Autor: Marcelo Camargo marcelo.camargo@ngi.com.br

Data: 16/04/2015

Relatório de Análise de Performance com AdvPL

Objetivo

Analisar a performance de componentes e elementos atuais built-in da linguagem `AdvPL` e realizar testes comparativos (micro benchmarkings) para comparação performática. Casos de testes exaustivos são realizados usando componentens equivalentes em sua composição externa, mas com uma implementação interna não tão similiar.

- Caso de Teste 1: HashTables (HashMaps) vs Arrays (Array -> HashMap)

Operações Realizadas

Bibliotecas: `protheus.ch`, `prelude.ch`.

Variáveis vazias: Inicialmente, iremos ter, sem valores, as seguintes variáveis:

- `nI`: Controle de índice do loop externo.
- `nJ`: Controle de índice do loop interno.
- `nTimer`: Controle interno de tempo.
- `nAll`: Controle total de tempo.
- `cSearch`: Acumulador para a chave de busca.
- `nPos`: Posicao do elemento encontrado (`Array`).
- `xValue`: Elemento encontrado (`HashMap`).
- `oHash`: Objeto de *Hash*.
- **Variáveis pré-carregadas**:
- `aData`: Apenas um *Array* não preenchido (`{}`).

Rotina de Execução: Seguiremos a seguinte rotina de execução e testes:

- Percorra todos os elementos da nossa estrutura (+ 1 elemento indefinido);
- Quando estourar o índice, torne a pesquisa um valor inexistente;
- Enquanto não estourar, realize, **cada vez**, 1.000.000 de buscas, utilizando `aScan`, `HMGet`;
- O número total de buscas será definido por `F(B).QN`, onde `Q` refere-se ao número total de elementos na estrutura e `N` refere-se à quantidade de buscas realizadas em cada elemento da estrutura.

Resultado dos Testes

Através dos testes, obtivemos a seguinte saída:

Usando Array:

```
Searching for [ER] took 2.3s
Searching for [HS] took 2.637s..
Searching for [JV] took 2.755s.
Searching for [JS] took 3.021s.
Searching for [PL] took 3.275s.
```

```
Searching for [OC] took 3.715s.
Searching for [CJ] took 3.842s.
Searching for [KO] took 4.153s.
Searching for [BA] took 4.572s.
Searching for [PH] took 4.614s.
Searching for [PY] took 4.876s.
Searching for [HB] took 5.299s.
Searching for [SC] took 5.284s.
Searching for [CP] took 5.632s.
Searching for [GR] took 5.83s.
Searching for [LS] took 6.155s.
Searching for [AG] took 6.226s.
Took 74.189s. with Array.
```

Usando HashMap:

```
Searching for [ER] took 0.935s
Searching for [HS] took 0.95s..
Searching for [JV] took 0.93s.
Searching for [JS] took 0.945s.
Searching for [PL] took 0.944s.
Searching for [OC] took 0.953s.
Searching for [CJ] took 0.918s.
Searching for [KO] took 0.933s.
Searching for [BA] took 0.912s.
Searching for [PH] took 0.941s.
Searching for [PY] took 0.918s.
Searching for [HB] took 0.93s.
Searching for [SC] took 1.012s.
Searching for [CP] took 0.92s.
Searching for [GR] took 0.949s.
Searching for [LS] took 0.255s.
Searching for [AG] took 0.922s.
Took 15.949s. with HashMap.
```

Dados Brutos:

- **Pico** *Array*: 6.226s.
- **Pico** *HashMap*: 1.012s.
- **Base** *Array*: 2.3s.
- **Base** *HashMap*: 0.918s.
- **Média** bruta *Array*: 4.1216s.
- **Média** bruta *HashMap*: 0.8860s.
- **Média** líquida *Array*: 4.3640s.
- **Média** líquida *HashMap*: 0.9381s.
- **Total** *Array*: 74.189s.
- **Total** *HashMap*: 15.949s.

Conclusão

Pudemos observar que, para grandes operações, *HashMaps* são exponencialmente mais performáticos do que *arrays* combinados. Nota-se que ocorre, em buscas em *Arrays*, uma progressão aritmética no tempo, contudo, mantendo-se este constante com `HashTables`, o que torna a diferença ainda mais exponencial com um caso ainda maior de testes:

Array: `P(Array).(D = R([R]+)) = D + ~(R / 3)`, onde `R` representa o elemento atual, `P` representa uma função progressiva, `D` representa o próximo elemento e a progressão é determinada por, aproximadamente, o valor do elemento atual

acrescido de 1/3 do tempo médio para o próximo elemento.

HashMap: `P(HashMap).R`, onde `R` representa o elemento atual e não há acréscimo ou decréscimo de acordo com o número de elementos da estrutura.

Nos nossos testes, o `HashMap` tomou `21,4%` do tempo que o *array* tomou para realizar as mesmas operações. Conseguimos, com 18 elementos, um percentual de `78,6% (~5x)` ganho performático, sendo este exponencialmente maior de acordo com o número de elementos da estrutura.

Foram realizados, em cada componente, `18.000.000` (18 milhões) de testes.

Fontes Utilizados

```
#include "protheus.ch"
#include "prelude.ch"

#define ELEMENT_KEY 1

Function TestHash()
   Let aData <- { }
   Let nI, nJ, nTimer, nAll, cSearch, nPos, xValue, oHash

   On aData aAdd { "ER", "Erlang"     }
   On aData aAdd { "HS", "Haskell"    }
   On aData aAdd { "JV", "Java"       }
   On aData aAdd { "JS", "Javascript" }
   On aData aAdd { "PL", "Perl"       }
   On aData aAdd { "OC", "OCaml"      }
   On aData aAdd { "CJ", "Clojure"    }
   On aData aAdd { "KO", "Kotlin"     }
   On aData aAdd { "BA", "BASIC"      }
   On aData aAdd { "PH", "PHP"        }
   On aData aAdd { "PY", "Python"     }
   On aData aAdd { "HB", "Harbour"    }
   On aData aAdd { "SC", "Scala"      }
   On aData aAdd { "CP", "C++"        }
   On aData aAdd { "GR", "Groovy"     }
   On aData aAdd { "LS", "Livescript" }
   On aData aAdd { "AG", "Agda"       }

   nAll <- Seconds()

   /**
    * Using simple arrays.
    */
   For nI <- 1 To Len( aData ) + 1
      If nI > Len( aData )
         cSearch <- "ML"
      Else
         cSearch <- aData[ nI ][ ELEMENT_KEY ]
         nTimer <- Seconds()
         For nJ <- 1 To 1000000
            nPos <- aScan( aData, { |X| X[ 1 ] Is cSearch } )
         Next
         nTimer <- Seconds() - nTimer
         ConOut("Searching for [" + cSearch + "] took " + ( cValToChar( nTimer ) ) + "s." )
      EndIf
   Next nI
   nAll <- Seconds() - nAll
```

```
ConOut( "Took " + cValToChar( nAll ) + "s. with Array." )
ConOut( "" )

/**
 * Using Hashtables (tHashMap).
 */
nAll <- Seconds()
oHash <- aToHM( aData )
For nI <- 1 To Len( aData ) + 1
   If nI > Len( aData )
      cSearch <- "ML"
   Else
      cSearch <- aData[ nI ][ELEMENT_KEY ]
      nTimer <- Seconds()
      For nJ <- 1 To 1000000
         HMGet( oHash, cSearch, /* out */ @xValue )
      Next nJ
      nTimer <- Seconds() - nTimer
      ConOut("Searching for [" + cSearch + "] took " + ( cValToChar( nTimer ) ) + "s." )
   EndIf
Next nI
nAll <- Seconds() - nAll

ConOut( "Took " + cValToChar( nAll ) + "s. with HashMap." )
ConOut( "" )

Return
```

- Caso de Teste 2: HashTables (HashMaps) vs Arrays (HashMap from Scratch)

Operações Realizadas

Variáveis vazias: Inicialmente, iremos ter, sem valores, as seguintes variáveis:

- `nI`: Controle de índice.
- `nKey`: Chave do elemento.
- `aValues`: Valores do elemento.
- `nAll1`: Controle de tempo para *array*.
- `nAll2`: Controle de tempo para *HashMap*.
- `nElements`: Propagação aritmética de base 2 para a quantidade de elementos.

- `oHash`: Objeto de *Hash*.

- **Variáveis pré-carregadas**:

- `aData`: Carregará todos os dados do teste.
- `aCheck`: Carregará dados **não** repetidos buscados.

Rotina de Execução: Seguiremos a seguinte rotina de execução e testes:

- De 2 até (32768 + 1) elementos, seguindo a base 2, iremos realizar operações.
- Preenchemos os elementos com valores de 1 até o número total de elementos.
- Percorremos os elementos e, buscando em `aCheck`, verificamos se há alguma chave duplicada para com `aData`.
- Havendo chave duplicada, vamos para o próxima tentativa.
- Não havendo, iremos dar `push` para nosso acumulador.
- O crescimento **deve** ser exponencial.

Resultado dos Testes

```
Qtd. - Array - HashMap
    2 |    0.000s. |    0.000s.
    4 |    0.000s. |    0.000s.
    6 |    0.000s. |    0.000s.
    8 |    0.000s. |    0.000s.
   16 |    0.000s. |    0.000s.
   32 |    0.000s. |    0.000s.
   64 |    0.001s. |    0.000s.
  128 |    0.002s. |    0.001s.
  256 |    0.009s. |    0.002s.
  512 |    0.042s. |    0.005s.
 1024 |    0.147s. |    0.010s.
 2048 |    0.616s. |    0.018s.
 4096 |    2.803s. |    0.040s.
 8192 |   15.182s. |    0.080s.
16384 |   67.501s. |    0.164s.
32768 |  286.509s. |    0.310s.
```

Dados Brutos:

- **Pico** *Array*: 286.509s.
- **Pico** *HashMap*: 0.310.
- **Base** *Array*: 0.0s.
- **Base** *HashMap*: 0.0s.
- **Média** *Array*: 26.324375s.
- **Média** *HashMap*: 0.039375s.
- **Total** *Array*: 373.19s.
- **Total** *HashMap*: 0,63s.

Conclusão

Nota-se que, com até 64 elementos, o ganho de `HashMap` não equivale a uma quantia suficientemente agradável para utilização, no entanto, quando trabalhamos com milhares de elementos, igualmente, a performance de `HashMap` é **milhares** de vez melhor do que a performance de um *array*, podendo as tabelas de *hashs* substituirem *TRBs*. O resultado final, contando com 32768 elementos, é de `286.509` para `0.310`, tornado a performance de um `HashMap` **922,5** vezes melhor, sendo que o `HashMap` consumiu `0.10 para 100` do tempo que consumiu um *array*.

Fontes Utilizados

```
Function HashScratch()
   Local aData := { }
   Local aCheck := { }
   Local nI, nKey, aValues, nAll1, nAll2, nElements, oHash

   nElements := 2

   ConOut("Qtd. - Array - HashMap")
   While nElements <= 32769
      aSize( aData, 0 )

      For nI := 1 to nElements
         aAdd( aData, { nI , "Value " + cValToChar( nI ) } )
      Next

      /**
```

```
      * Array
      */
     aSize( aCheck, 0 )
     nAll1 := Seconds()
     For nI := 1 to nElements
        nKey := aData[ nI ][ 1 ]
        If aScan( aCheck , {|x| x[ 1 ] == nKey }) == 0
           aAdd( aCheck, aData[ nI ] )
        Endif
     Next
     nAll1 := Seconds() - nAll1

     /**
      * HashMap
      */
     aSize( aCheck, 0 )
     nAll2 := Seconds()
     oHash := HMNew()
     For nI := 1 to nElements
        nKey := aData[ nI ][ 1 ]
        If !HMGet(oHash, nKey, @aValues)
           HMAdd(oHash, aData[ nI ])
           aAdd(aCheck, aData[ nI ])
        Endif
     Next
      nAll2 := Seconds() - nAll2

     ConOut( Str( nElements,5 ) + " | " +     ;
             Str( nAll1, 8, 3 )   + "s. | " + ;
             Str( nAll2, 8, 3 )   + "s." )

     nElements *= 2
  EndDo
  Return
```

- Caso de Teste 3: Paralelismo Puro (Monothread vs Multithread)

Operações Realizadas

Bibliotecas: `protheus.ch`, `prelude.ch`.

Variáveis vazias: Inicialmente, iremos ter, sem valores, as seguintes variáveis:

- `nCount`: Controle do tempo.

Rotina de Execução: Seguiremos a seguinte rotina de execução e testes:

- Execute o `Processo1`. Ele deverá mapear um array de 1 até 65535 e elevar cada elemento ao quadrado.
- Executo o `Processo2`. Ele deverá mapear um array de 1 até 65535 e multiplicar cada número por 1000.

Resultado dos Testes

```
Optimal time:             0.672s.
Simple time:             0.51s.
```

Conclusão

Utilizando funções puras com paralelismo e múltiplas *threads*, **não** há ganho de performance, justamente pelo contrário, torna-se aproximadamente `20%` mais lento do que se executado de forma comum.

Fontes Utilizados

```
#include "protheus.ch"
#include "prelude.ch"

Function Parallel()
   Local nCount

   /**
    * Tests with parallel computation.
    */
   nCount <- Seconds()
   @Parallel {  ;
     "Process1" ;
   , "Process2" ;
   }
   ConOut( "Optimal time: " + Str( Seconds() - nCount ) + "s." )

   nCount <- Seconds()
   Process1()
   Process2()
   ConOut( "Simple time: " + Str( Seconds() - nCount ) + "s." )

   /**
    * Tests without parallel computation, one processor, one core.
    */
   Return

   Function Process1( aData )
      Return @Map { ( Fun (X) -> X * X ), @{ 1 .. 65535 } }

   Function Process2( aData )
      Return @Map { ( Fun (X) -> X * 1000 ), @{ 1 .. 65535 } }
```

- Caso de Teste 3: Paralelismo c/ Side-Effect (Monothread vs Multithread)

Operações Realizadas

Bibliotecas: `protheus.ch`, `prelude.ch`.

Variáveis vazias: Inicialmente, iremos ter, sem valores, as seguintes variáveis:

- `nParallel`: Controle do tempo paralelo.
- `nSeq`: Controle do tempo sequencial.

Rotina de Execução: Seguiremos a seguinte rotina de execução e testes:

- Em 3 processos, em um total de 150.000 vezes, daremos a saída à tela de um numeral.

Resultado dos Testes

```
Parallel:               34.075s.
Sequential:             30.563s.
```

Conclusão

O modelo atual de paralelismo e *multithreading* implementado em AdvPL não é conciso o suficiente para nossa implementação, nem satisfaz às necessidades performáticas.

Fontes Utilizados

```
#include "protheus.ch"
#include "prelude.ch"

Function Parallel()
   Local nParallel, nSeq

   /**
    * Tests with parallel computation.
    */
   nParallel <- Seconds()
   @Parallel { "Out1", "Out2", "Out3" }
   nParallel <- Seconds() - nParallel

   nSeq <- Seconds()
   Out1()
   Out2()
   Out3()
   nSeq <- Seconds() - nSeq

   ConOut( Replicate( "-", 35 ) )
   ConOut( "Parallel:   " + Str( nParallel ) + "s." )
   ConOut( "Sequential: " + Str( nSeq ) + "s." )
   ConOut( Replicate( "-", 35 ) )
   Return

Function Out1
   Local nI
   For nI <- 1 To 50000
      ConOut( "[1] Reached: " + Str( nI, 12) )
   Next nI
   Return

Function Out2
   Local nI
   For nI <- 1 To 50000
      ConOut( "[2]Reached: " + Str( nI, 12) )
   Next nI
   Return

Function Out3
   Local nI
   For nI <- 1 To 50000
      ConOut( "[3] Reached: " + Str( nI, 12) )
   Next nI
   Return
```

- Caso de Teste 4: dbSeek vs msSeek

As funções `dbSeek` e `msSeek` tem, basicamente, a mesma funcionalidade. A vantagem de `msSeek` é que ela não precisa reacessar a base de dados para encontrar uma informação já em uso pela **thread**.

Assim, a **thread** mantém na memória os dados necessários para carregar registros já encontrados através de `dbSeek` (no caso, `RecNo`). assim, a aplicação pode simplesmente executar sem recarregar, aplicando, implicitamente, memoization.

Operações Realizadas

Bibliotecas: `totvs.ch`.

Variáveis vazias: Inicialmente, iremos ter, sem valores, as seguintes variáveis:

- `nSec`: Controle do tempo paralelo.
- `nI`: Controle de índice.

Rotina de Execução: Seguiremos a seguinte rotina de execução e testes:

- 100.000 vezes, acesse a `SX1`.
- Execute `SetOrder`
- Execute `msSeek` ou `dbSeek`

Resultado dos Testes

```
Time dbSeek:              11.346s.
Time msSeek:              17.642.
```

Conclusão

Apesar da TOTVS indicar o `msSeek`, o `dbSeek` mostrou-se mais rápido, em oposição ao teste desta, isso porque o sistema de memoization de AdvPL não foi projetado para suportar grandes volumens acumulados em cache, provavelmente, em sua implementação, foram utilizados acumuladores simples, o que ocasionou uma queda performática.

Fontes Utilizados

```
#include "totvs.ch"

Function TestSeek()
   TestDBSeek()
   TestMSSeek()
   Return

Static Function TestDBSeek
   Local nSec := 0
   Local nI

   nSec := Seconds()
   For nI := 1 To 100000
      dbSelectArea( "SX1" )
      dbSetOrder( 1 )
      dbSeek( xFilial( "SX1" ) + "000001" + "01" )
   Next nI
   nSec := Seconds() - nSec
   ConOut("Time dbSeek: " + Str( nSec ) )
   Return

Static Function TestMSSeek
   Local nSec := 0
   Local nI
```

```
nSec := Seconds()
For nI := 1 To 100000
   dbSelectArea( "SX1" )
   dbSetOrder( 1 )
   msSeek( xFilial( "SX1" ) + "000001" + "01" )
Next nI
nSec := Seconds() - nSec
ConOut("Time msSeek: " + Str( nSec ) )
Return
```