Annexes

Références

1. **Web :**

- http://genamics.com/developer/csharp_comparative.htm#1
- http://www.25hoursaday.com/CsharpVsJava.html
- http://www.zdnet.fr/builder/commentaires/0,39021559,2137675,00.htm
- http://www.dotnetguru.org/articles/CSharpVsJava.htm
- http://www.extremetech.com/article2/0,1558,11777,00.asp
- http://alain.vizzini.free.fr/cours01.html#IndexRapide
- http://www.pbs.org/cringely/pulpit/pulpit20011101.html
- http://sourceforge.net/projects/dotnetwebserver/
- http://java.sun.com/features/2003/05/bloch_qa.html
- http://msdn.microsoft.com/library/fre/default.asp?url=/library/FRE/cpguide/html/cpovrIntroductionToNETFrameworkSDK.asp
- http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html
- http://alain.vizzini.free.fr/
- http://www.go-mono.org/
- http://www.developpez.com
- http://msdn.microsoft.com/vstudio/downloads/tools/jlca/default.aspx
- http://www.codeproject.com/csharp/C__vs_Java.asp#25
- http://www.softsteel.co.uk/tutorials/cSharp/cIndex.html
- http://alain.vizzini.free.fr/article04.html
- http://www.labo-sun.com/
- http://blogs.labo-dotnet.com/erebuss/archive/2004/04/30/1530.aspx
- http://www.dotnetguru.org/articles/TemplatesInCSharp/TemplatesEnCSharp.htm
- http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/whatsnew-1.5.0.html
- http://www.dotnet-fr.org/documents/andymc_csharp_faq_fr.html
- http://www.zdnet.fr/techupdate/applications/imprimer.htm?AT=2135968-39020852t-39000765c

2. **Livres :**

- Programming C# - O'Reilly
- Formation à C# - Microsoft Press
- Java 2 GrandLivre – Micro application

1er Benchmarck : http://www.tommti-systems.de

Il est intéressant de noter la grosse faiblesse de la VM de Sun pour les opérations mathématiques , et celle du C# pour la gestion des listes...

Test System

- AMD Athlon XP 2200+
- 1024 MB DDR 266
- VIA KT266A
- Windows XP Prof. + SP1

Java Version + Options

- Java 1.4.2_03
- Java 1.5 (alpha, 11. Dec. 03)
- compiler options: -g:none
- running options: -mx1024m

C# Version + Options

- .Net Framework 1.1.4322 (Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4)
- compiler options: /optimize+ /debug- /checked-
- running options:

C++ Version + Options

- Microsoft Visual Studio Net 2003 (Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86)
- compiler options: /O2 /Oi /Og /Ot /EHsc /arch:SSE
- Intel(R) C++ Compiler for 32-bit applications, Version 8.0 Build 20031017Z
- compiler options: /fast /G6 /Oi /Og /Ot /EHsc /arch:SSE
- running options:

Source Code

Original Source Code is from Christopher W. Cowell-Shah http://www.cowell-shah.com/research/benchmark/code and from http://dada.perl.it/shootout/ and Doug Bagley http://www.bagley.org/~doug/shootout . I took portions and whole parts from both and put them into a single file and did some changes + bugfixes.

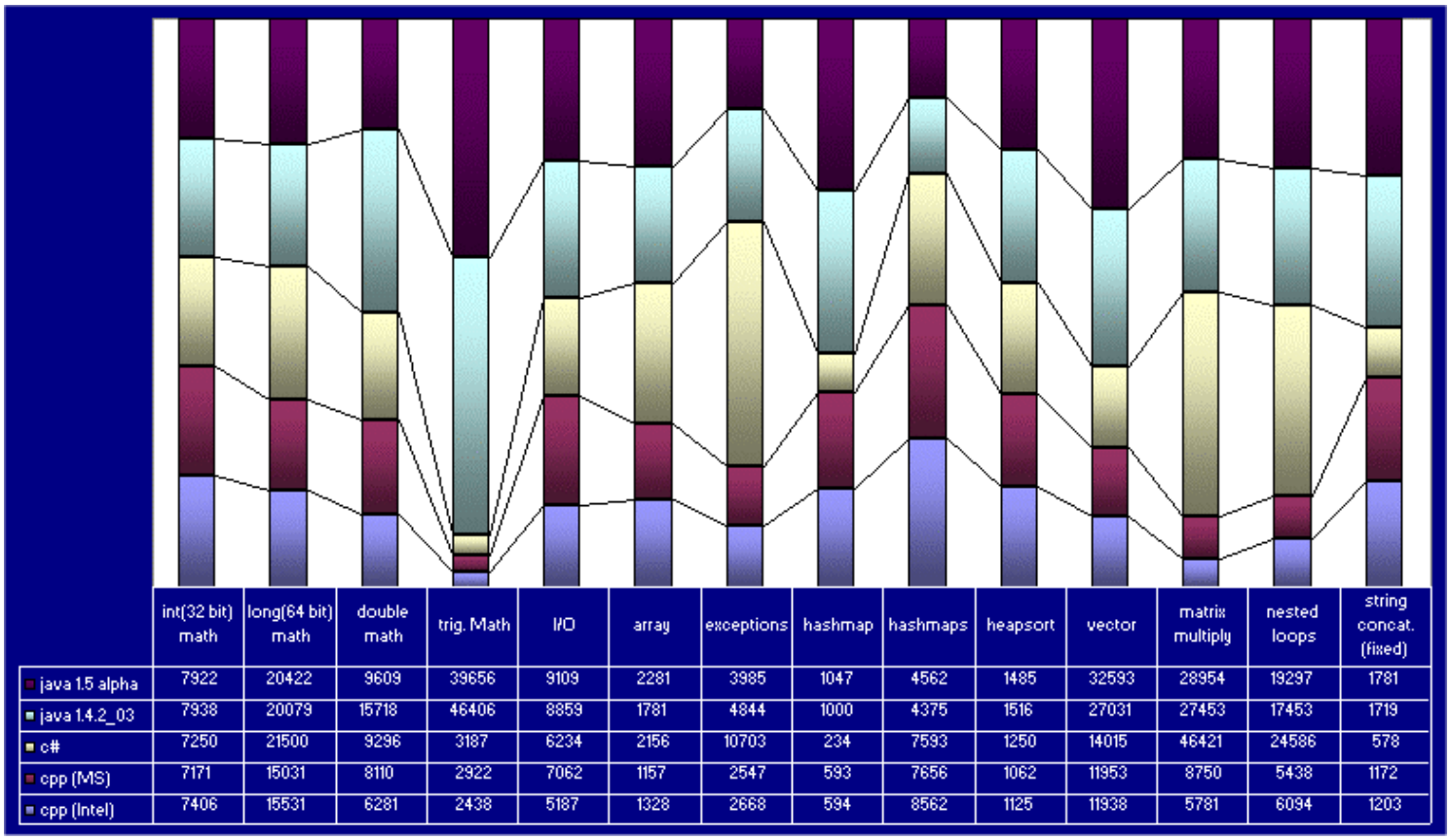
- Java
- C#
- C++

Results

Maximum memory usage:

- Java - 163 MB
- C# - 111 MB
- Cpp - 98 MB

Performance in ms:



	int(32 bit) math	long(64 bit) math	double math	trig. Math	I/O	array	exceptions	hashmap	hashmaps	heapsort	vector	matrix multiply	nested loops	string concat. (fixed)
java 1.5 alpha	7922	20422	9609	39656	9109	2281	3985	1047	4562	1485	32593	28954	19297	1781
java 1.4.2_03	7938	20079	15718	46406	8859	1781	4844	1000	4375	1516	27031	27453	17453	1719
c#	7250	21500	9296	3187	6234	2156	10703	234	7593	1250	14015	46421	24586	578
cpp (MS)	7171	15031	8110	2922	7062	1157	2547	593	7656	1062	11953	8750	5438	1172
cpp (Intel)	7406	15531	6281	2438	5187	1328	2668	594	8562	1125	11938	5781	6094	1203

Cpp is the fastest, except the STL "hashmaps" test is a lot slower, maybe someone can take a look at the source code, I use STL <map>, because <hash_map> was slower.... The memory footprint of Cpp is also the lowest, which was expected. C# is quit fast (I would have been surprised, if C# were much slower than Java), but seems to have some problems with exception handling, matrix multiply and nested loops. Java's hash maps are very fast and the exception handling is also very strong. If you sum it up, Java gets 5 wins against C#, C# gets 9 wins against Java and Cpp gets 11 wins against C#. This black and white comparison only takes performance into account, but other values like development speed and security are also very important issues today. In my opinion it goes much faster to develop complex apps with Java or C#, than with plain Cpp and the STL. Type safety and buffer overrun issues are also much weaker in Cpp, so you have to do more work, which takes more time and will cost some speed! So every language has its strong areas, but as always, nothing comes for free...

Update 15.04.04

Java Version + Options

- Java 1.5 (alpha, 11. Dec. 03)
- compiler options: -g:none
- running options: -mx1024m -Xbatch and -server for the server VM

C# Version + Options

- .Net Framework 1.1.4322 (Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4)
- compiler options: /optimize+ /debug- /checked-
- running options:

C++ Version + Options

- Microsoft Visual Studio Net 2003 (Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86)
- compiler options: /Op /Oy /O2 /Oi /Og /Ot /EHsc /arch:SSE
- Intel(R) C++ Compiler for 32-bit applications, Version 8.0 Build 20031017Z
- compiler options: /Op /Oy /Qpc80 /fast /G6 /Oi /Og /Ot /EHsc /arch:SSE
- running options:

Source Code Changes

C++ :SLT VECTOR was replaced with the STL LIST, that's why the benchmark is now called "List". STDIO(C) was replaced with FSTREAM(C++).
Java :STRINGBUFFER was replaced with STRINGBUILDER, timeing routines changes, "new Integer()" was replaced with Integer.valueOf(), VECTOR was replaced with LINKEDLIST, so all "List" benchmarks use now a list, only C# uses its ARRAYLIST, because there is no list or linked list in C# and thats the reason, why these benchmark is marked red!

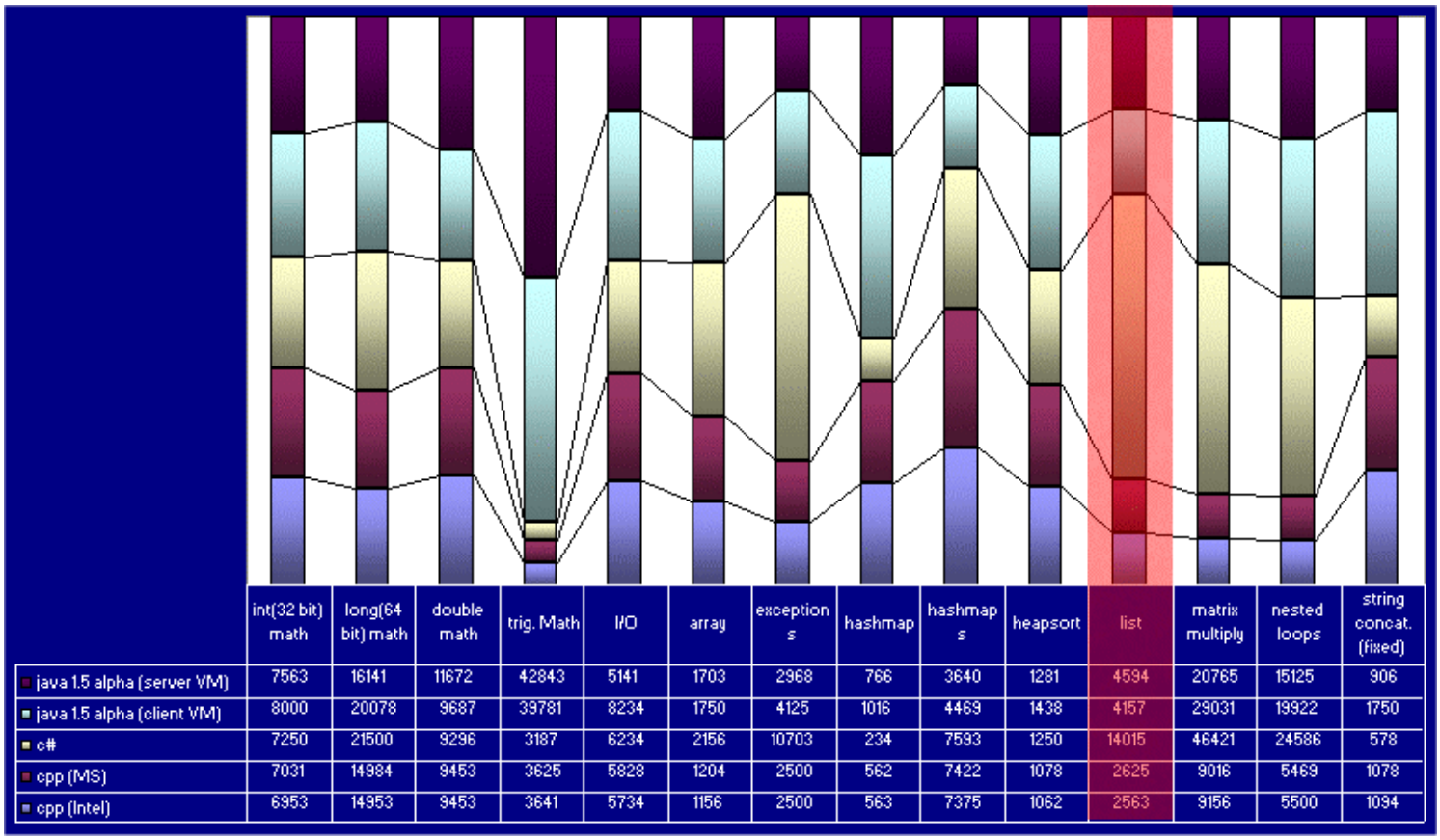
- Java
- C++

Results

Maximum memory usage:

- Java - 163 MB
- C# - 111 MB
- Cpp - 98 MB

Performance in ms:



	int(32 bit) math	long(64 bit) math	double math	trig. Math	I/O	array	exceptions	hashmap	hashmaps	heapsort	list	matrix multiply	nested loops	string concat. (fixed)
java 1.5 alpha (server VM)	7563	16141	11672	42843	5141	1703	2968	766	3640	1281	4594	20765	15125	906
java 1.5 alpha (client VM)	8000	20078	9687	39781	8234	1750	4125	1016	4469	1438	4157	29031	19922	1750
c#	7250	21500	9296	3187	6234	2156	10703	234	7593	1250	14015	46421	24586	578
cpp (MS)	7031	14984	9453	3625	5828	1204	2500	562	7422	1078	2625	9016	5469	1078
cpp (Intel)	6953	14953	9453	3641	5734	1156	2500	563	7375	1062	2563	9156	5500	1094

Cpp is a little bit slower with more accurate floating point calculations and omitted frame pointers. The Java server VM is faster than the client VM, which was new to me, because I don't have that much Java experience. Trig. functions and maybe the memory footprint are the only weak points in Java today. The "hashmaps" benchmark uses a helper class in Java, to achive this high performance. If you don't use it, you will get 11000ms instead of 3640ms, but this optimisation isn't possible and necessary for the C++ and C# versions!

Update 17.04.04

Java Version + Options

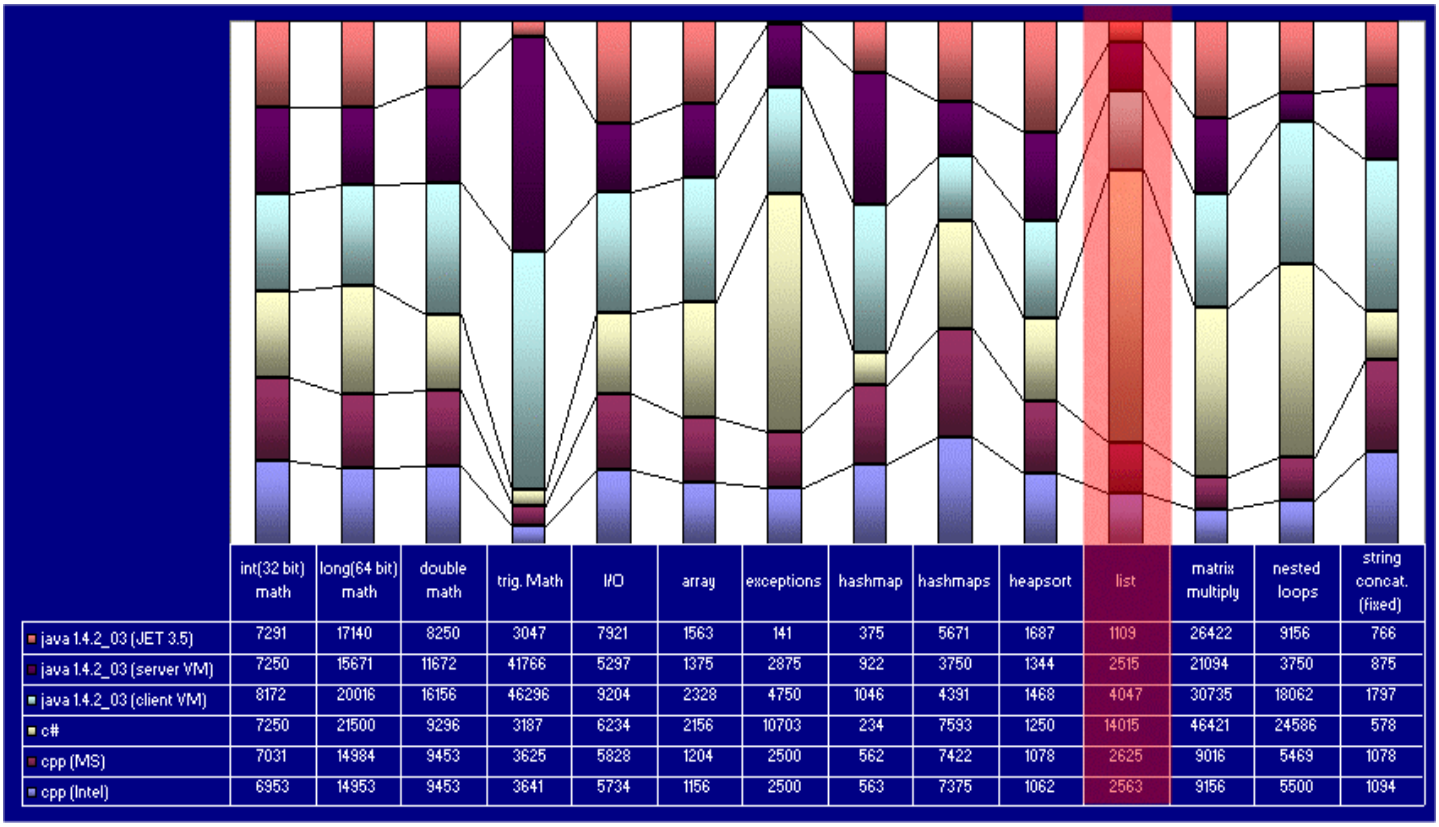
- Java 1.4.2_03
- compiler options: -g:none
- running options: -mx1024m -Xbatch and -server for the server VM
- Excelsior JET 3.5

Results

Maximum memory usage:

- Java - 163 MB
- C# - 111 MB
- Cpp - 98 MB

Performance in ms:



	int(32 bit) math	long(64 bit) math	double math	trig. Math	I/O	array	exceptions	hashmap	hashmaps	heapsort	list	matrix multiply	nested loops	string concat. (fixed)
java 1.4.2_03 (JET 3.5)	7291	17140	8250	3047	7921	1563	141	375	5671	1687	1109	26422	9156	766
java 1.4.2_03 (server VM)	7250	15671	11672	41766	5297	1375	2875	922	3750	1344	2515	21094	3750	875
java 1.4.2_03 (client VM)	8172	20016	16156	46296	9204	2328	4750	1046	4391	1468	4047	30735	18062	1797
c#	7250	21500	9296	3187	6234	2156	10703	234	7593	1250	14015	46421	24586	578
cpp (MS)	7031	14984	9453	3625	5828	1204	2500	562	7422	1078	2625	9016	5469	1078
cpp (Intel)	6953	14953	9453	3641	5734	1156	2500	563	7375	1062	2563	9156	5500	1094

The server VM (1.4.2_03) is faster than the 1.5(alpha) server VM from above and it's always faster than its client version. The JET 3.5 code is faster than the client VM, but sometimes it is slower than the server VM. Trig. functions are as fast as the Cpp and the C# code, but they seems to use some lower precision for the sin, cos and tan functions.

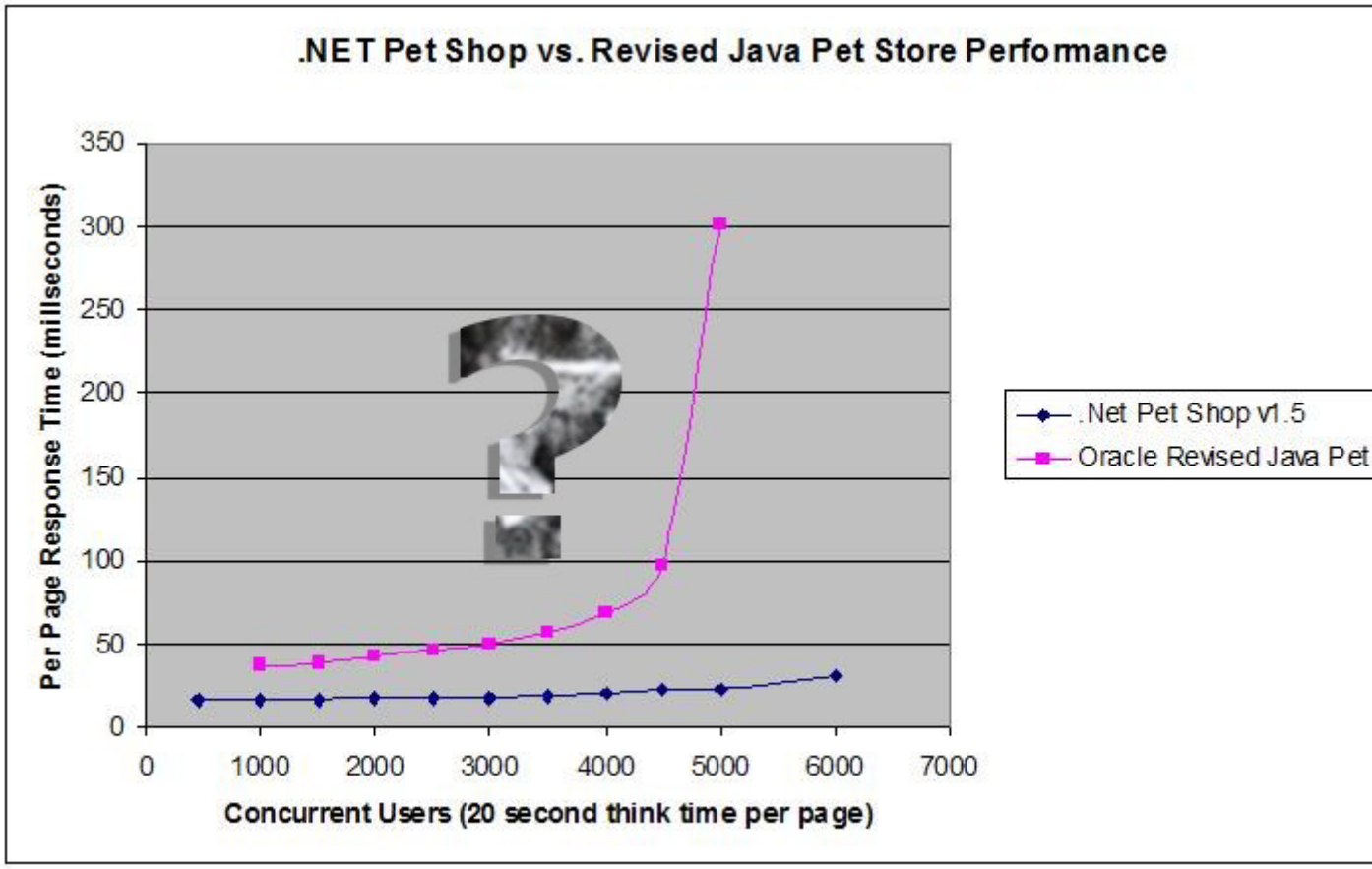
2 ème Benchmark :
http://www.dotnetguru.org/articles/Comparatifs/benchJ2EEDotNET/J2EEvsNETBench.html

.NET en moyenne 2 à 3 fois plus rapide que J2EE ! *par Thomas GIL (thomas.gil@valtech.fr)*

Table des matières

Introduction

Une adaptation .NET réalisée par Microsoft de l'application phare J2EE, le **PetStore**, s'est avérée **10 fois plus performante** que la même application écrite en Java et testée sur un serveur d'application J2EE (Oracle en l'occurrence). Le graphique suivant le prouve !



Cette déclaration tonitruante, publiée sur le site http://www.gotdotnet.com/team/compare/veritest.aspx, a déjà fait couler beaucoup d'encre. Fidèles à notre habitude, nous avons voulu nous faire notre propre opinion en effectuant un Benchmark consistant à reprendre les principaux éléments de l'architecture J2EE et en les testant avec .NET. Vous trouverez donc dans cet article : des algorithmes écrits en Java et C#, de petites applications graphiques en JavaSwing et WinForms, des WebServices et l'utilisation des middlewares .NET Remoting, RMI sur IIOP (Corba) et RMI sur JRMP (Le protocole de Sun).

Au fur et à mesure que nous implémentions ces tests, nous nous sommes rendus compte de l'étendue du travail et de l'importance des résultats auxquels nous étions confrontés. Nous publierons donc certainement un deuxième article, complémentaire à celui-ci, dont l'objectif sera de tester les autres composantes des plateformes .NET et J2EE telles que les sites Web, les serveurs d'applications (Serviced Component et EJB), les outils du monde XML, XSL, etc ... Mais pour l'heure, nous allons comparer les fondements des langages et des bibliothèques (Framework) des mondes C#.NET et Java/J2EE.

Nous avons réalisé ces tests de la manière la plus objective possible, en essayant d'user de toutes les ficelles, trucs et astuces de J2EE et .NET afin de tirer le meilleur de chaque monde. Toutefois, il se peut que nous soyons passés à côté d'une optimisation importante qui fausserait la lecture de nos résultats. Nous vous invitons donc à réagir à cet article et à nous aider à rendre ce comparatif le plus neutre possible.

Conditions d'expérimentations, méthodologie

Tous les tests de cet article sont réalisés sur un ordinateur banalisé doté d'un processeur **Intel PIII** de fréquence **1,133 Ghz**, de **512 Mo** de RAM, et du système d'exploitation **Windows 2000 Professional Service Pack 2**. Le but n'est pas de mettre en place un laboratoire complexe basé sur des serveurs multiprocesseurs, mais de fournir des résultats conformes à une plateforme classique et surtout identique pour .NET et J2EE.

Les programmes Java fonctionnent sur la machine virtuelle de Sun Microsystems en version 1.4.1 (**SDK 1.4.1**) pour Windows; les applications .NET sur le CLR sont en version **1.0.3705**.

Langages de programmation

Objectifs

Prenons un algorithme type, et essayons de l'implémenter à la fois en C# et en Java.

Tableaux

Algorithme

Remplissons un tableau de n éléments numériques (double), et effectuons le calcul du minimum de ces nombres. L'implémentation Java est triviale :

```
public final class TestTableaux {
    private double[] valeurs;
    public  TestTableaux(int  nombreElements) {
        valeurs = new double[nombreElements];
        for (int i=0; ipublic double min(){
        double minimum = Double.MAX_VALUE;
        double longueur = valeurs.length;
        double valeurCourante = 0;
        for (int i=0; iif  (valeurCourante < minimum){
                minimum = valeurCourante;
            }
        }
        return minimum;
    }
    public long testMin(int nbItérations){
        long start = System.currentTimeMillis();
        for (int i=0; ireturn  (end - start);
    }
    public static void main(String[] args) {
        if (args.length < 2){
            System.out.println("Usage : TestTableaux nbValeursDsTableau
nbIterationsPourMoyenne");
            return;
        }
        TestTableaux tt = new  TestTableaux(Integer.parseInt(args[0]));
        System.out.println("Temps écoulé : " +
tt.testMin(Integer.parseInt(args[1])));
    }
}
```

TestTableaux.java

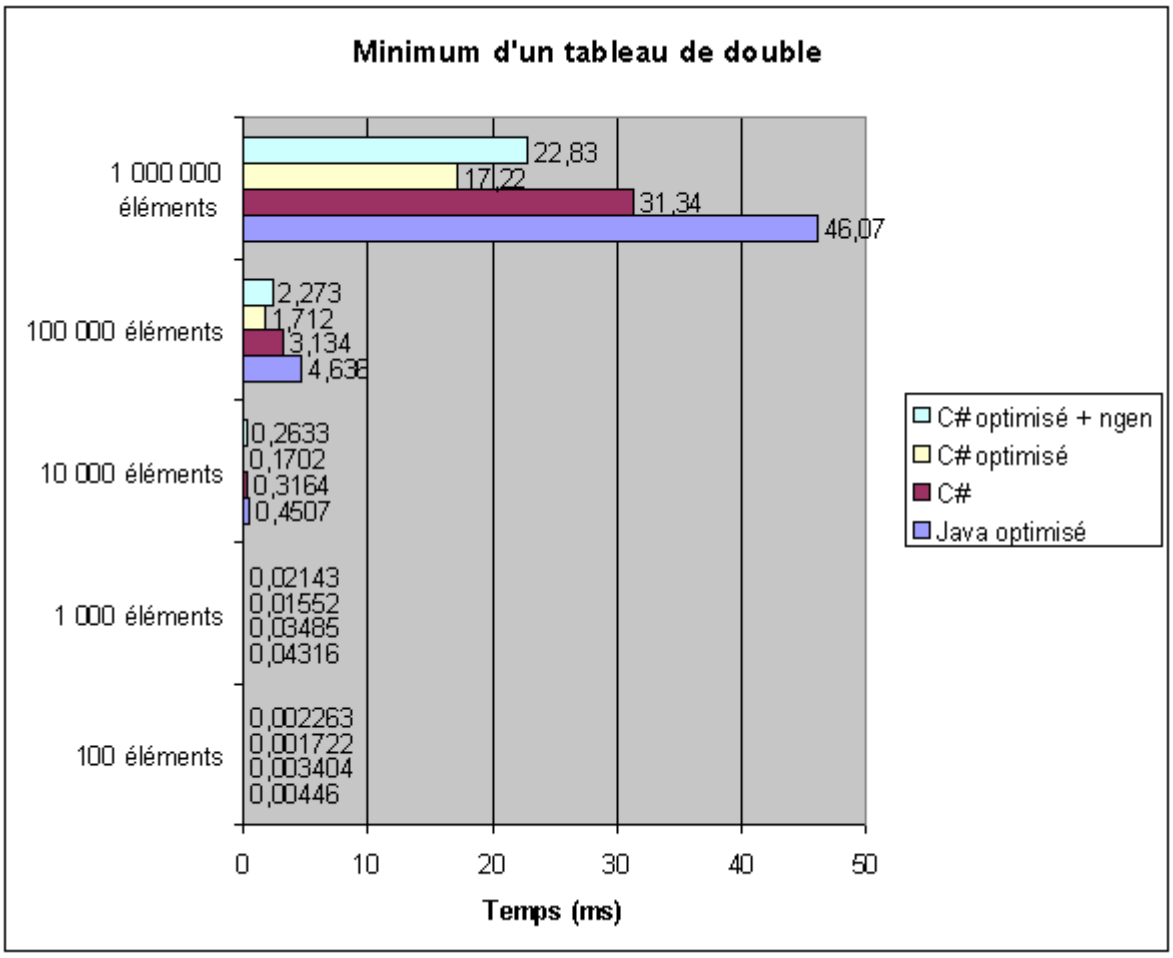
Le code C# équivalent ressemble énormément au code Java :

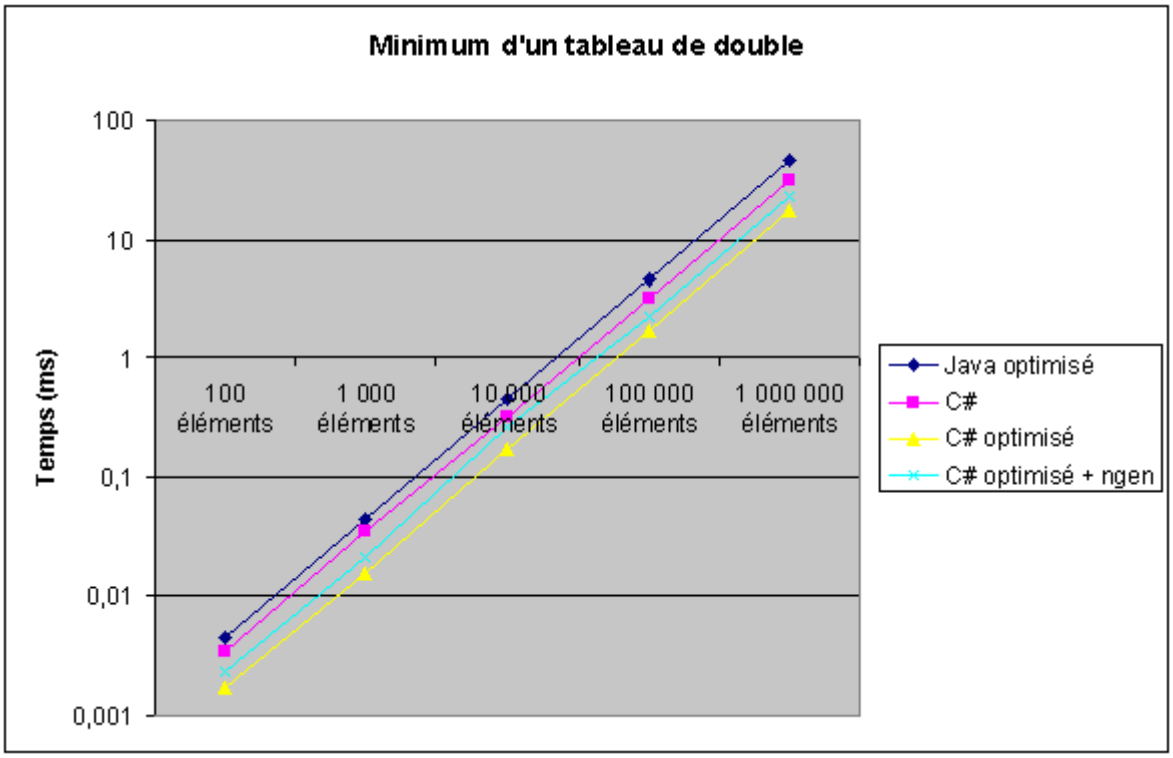
```
using System;
public sealed class TestTableaux
{
    private double[] valeurs;
    public  TestTableaux(int  nombreElements)
    {
        valeurs = new double[nombreElements];
        Random r = new  Random();
        for (int i=0; ipublic double min()
    {
        double minimum = Double.MaxValue;
        double longueur = valeurs.Length;
        double valeurCourante = 0;

        for (int i=0; iif  (valeurCourante < minimum)
            {
                minimum = valeurCourante;
            }
        }
        return minimum;
    }

    public double  testMin(int nbItérations)
    {
        DateTime start = DateTime.Now;
        for (int i=0; ireturn  ts.TotalMilliseconds;
    }
    public static void Main(String[] args)
    {
        if  (args.Length < 2)
        {
            Console.WriteLine
                ("Usage : TestTableaux nbValeursDsTableau
nbIterationsPourMoyenne");
            return;
        }
        TestTableaux tt = new  TestTableaux(int.Parse(args[0]));
Console.WriteLine("Temps écoulé : " + tt.testMin(int.Parse(args[1])));
    }
}
```

TestTableaux.cs

Résultats

Nous avons compilé le code Java en mode optimisé (*javac -O*). Quant à C#, nous nous sommes attachés à le compiler d'abord en mode normal, puis en intégrant les optimisations pour enfin appliquer l'outil **ngen** sur l'exécutable. Chaque exécutable a été testé en faisant varier la taille des tableaux parcourus de 100 éléments à 1 000 000. Les résultats sont donc présentés en mode normal et en mode logarithmique :





Les performances pures de C# sont environ **2.7 fois meilleures** que celles de Java pour une manipulation "bas niveau" d'un tableau de numériques. Le plus inquiétant, c'est que ce facteur est quasiment le même quelle que soit la taille du tableau en question. Il est également à noter que l'optimisation par le compilateur C# a un impact notable sur les performances (**quasiment 50%**), alors que l'utilisation de **ngen** nuit à l'exécution de notre petit programme (**perte d'environ 25 % des performances**) !

Listes, boxing et unboxing

Algorithme

Remplissons une collection de n éléments numériques (double), et effectuons le calcul du minimum de ces nombres.

Comme précédemment, l'implémentation Java est relativement simple. Notez toutefois qu'il est du ressort du développeur, en Java, d'effectuer le boxing / unboxing lors du passage d'un type primitif (un **double** ici) stocké sur la pile à un type objet (**Double**) stocké sur le tas :

```
import java.util.*;
public final class TestListes {
    private List valeurs;
    public TestListes(int nombreElements) {
        valeurs = new  ArrayList();
        for (int i=0; inew Double((Math.random() - 0.5) *
Double.MAX_VALUE));
        }
    }
    public double min(){
        double minimum = Double.MAX_VALUE;
        double valeurCourante = 0;
        Iterator i = valeurs.iterator();
        while (i.hasNext()){
            valeurCourante = ((Double) i.next()).doubleValue();
            if (valeurCourante < minimum){
                minimum = valeurCourante;
            }
        }
        return minimum;
    }

    public long testMin(int nbItérations){
        long start = System.currentTimeMillis();
        for (int i=0; ireturn (end - start);
    }
    public static void main(String[] args) {
        if (args.length < 2){
            System.out.println
                ("Usage : TestListes nbValeursDsTableau
nbIterationsPourMoyenne");
            return;
        }
        TestListes tl = new TestListes(Integer.parseInt(args[0]));
        System.out.println("Temps écoulé : " +
tl.testMin(Integer.parseInt(args[1])));
    }
}
```

TestListes.java

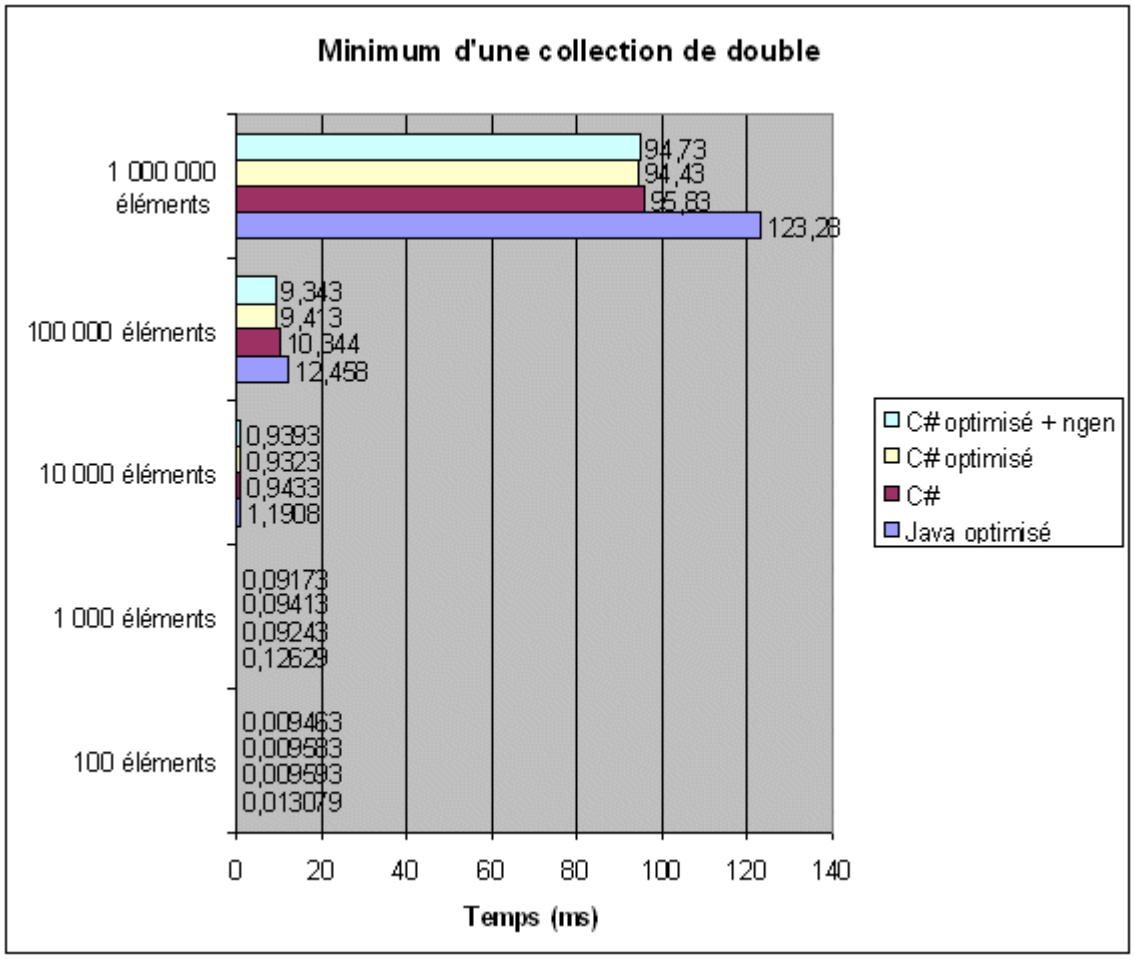
Le code C# équivalent est notablement plus simple au niveau du boxing / unboxing car ce langage traite ce problème de manière transparente :

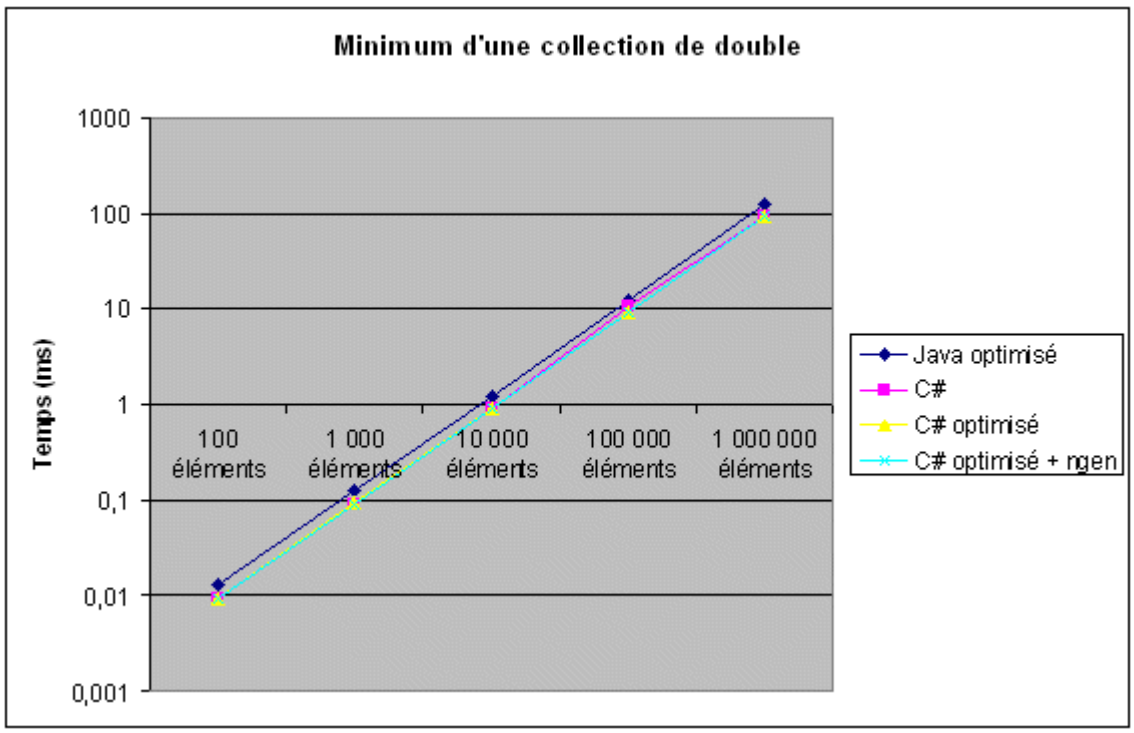
```
using System;
using System.Collections;

public sealed class TestListes
{
    private IList valeurs;
    public TestListes(int nombreElements)
    {
        valeurs = new System.Collections.ArrayList();
        Random r = new Random();
        for (int i=0; ipublic double min()
    {
        double minimum = Double.MaxValue;

        foreach (double valeurCourante in valeurs)
        {
            if (valeurCourante < minimum)
            {
                minimum = valeurCourante;
            }
        }
        return minimum;
    }

    public double testMin(int nbItérations)
    {
        DateTime start = DateTime.Now;
        for (int i=0; ireturn ts.TotalMilliseconds;
    }
    public static void Main(String[] args)
    {
        if (args.Length < 2)
        {
            Console.WriteLine("Usage : TestTableaux nbValeursDsTableau
nbIterationsPourMoyenne");
            return;
        }
        TestListes tl = new TestListes(int.Parse(args[0]));
        Console.WriteLine("Temps écoulé : " +
tl.testMin(int.Parse(args[1])));
    }
}
```

TestListes.cs

Résultats

Nous avons procédé aux mêmes compilations et optimisations automatiques que dans le test précédent. A nouveau, les résultats sont saisissants :





Même s'ils sont plus mitigés que précédemment : ces résultats démontrent que Java tire son épingle du jeu pour de petits volumes de listes (jusqu'à 10 000 éléments), pour lesquels ses performances sont **meilleures que celles de C# d'environ 3 %**. Autant dire que les deux langages se valent quasiment à ce niveau-là. Par contre, dès que le volume devient important (100 000 éléments et plus), les performances pures de Java deviennent dramatiques : à 1 000 000 éléments, **C# va 30% plus vite que Java** pour déterminer le minimum de la liste. A noter que la taille du tas alloué par défaut par la JVM et le CLR n'a pas été modifié car ces volumes ne nécessitent en théorie aucun Swap, cela aurait été pertinent dans le cadre d'une dizaine de Millions d'éléments.

Clients lourds (Couche de Présentation)

Méthodologie, objectifs

Il nous est apparu trop complexe ou trop subjectif d'établir un comparatif global sur les applications graphiques utilisateur. En effet, comment quantifier la fluidité, les temps de réponse d'une IHM ? Donc nous nous sommes contentés ici de développer la même application en C# - Windows Forms d'un côté, et en Java - Swing de l'autre, et de mesurer le temps de lancement de l'application, c'est-à-dire la durée qui s'écoule entre le déclenchement et l'apparition à l'écran de notre application.

Description de l'application

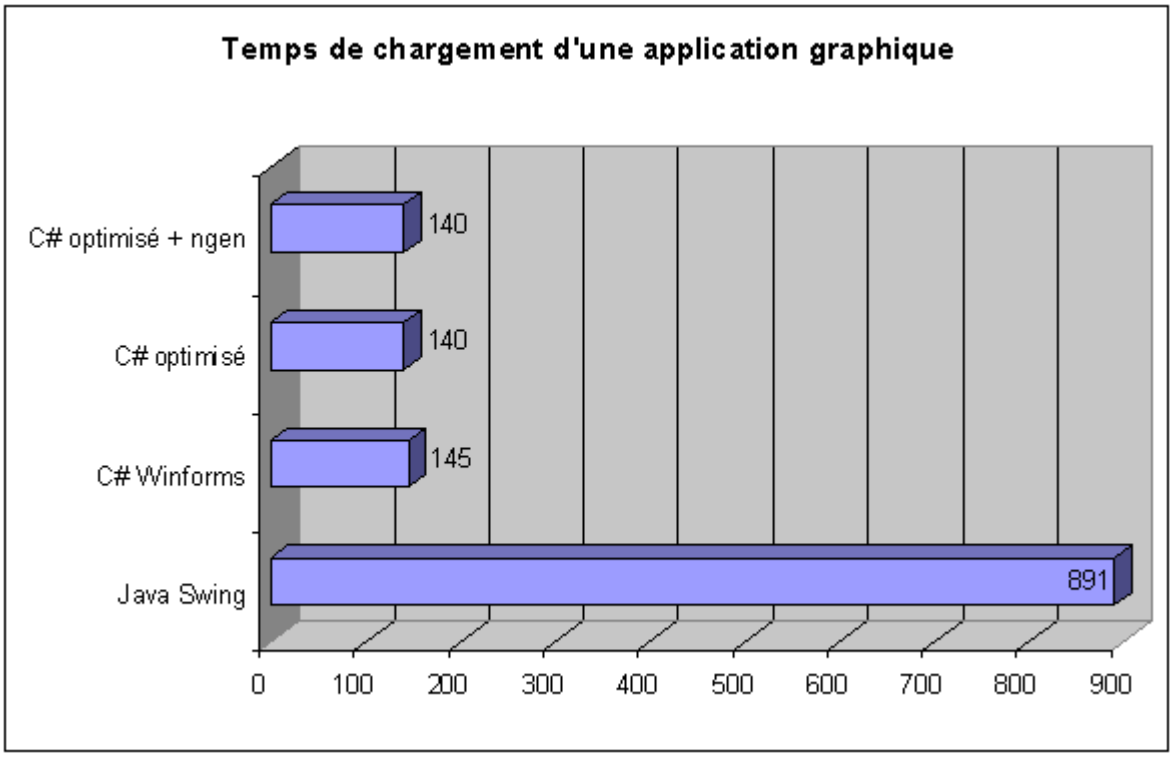
Nous voulions une application à la fois simple, et représentative des possibilités offertes par les deux bibliothèques (WinForms et Swing). Nous avons choisi de développer un système simple de gestion du personnel de DotNetGuru : chaque Guru y est enregistré, photo comprise. Voilà le résultat en Java Swing :

Et l'équivalent en Windows Forms :

Résultats

Le chargement de l'application graphique n'est qu'un paramètre, direz-vous, mais c'est la première impression que laisse une application graphique. Développeurs Java, combien de fois avez-vous pesté contre NetBeans ou JBuilder qui mettent souvent plusieurs dizaines de secondes à apparaître à l'écran ?

Quoi qu'il en soit, notre petite application ne laisse aucune équivoque : l'application graphique **C# WinForms apparaît à l'écran 6,4 fois plus vite que son équivalent Java Swing** comme en témoigne le graphique suivant :



Nous avons toutefois été étonnés de découvrir que, encore une fois, l'utilisation de **ngen** ne changeait pas les performances globales de notre application.

WebServices

Méthodologie, hypothèses

Dans une relation client-serveur utilisant les WebServices, il est très difficile d'affirmer quelles sont les activités les plus consommatrices de temps : s'agit-il du pliage / dépliage (marshalling / unmarshalling) entre la représentation mémoire des données et la représentation en XML, ou de l'initialisation de la connexion réseau ? Passe-t-on le plus clair de son temps dans les couches réseau, ou au contraire dans le code métier ?

Plutôt que d'être exhaustifs, nous avons préféré mesurer les performances globales du même algorithme invoqué par le biais des WebServices. C'est-à-dire que nous avons mesuré le temps total entre l'invocation du service (à travers un proxy) et l'obtention du résultat par le client. Voyons le détail des activités que cela comprend :

- marshalling des données passées en paramètre de l'invocation du service en XML
- envoi de la requête SOAP au service visé, avec les paramètres XML
- réception par le service de la requête SOAP
- unmarshalling des paramètres XML en mémoire côté serveur
- invocation de la méthode côté serveur, récupération des paramètres résultants
- marshalling des résultats en XML (résultat + paramètres out et in-out)
- envoi de la réponse SOAP au proxy qui a sollicité le service
- réception des résultats XML par le proxy
- unmarshalling des résultats XML en mémoire côté client

L'algorithme que nous avons choisi est exactement le même que lors de la comparaison des langages de programmation : le calcul du minimum dans un tableau de double. Nous avons donc procédé à son implémentation sur le framework .NET sous la forme d'un fichier ASMX et de son CodeBehind :

```
using  System.ComponentModel;
using  System.Web;
using  System.Web.Services;

namespace DngServices
{
    public class Calculateur : System.Web.Services.WebService
    {
        private IContainer components = null; [WebMethod]
        public double min(double[] valeurs)
        {
            double minimum = double.MaxValue;
            double longueur = valeurs.Length;
            double valeurCourante = 0;

            for (int i=0; iif  (valeurCourante < minimum)
```

```
                {
                    minimum = valeurCourante;
                }
            }
            return minimum;
        }
    }
}
```

DotNetService.asmx.cs

Nous avons ensuite également transposé cet exemple en Java sur le produit **Apache Axis**. Il existe deux choix de déploiement avec ce produit : soit sous la forme d'un simple fichier texte (contenant du code source Java, et possédant l'extension JWS), soit sous la forme d'une classe précompilée que l'on appelle un "Handler" dans Axis, ou un "Servant" dans la terminologie JAX-RPC. Nous avons donc implémenté et testé les deux, mais le code Java reste le même que dans les exemples précédents.

Enfin, vus les résultats de Axis (vous comprendrez dans quelques secondes), nous avons décidé de donner une seconde chance à Java en implémentant toujours le même algorithme sur le produit **Glue** (www.themindelectric.com). Pour être le plus efficace possible, nous avons même opté pour l'option de déploiement "intégrée" de glue : le processus du serveur de WebServices héberge un mini serveur HTTP, ce qui minimise les communications et les couches côté serveur. Le code est très simple, jugez par vous-même :

```
public interface IGlueMin{
    public double min(double[] tab);
}

/**************************/

import  electric.registry.Registry;
import  electric.server.http.HTTP;

public class GlueMinImpl implements
IGlueMin{
    public double min(double[] tab){
    // Idem précédemment
    }

    public static void main(String[]
argv) throws Exception{ HTTP.startup( "http://localhost:8000/glue" );
Registry.publish( "min", new  GlueMinImpl() );
    }
}
```

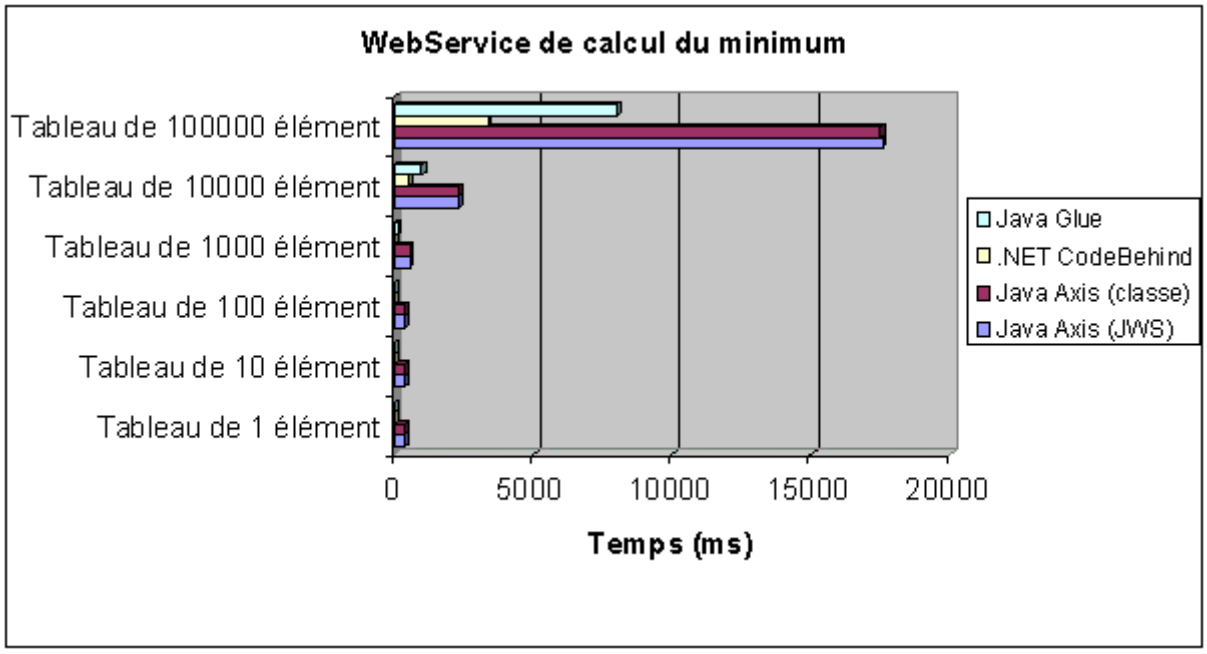
GlueMinImpl.java

Résultats

Bien entendu, pour tester nos deux WebServices, nous avons utilisé le même client (écrit en C#) pour avoir la garantie de tester seulement les performances du

Invocation du service (ms)	Java Axis (JWS)	Java Axis (classe)	.NET CodeBehind	Java Glue
Tableau de 1 élément	358,5	354,5	4,5	5,5
Tableau de 10 éléments	360,5	357,5	4,5	6
Tableau de 100 éléments	368,5	365,5	8	11
Tableau de 1000 éléments	547,7	541,4	38	115,1
Tableau de 10000 éléments	2300,3	2285,2	511,7	970,3
Tableau de 100000 éléments	17557,2	17478	3374,8	8010,5

WebService. Les résultats sont les suivants :



Notre grand étonnement est venu du produit Apache Axis qui affiche des performances (par JWS ou par la classe précompilée) **4 à 100 fois moins bonnes que celles de .NET**. A cet égard, le comparatif entre .NET et Glue semble avoir plus de sens, en tous cas pour les méthodes dont le volume des paramètres est faible (jusqu'à 100 éléments dans notre tableau) : Java Glue a des performances globales qui varient de l'ordre de **20% à 40% en déperdition** par rapport aux WebServices .NET. Par contre (comme nous l'avons déjà noté dans d'autres articles), plus le nombre d'éléments de notre tableau est important, plus les écarts se creusent; ils atteignent **un facteur 2,5 pour un tableau de 100 000 éléments** ! Mais nous sommes convaincus que les cas d'utilisation les plus commun des WebServices mettront en oeuvre des invocations de fonctions dont les paramètres et types de retour sont assez concis, du moins dans un premier temps.

Middleware

Méthodologie, hypothèses

La comparaison des trois middlewares mythiques que sont RMI, CORBA et DCOM est un sujet critique qui a donné lieu à de nombreux débats d'experts. La question est remise au goût du jour puisque .NET Remoting n'est pas compatible avec DCOM (il utilise un protocole binaire différent, toujours sur TCP), et RMI peut aujourd'hui fonctionner à la fois sur son protocole initial (JRMP) et sur le protocole de CORBA (IIOP).

Ce test se propose de mettre en oeuvre le même algorithme que précédemment, le calcul du minimum d'un tableau, à la fois sur .NET Remoting (sur un canal TCP et un formateur binaire), sur RMI/IIOP et RMI/JRMP (nous avons utilisé pour la partie Java les implémentations par défaut de IIOP et JRMP incluses dans la machine virtuelle de Sun Microsystems).

L'implémentation d'un serveur de calcul en C# sur .NET Remoting est très simple, encore plus que ne l'était l'implémentation d'un composant DCOM à l'époque (surtout en termes de déploiement) :

```
public interface IGlueMin{
    public double min(double[] tab);
}

/**************************/

import  electric.registry.Registry;
import  electric.server.http.HTTP;

public class GlueMinImpl implements
IGlueMin{
    public double min(double[] tab){
    // Idem précédemment
    }

    public static void main(String[]
argv) throws Exception{ HTTP.startup( "http://localhost:8000/glue" );
Registry.publish( "min", new  GlueMinImpl() );
    }
}
```

GlueMinImpl.java

L'implémentation du même serveur en Java sur RMI est tout aussi triviale :

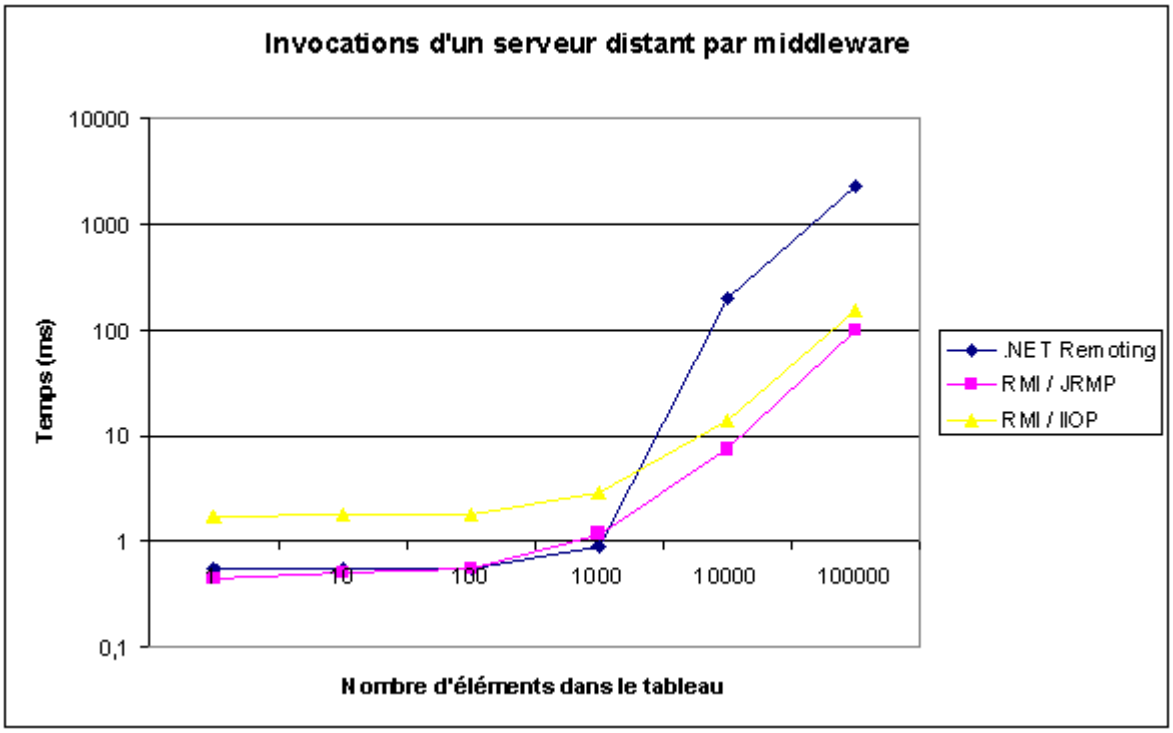
```
public interface IRmiJRMPServeur extends java.rmi.Remote {
    public double min(double[] tab) throws
java.rmi.RemoteException;
}

/**********************************/
import  java.rmi.*;
import  java.rmi.server.*;
import  java.rmi.registry.*;
import  java.net.MalformedURLException;

public class IRmiJRMPServeurImpl extends
java.rmi.server.UnicastRemoteObject implements IRmiJRMPServeur {
    public  IRmiJRMPServeurImpl() throws RemoteException {
        super();
    }

  public double min(double[] valeurs) throws java.rmi.RemoteException {
        double minimum = Double.MAX_VALUE;
        double longueur = valeurs.length;
        double valeurCourante = 0;

        for (int i=0; iif  (valeurCourante < minimum){
                minimum = valeurCourante;
            }
        }
        return minimum;
    }

    public static void
registerToRegistry(String  name, Remote obj, boolean
create) throws  RemoteException, MalformedURLException{
        try { Naming.rebind(name, obj);
        } catch (RemoteException ex){
            if  (create) {
                Registry r =
LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
r.rebind(name, obj);
            } else throw ex;
        }
    }
    public static void main(String[]
args) throws Exception{
    IRmiJRMPServeurImpl obj = new  IRmiJRMPServeurImpl();
registerToRegistry("IRmiJRMPServeurImpl", obj, true);
    }
}IRmiJRMPServeurImpl.java
```

Nous ne reproduisons pas ici l'implémentation du serveur sur RMI / IIOP car il suffit de changer de squelette générique : un remplacement de la classe de base UnicastRemoteObject par PortableRemoteObject suffit donc pour que le serveur change de protocole d'échange.

Résultats

Invocation d'un serveur distant par middleware (ms)	.NET Remoting	RMI / JRMP	RMI / IIOP
Tableau de 1 élément	0,55	0,45	1,7
Tableau de 10 éléments	0,55	0,5	1,8
Tableau de 100 éléments	0,55	0,55	1,8
Tableau de 1000 éléments	0,9	1,2	2,9
Tableau de 10000 éléments	200	7,51	13,82
Tableau de 100000 éléments	2300	101,2	153,42

Ces résultats sont bien plus explicites sous la forme d'un graphique logarithmique :



On peut noter dans notre petit exemple que **.NET Remoting et Java RMI sur JRMP ont des performances équivalentes pour les tableaux de petit volume** (jusqu'à 1000 éléments). Dans cette gamme, **RMI sur IIOP se comporte moins bien (environ 3 fois plus lent)**.

Au-delà de ce volume, les performances de RMI / JRMP se dégradent progressivement et semblent rejoindre celles de RMI / IIOP pour des volumes de données importants en paramètre. Par contre, **.NET Remoting ne semble pas du tout prévu pour supporter ce genre d'utilisation : ses performances se dégradent très vite, pour devenir environ 23 fois moins bonnes que RMI / JRMP.**

Les résultats que nous présentons ici correspondent à une moyenne arithmétique sur un ensemble d'exécutions bien entendu. Pour les autres tests, tous les résultats se situaient dans un voisinage proche de la moyenne; mais pour celui de .NET Remoting avec un tableau de 10 000 ou 100 000 éléments, il s'est passé une chose étrange : de temps en temps, les performances étaient étonnamment bonnes (environ une fois sur 5 ou 10 tests), pour retomber la fois suivante sur un résultat moyen. Nous vous laissons le soin d'imaginer une explication rationnelle à ce comportement surprenant ...

Conclusion

Nous espérons vous avoir convaincus de l'objectivité de notre démarche. Lors de cette étude, nous souhaitions réellement nous rendre compte des performances respectives des plateformes J2EE et .NET sur des exemples certes partiels, mais qui apportent des éléments de réponse significatifs.

Faisons le point :

- le langage Java est moins efficace que C#
- la manipulation de tableaux en Java est environ 2 à 3 fois plus lourde qu'en C#
- les collections Java et .NET sont équivalentes pour de petits volumes, mais les performances de Java se dégradent rapidement avec le nombre d'éléments d'une liste (au-delà de 100 000)
- un client riche écrit en C# / Windows Forms se lance environ 6 fois plus vite que son équivalent Java / Swing
- les WebServices Java et .NET ont des performances du même ordre de grandeur (.NET restant environ 30% plus rapide) lorsque l'on invoque des méthodes dont les paramètres ne sont pas trop volumineux, mais .NET peut aller jusqu'à 2.5 fois plus vite que son adversaire pour des volumes plus importants
- les choses sont inversées entre C# / .NET Remoting et Java / RMI sur JRMP : même ordre de grandeur à petit volume, mais Java RMI supplante .NET Remoting à grand volume (jusqu'à 23 fois plus rapide !)

Bien entendu, notre test est critiquable à plusieurs égards :

- il a été réalisé sur Windows : à l'heure actuelle, il nous est difficile de tester sérieusement .NET sur d'autres plate-formes. Mais nous espérons pouvoir ré-organiser ce match sur Linux dans quelques temps (grâce à l'équipe du projet Mono)
- nous n'avons testé qu'une seule machine virtuelle Java, celle de Sun Microsystem (version 1.4.1). Si la JVM de Sun n'est pas forcément la plus lente, il nous aurait fallu lancer nos tests sur d'autres JVM pour être complet.
- les tests de WebServices et de middlewares sont très dépendants de l'implémentation (en Java du moins). Là encore, il faudrait tester nos petits programmes sur les implémentations des WebServices des différents serveurs d'application du marché, ou encore les implémentations indépendantes telles que WASP (Systinet). Quant au produit Apache Axis, à notre grande surprise, il ne tient pas la comparaison avec .NET .

Après les déclarations diverses que nous avons tous pu lire dans la presse, où .NET battait J2EE à 26 contre 1, puis 10 contre 1, nous y voyons maintenant un peu plus clair. Il est indéniable que **.NET est plus efficace que J2EE pour la majorité des parties que nous avons testé ici, d'un facteur global de 2,5**. C'est déjà énorme : Java a du souci à se faire si les performances pures sont considérées comme un critère majeur dans le choix d'environnement par les décideurs et les particuliers ...

Enfin, nous invitons nos lecteurs à compléter ce test par de petites expérimentations personnelles, et à alimenter le débat des performances de nos deux plate-formes préférées. De notre côté, nous essaierons de continuer cet effort de comparaison objective et systématique sur d'autres sujets à l'avenir, tels que les composants métier, les sites Web, l'accès aux bases de données, et les outils XML. Cela promet de belles surprises en perspective.

Auteur : Thomas GIL

Ressources

Téléchargez ci-dessous le fichier Excel contenant l'ensemble des résultats ainsi que les Matrices, n'hésitez pas à personnalisez vos graphiques pour une meilleure compréhension.

Le Fichier Excel : Results.xls (mis à jour après modification des collections Java)

Téléchargez les codes sources

TestListes.cs TestListes.java (les autres fichiers sources seront bientôt disponibles, le temps de les packager)

GuruPersonnel.cs et GuruPersonnel.java (la partie Swing vs WinForms)