suivant.

```
public class CrossProdClass {

      /**
       * @param args
       */
      public static void main(String[] args) {
        int n = 10000;
        int p = 200;

        double[][] x;
        double[][] mat;


        x = new double[n][p];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < p; j++)
                x[i][j] = (i * j) % 2;

        mat = new double[p][p];

        System.out.println("debut des calculs...");

        long t = System.currentTimeMillis();

        int j1, j2;

        int nbOp = 0;

        for (int i = 0; i < n; i++)
            for (j1 = 0; j1 < p; j1++)
                for (j2 = 0; j2 < p; j2++)
                {
                    mat[j1][j2] = mat[j1][j2] + x[i][j1] * x[i][j2];
                    nbOp = nbOp + 1;
                }

        t = System.currentTimeMillis() - t;

        System.out.print("Temps de calcul = "); System.out.print(t);
        System.out.println(" ms.");
        System.out.print("Nb operation = "); System.out.println(nbOp);


      }
}
```

La modification n'est pas anodine du tout. Le nombre total d'opérations n'est pas modifié, il est toujours égal à 400.000.000. A la différence que pour chaque valeur de l'indice i, nous travaillons dans un espace mémoire contigu de la matrice X.

En fait, le problème est déporté sur le parcours de la matrice M. Mais elle ne comporte que P = 200 lignes, et sa traversée (la boucle FOR J1) n'est réalisée que 10.000 fois.

Voyons ce que tout cela donne en termes de temps de calcul selon le langage de programmation.

Compilateur	Temps Exec. (ms.)
C++ Builder	2393
Java	2395
FPC Lazarus	2454
Delphi	2611
Visual C#	4175
C++ G++	4790

Voilà un résultat édifiant. Nous obtenons bien la même matrice M au bout. Mais le temps de calcul est réduit dans des proportions considérables. Le grand gagnant de la réorganisation du code est Java pour lequel la durée d'exécution a été divisée par 6 ! Visual C#, impérial lors de la manipulation d'un seul vecteur dans le tri à bulles, rentre dans le rang ici, même après amélioration du code. Les « sauts » en mémoire ne semblent pas être sa tasse de thé. Reste le cas du compilateur G++ qui me laisse réellement perplexe. Lui aussi a bénéficié de l'amélioration, mais il demeure en dernière position, bien loin de C++ Builder à code source égal. Je pense que j'ai vraiment mal paramétré le compilateur, j'ai bien cherché pourtant. Il y a encore du travail de ce côté-là.

En tous les cas, tenir compte des structures de données (d'où le terme « pro-structure ») est une excellente manière d'améliorer les performances de son programme, quel que soit le langage de développement. Finalement, le principal acteur de la réussite d'une application est le programmeur. Ca paraît évident après coup, mais j'ai l'impression parfois qu'on l'oublie un peu trop vite en reportant la responsabilité de la réussite ou de la contre-performance de l'application sur le langage ou sur la technologie utilisée. On notera d'ailleurs que Java tient parfaitement la route en termes de rapidité d'exécution dans nos expérimentations.

3 Discussion

3.1 Comment optimiser son programme ?

En ce qui me concerne, je divise souvent le temps de calcul par un facteur non négligeable entre le premier jet, reprenant mot à mot un algorithme lu dans un livre, et la version finale, optimisée. Dans quel domaine nous est-il possible de gagner du temps de calcul ?

- Le choix des algorithmes est déjà un élément crucial. Plutôt que de s'énerver à optimiser un tri à bulles, peut être qu'un « quicksort » serait plus approprié non ? L'amélioration des algorithmes est un thème qui dépasse largement le cadre de ce didacticiel, mais il est clair qu'une grande partie (une partie essentielle) de la solution est à ce niveau.
- La connaissance de l'algorithme est aussi un thème à privilégier. La matrice des produits croisés ci-dessus est symétrique, il suffit de calculer la partie triangulaire supérieure (ou inférieure), et reporter les résultats sur l'autre partie. Nous diviserions la durée d'exécution par (à peu près) deux dans ce cas.
- Par un choix judicieux des structures de données. Notre matrice de produits est symétrique disions-nous, pourquoi nous encombrer d'une matrice (p x p) ? Avec une organisation plus judicieuse, nous pouvons réduire l'occupation mémoire.
- Enfin, comme nous avons pu le montrer dans la section précédente, en adaptant notre code à nos structures de données, nous pouvons gagner énormément. Souvent, il s'agit de réduire les

sauts répétés dans des zones mémoires éloignées. Nous pouvons aussi stocker des valeurs intermédiaires pour éviter d'avoir à les recalculer répétitivement.

- Après, on peut gagner encore en s'appuyant sur des astuces plus ou moins spécifiques au compilateur (ex. http://www.delphifr.com/tutoriaux/TACTIQUES-OPTIMISATION-VITESSE-EXECUTION-CODE_755.aspx). Mais cela se fait souvent au détriment de la lisibilité du programme. La maintenance devient difficile. Ca vaut rarement le coup.

Ces principes simples permettent d'améliorer les performances de nos programmes. N'oublions pas cependant que le développement d'une application est tributaire d'une série de contraintes. La rapidité à tout prix n'est pas la seule optique. D'autres critères sont tout aussi importants : l'occupation mémoire, la robustesse, la maintenabilité, etc. Nous ne devons pas les négliger.

3.2 Et Tanagra dans tout ça ?

Tanagra est développé avec Delphi 6. Lors de l'étude préalable en 2003, j'avais réalisé des prototypes avec Delphi, C++ Builder et Java. J'étais arrivé à la conclusion que Delphi convenait très bien, en tous les cas que ce choix ne me pénaliserait pas, ce qui était ma principale inquiétude. En effet, je développe en Pascal depuis près de 20 ans maintenant, même si je ne suis pas « langage dépendant » (je fais des cours entres autres de C++ et de VB ; je travaille – ou j'ai travaillé – sur des projets réalisés en JAVA et en C#), je garde une sympathie certaine pour Delphi.

Même si Delphi 6 n'est plus vraiment « up-to-date » aujourd'hui, il convient parfaitement pour ce que je veux faire : développer rapidement des nouvelles techniques de Data Mining pour pouvoir les évaluer et en parler. Au fil des années, j'ai appris à cerner les humeurs du compilateur. Je sais à peu près ce qu'il faut faire pour optimiser mon code. J'utilise par ailleurs une série d'outils qui me donnent entière satisfaction (FastMM pour la gestion de la mémoire, GpProfile pour le « profiling »). Je ne vois pas trop l'intérêt de courir après les nouvelles versions du compilateur. Enfin, et surtout je devrais dire, j'ai développé ou expérimenté un grand nombre de bibliothèques de calcul dont je connais parfaitement le comportement, et dont le résultat est validé. Repartir à zéro, c.-à-d. développer ces modules dans un nouveau langage, est toujours une opération risquée.

Dernier point important, Tanagra est un outil à vocation pédagogique. Il a été volontairement simplifié pour que sa maintenance soit facilitée (aucune modification structurelle depuis sa création il y a 7 ans), pour que sa diffusion soit aisée (le fichier setup – version 1.4.37 – fait 2.7 Mo, aucune bibliothèque additionnelle n'est installée), pour qu'il puisse fonctionner sur des machines très peu puissantes (l'occupation mémoire est de 3.5 Mo au lancement du programme). Il n'en reste pas moins qu'il propose un niveau de performances tout à fait satisfaisant en termes de rapidité et de capacité à traiter de grandes bases de données. Il surclasse les mastodontes développés par des équipes de programmeurs (je n'ai eu accès qu'aux versions gratuites, je ne saurai jamais comment se comportent les variantes commerciales) en matière d'importation de données et de construction d'arbres de décision (http://tutoriels-data-mining.blogspot.com/2010/12/arbres-sur-les-grands-fichiers-mise.html) ; il ingère mieux les très grandes bases de données en nombre d'observations (une des très grosses limitations des implémentations basées sur Java pour l'instant) (http://tutoriels-data-mining.blogspot.com/2009/10/sipina-traitement-des-tres-grands.html); il est mal à l'aise en revanche lorsqu'il s'agit de traiter les bases très larges (avec un très grand nombre d'attributs) à l'aide de techniques utilisant intensivement le produit scalaire (http://tutoriels-data-mining.blogspot.com/2008/10/svm-comparaison-de-logiciels.html).

4 Conclusion

***And the winner is*... le développeur !!!**

Moralité de tout ceci, nous (programmeurs) sommes les principaux responsables de la qualité de nos applications. Pas la peine de se défausser sur le « langage » ou les outils de développement. Par nos choix de structures, par une écriture judicieuse du code, adapté au problème à résoudre, adapté à la structure utilisée, nous pouvons gagner énormément en performances. Ce n'est pas vraiment une surprise finalement. Mais il est parfois bon de le dire et de le redire.

Après, le choix du « langage » dépend de la politique ou de la culture de la communauté dans laquelle on évolue (du laboratoire, de l'entreprise) ; des choix technologiques (un programme JAVA pseudo-compilé peut être exécuté sur tout type de plate-forme ; un programme en pascal développé sous Lazarus peut être compilé tel quel sur différents OS) ; des possibilités d'évolution de l'outil à moyen et long terme (les louvoiements de Borland et de ses repreneurs autour de l'évolution de Delphi n'ont pas vraiment mis en confiance les développeurs candidats ces dernières années) ; etc. Ces éléments vont largement au-delà de l'affinité personnelle ou de la bêtise cultuelle. Car il y a de tout : certains vénèrent aveuglement microsoft ; d'autres vouent un culte irraisonné au gcc, sous Linux bien évidemment, pas sous cette « grosse daube de Windows »... allons, allons... bon ben gardons nos distances avec tout cela, l'essentiel est ailleurs.

Enfin, le data mining ne se résume pas à des tris et des produits croisés. Les deux configurations étudiées dans ce didacticiel sont loin de couvrir les analyses que l'on peut mener. Elles ont le mérite de montrer que performances riment avec type de problème à résoudre et structure de données utilisées. Le débat est loin d'être clos. J'imagine que les résultats concernant Dev C++ (via le compilateur G++) doivent en énerver plus d'un. Je confesse que ma maîtrise concernant les options du compilateur incriminé reste très rudimentaire. Il faut prendre ce résultat avec beaucoup de prudence. Il ne tient qu'à nous pour approfondir l'idée en étudiant d'autres situations, peut être avec d'autres outils, en faisant varier le volume de données à traiter, en modifiant les paramétrages, etc. Bref, le boulot ne manque pas. En tous les cas, pour que les affirmations soient crédibles, pour que la discussion soit possible, il faut que l'expérimentation décrite soit reproductible par tous. De fait, **la publication du code source est une condition *sine qua non* dès que l'on veut réaliser des comparaisons**.

***Mise à jour (11 février 2011)**. C'était prévisible, ce didacticiel a suscité des commentaires plus ou moins passionnés : « pourquoi il n'a pas utilisé Visual C++ ? », « même pas foutu de mettre les bonnes options de compilation », etc. Je reconnais bien là mes frères informaticiens. Ne comptez pas sur moi pour entrer dans des discussions stériles. Je constate seulement que ces remarquent corroborent totalement mon propos. Le langage de programmation n'est rien sans une bonne connaissance des outils, un paramétrage adéquat, et une programmation qui tire pleinement parti des spécificités du problème traité.*