A Comparison Study on Execution Performance of MATLAB and APL

by Han-feng Chen, Wai-Mee Ching* and Da Zheng**

Dept. of Computer Science, Zhejiang Normal University, Jinhua, Zhejiang, China, 321004*

Dept. of Computer Science, Johns Hopkins University, Baltimore, Maryland 21210, USA**

wufeke2010@yahoo.cn *waimee_ching@yahoo.com **zhengda1936@gmail.com

Abstract We present a study on the execution performance of APL and MATLAB on a suite of five programs ranging from one of highly iterative nature to ones mainly do array operations. The comparison on performance is carried out in three different modes of execution: interpreted, compiled and parallel. We found that MATLAB interpreter is in general much faster than APL; and compiled MATLAB code executes not necessary faster while in APL the compiler provides significant performance improvement. The strategies for parallel execution in MATLAB and APL are different: one is interpreter-based, the other compiler-based. We discuss some insights into programming language implementation from this study.

1. Introduction

Execution efficiency is one of the most important issues in the study of programming language implementations, and there are many studies in that area, especially that concerning compiler optimization. Most studies of programming language execution performance confine to one language but with different implementations such as the performance differences due to different compilation techniques or parallelization strategies. It is rare to find a cross-language study which compares performance of different languages, other than general acknowledgement that programs written in a lower-level language such as C execute much faster than those written in a higher-level, or interpreter-based languages. Therefore, it is worthwhile to do a comparison study on execution performances of two existing languages to gain insights into various programming language implementation issues. To this end, we selected two language systems that have long been used in a wide range of real-world applications: APL and MATLAB.

Both APL and MATLAB are very high-level array-based languages; they are similar in providing arrays as the main programming data structure but have different implementation heritages. Both execute much slower with scalar data in interpretative mode compared with that of C, while there are compilers for each which can possibly improve execution performance. Moreover, there are recent works in both language systems to improve the execution performance further by utilizing the parallel computation power afforded by modern multi-core machines. Hence, it is interesting and feasible to compare the execution performance of equivalent programs written in these two languages under various implementations, i.e. interpreted, compiled and parallelized.

For this purpose, we picked a suite of five programs: *floyd, poisson, jacobi, rprime* and *morga*n; these programs are chosen for their different characteristics. *floyd* is a typical scalar iterative program while *poisson* and *jacobi* are typical array-oriented programs. *rprime* involves recursion and *morgan* has function calls. None of the five programs is a large application, but neither are they trivially simple. All comparisons are measured on a Window-based platform. With one exception, MATLAB

interpreter is at least twice faster than APL. On the other hand, the improvement of MATLAB code in execution time after compilation is often modest if exist at all; in case of APL, the compiler gives significant performance improvement: APL outperforms MATLAB after compilation with one exception. For parallel execution, MATLAB provides a toolbox of parallelization facilities for user to use; in APL parallelization is done to compiled code by a new component of the compiler without user intervention.

While our study is limited to a small sample set, it does provide concrete data on the execution performance of this program-set for us to compare and study. This study gives us valuable insights into several programming language implementation issues, and these in turn suggest pragmatic methods to improve performance of existing programming languages in the age of parallelism. In particular, many techniques we developed in implementing a parallelizing APL compiler can be adopted by other very high-level languages to benefit from multi-core machine currently available; on the other hand, MATLAB suggests the importance of linking to highly efficient linear algebra routines which we believe are already parallelized on multi-core machines, thus provide parallelism transparent to the user.

In the next section, after a brief history of each language, we describe the relevant information about the particular systems of APL and MATLAB which we used for our experiment. In Sec.3, we present five example programs, their characteristics and the performance data for different mode of execution. In Sec.4, we state what we learned from the performance comparison and discuss its implications. The paper ends with a conclusion.

2. APL and MATLAB, languages and systems

Both APL and MATLAB are array-oriented programming systems invented by university professors of mathematics. APL was invented by Ken Iverson at Harvard for teaching applied mathematics around 1960, and later he joined IBM T. J. Watson Research Center, where Larry Breed and colleagues implemented the first APL interpreter on S/360 using assembly language in 1966. MATLAB was invented by Cleve Moler of MathWorks; he wrote the first implementation in FORTRAN around 1980 while he was a professor of mathematics and computer science at the University of New Mexico. Iverson intended APL to be an executable mathematical notation (see his Turing Award Lecture [3]) while Moler's original goal was far more practical, i.e. to provide engineers and scientists a way to access the efficient LINPACK and EISPACK without writing FORTRAN programs. Consequently, APL with uniform syntax and succinct symbols has a programming style distinct from any other languages, while the syntax and notations of a MATLAB program are more conventional.

The application areas of APL and MATLAB overlap but are far from the same. APL is used more in actuaries and finance while MATLAB is used more in engineering and industrial simulation. As mentioned earlier APL developed more than 15 years before MATLAB but its traditional focus on mainframe platform made it less popular when many of its existing business applications can be replaced by PC-based mass packages such as Excel. On the other, the continuous development and improvement of MATLAB on workstations and PCs made it more popular and powerful. Still, there are APL vendors other than IBM today while MATLAB is exclusively marketed by MathWorks! We

note here that Morgan Stanley has its own dialect of APL, called A+, which is an open source software (www.aplusdev.org); a line in APL executes from right to left with all primitives of *equal* precedence.

As both APL and MATLAB are mainly implemented by interpreters, an APL or MATLAB program executes slower than a corresponding C program in general. Of course, an APL or MATLAB user is well compensated in productivity and ease of use provided by the respective systems. The systems we use for our base-line comparison are the following: the APL interpreter is a free demo version of APL2 on Windows XP downloaded from IBM (http://www-01.ibm.com/software/awdtools/apl); this does not imply that IBM APL2 represents the current state of performance of APL interpreters on the market such as those of Dyalog APL and APL2000. It simply means that APL2 is easily available to us. The MATLAB version is a 7.8.0.347 (R2009a) release version on Windows XP from MathWorks(http://www.mathworks.com).

MATLAB provides a compiler [6] to improve the efficiency of MATLAB code as well as to provide a way to execute MATLAB code independent of its system environment. To compile and run, we need to develop an environment. We run "***mbuild -setup***" in MATLAB command window to setup compiler, and choose the second option which is named "Microsoft Visual C++ 6.0"; then follow instructions to the end. Upon completion, we can switch to resulting M-file(s) for independent external execution by issuing the command "*mcc –m Mainfile.m file1.m fiel2.m ..*" to generate a standalone application. We note that each function in a MATLAB code will produce a separate m-file after compilation.

IBM does not offer a compiler product for its APL system no are we aware of compilers offered by other APL vendors, but there exists a prototype compiler developed earlier at IBM T.J. Watson Research Center [1], and the APL compiler we use has this origin. It consists of an APL workspace, called *COMPC*, of about 14k lines of APL and a small C library, apl3lib.cpp, to be linked with compiler generated C code to produce executables. A compilation unit *W* is a group of APL functions in a workspace consisting of a main function *F* and all other functions called directly or indirectly by this main function. To compile *W*, we prepare 2 variables: *LPARM* which a character string consists of the name of *F* followed by a blank and types of *F*'s parameters if any, and *RPARM* which is a numeric vector- started with the intended index origin of *W* followed by the shapes (rank, and dimensions if known) of *F*'s parameters. To compile, simply call the main function *COMPILE* in *COMPC*

```
)LOAD COMPC
)COPY W
LPARM COMPILE RPARM
```

MATLAB provides a toolbox to help users to parallel their code in the interpreter environment [7]. Hence，a user need to modify his code appropriately to utilize the facilities in this toolbox such as *parfor*, *distributed arrays* and *spmd*. For reasons to be explained in the next section we are able to parallelize only one example due to either limitations of facilities offered or it requires extensive recoding. Moreover, MATLAB's parallel tool box is really designed for structuring irregular code for parallelism; for array operations, which dominate in our examples, it provides no advantage in speed as we list the performance data of different ways to do a matrix assignment in MATLAB in the appendix. In contrast, our recent work on automatic parallelism [2] provides an extensive and transparent parallelizing tool which is quite simple to use. It consists of a new workspace *PARAC* which needs to

be copied in right after loading *COMPC* to produce C code embedded with OpenMP directives for parallel execution. To execute the parallelized C code, one just compile the resulting code with any C compiler which supports OpenMP and link to a new version of the small C library apl3lib.cpp (we refer to [2] for a brief description of the base compiler *COMPC* as well as the new parallelizing component *PARAC*). The speedup thus gained greatly depends on the nature of the programs to be parallelized, but it requires no additional effort from the user other than compilation. We do not claim that the work on automatic parallelization used here is necessary the most advanced result of parallelization in research on APL-like functional languages.

3. Examples programs and execution time measurement

One of the difficulties in making a cross-language study is to wisely select a small sample of programs which is both representative but has distinct characteristics. The first program is the well known Floyd algorithm of minimum weighted distance between cities. This program is inherently iterative. All other 4 examples are from the testing suite of [1], hence originally written in APL. The 4th example is to find primes up to *N* using a recursive sieve method. The other three are all array-oriented, i.e. they involve large array operations. The last one involves function-calls. We listed *floyd* in MATLAB, and others in APL here (MATLAB counterparts are available upon request).

All measurements are run-time on a 4-core computer, an AMD Athlon II X4 620, clocked at 2.6GHz and with 4 x 512KB L2 cache under Window XP. For parallel performance, we run all 4 threads. We note that parallel speedup is counted against execution time of compiled code in case of APL, and against interpretation time in case of MATLAB. We run each program in both languages at least 5 times to get average timing for all cases. In APL, ⍝ starts a comment.

1. ***floyd*** The first example is a graph analysis algorithm for finding shortest paths in a weighted graph. The output is a matrix of the shortest distance (summed weights) between all pairs of vertices, without the detail of paths. The program is a triply nested loop. We can see that MATLAB interpreter is much faster than APL; but compiled APL is markedly faster than compiled MATLAB. The input data type used is double-precision floating point.

```
function floyd(D)
[n,n] = size(D);
for k=1:n
   for i=1:n
      for j=1:n
         D(i,j) = min(D(i,k)+D(k,j),D(i,j));
      end
   end
end
```

floyd(200x200)	interpreter	compiled	speedup	parallelized	speedup
APL	42352 ms	78 ms	542.97	65.2 ms	1.1963
MATLAB	423 ms	416 ms	1.0168	*	*
ratio	100.123	0.1875			

We did make an effort to parallelized the MATLAB code by using *parfar* facility in [7]; but as the for loops are nested and data dependent, it ended up in a much worse performance than the original code

which we did not list here (around 33 sec.), and we cannot ascertain its correctness formally.

2. ***poisson*** The second example is to solve a Poisson equation with boundary condition. The input data type used is double-precision floating point.

```
[0]  Z←POISSON RMINBU;P;Q;L;M;S;T;V
[1]  Z←0 0⍴0                          ⍝ set up an empty matrix
[2]  →(2≠⍴⍴RMINBU)/0                  ⍝ check to see whether it is 2-dim
[3]  P←1+¯1↑⍴RMINBU                   ⍝ P is dim2+1
[4]  Q←1+1↑⍴RMINBU                    ⍝ Q is dim1+1
[5]  L←¯4×(1○○(⍳Q-1)÷2×Q)*2           ⍝ W*2 is square of W. ⍳5 is 0 1 2 3 4
[6]  M←¯4×(1○○(⍳P-1)÷2×P)*2           ⍝ ○A is pi×A. 1○A is sin A
[7]  S←1○○(⍳Q-1)∘.×(⍳Q-1)÷Q
[8]  S←S÷(+/S[1;]*2)*0.5              ⍝ +/V is sum of V
[9]  T←1○○(⍳P-1)∘.×(⍳P-1)÷P
[10] T←T÷(+/T[1;]*2)*0.5
[11] V←L∘.+M                             ⍝ V is the addition table of L and M
[12] Z←S+.×((S+.×RMINBU+.×T)÷V)+.×T ⍝ W+.×T is the matrix multi. of W & T
```

poisson(501x501)	interpreter	compiled	speedup	parallelled	speedup
APL	1110 ms	718 ms	1.546	354 ms	2.028
MATLAB	146 ms	213 ms	0.685	154 ms *	0.948
ratio	7.603	3.37		2.299	

In this example, we see some that MATLAB interpreter is not only more than 7 times faster than APL, it is faster than its' compiled code and that of compiled APL. At first glance, this is counter-intuitive, but a closer look explains why. We note that ∘.× is outer-product, +.× is inner-product, i.e. matrix multiplication, and the whole program running time is dominated by these huge matrix operations. If we recall the root of MATLAB, i.e. to provide a user to have access of the most efficient FORTRAN linear algebra packages, we realize that the efficiency thus gained overshadows minor cost of interpretation. And in this case, although compiled APL improves upon interpreter, it is no match for the efficiency of *BLAS* routines, even with a respectable parallel speedup of more than 2.

We parallelized the MATLAB code by using ***distributed arrays*** facility from [7] by simply adding the following lines in front of the existing code (where oRMINBU stands for input data):

```
[q,p] = size(oRMINBU);
RMINBU = codistributed(oRMINBU,'convert');
```

Notice that the execution performance is actually a bit worse than the original interpreter number. We believe this is because matrix multiplication in MATLAB already takes advantage of multi-cores hardware whether one uses its parallel toolbox or not. In fact, our use of *distributed arrays* may well have incurred the cost of extra data movement.

3. ***jacobi*** The third example is to iteratively averaging temperature from a rectangular boundary to interior to reach within a pre-set error bound**.** The input data type is double-precision floating point.

```
[0]  Z←F JACOBI A;C;E  ⍝ A with values on boundary and 0s in interior
[1]  E←0.1             ⍝ F is 0s on boundary and 1s in interior
[2]  C←(Z←A)×~F        ⍝ ~0 1 1 0 0 1 is 1 0 0 1 1 0
```

```
[3] L:R1←¯1⊖A←Z          ⍝ ¯1⊖A rotates A upwards by one row
[4]  R2←1⊖A              ⍝ 1⊖A rotates A downwards by one row
[5]  R3←1⌽A              ⍝ 1⌽A rotates A rightwards by one column
[6]  R4←¯1⌽A             ⍝ ¯1⌽A rotates A leftwards by one column
[7]  →(E<⌈/|,A-Z←C+0.25×F×R4+R3+R2+R1)/L ⍝ if max of abs(A-Z)> E jump to L([3])
```

jacobi(101x101)	interpreter	compiled	speedup	parallelized	speedup
APL	448 ms	93 ms	4.817	78 ms	1.1923
MATLAB	126 ms	128 ms	0.984	*	*
ratio	3.556	0.726			

First we note that in case APL, we can achieve better parallel speedup here as well as in *poisson* by using a bigger matrix [2]. For MATLAB, *jacobi* is a good example for using *spmd* from the toolbox [7] as it requires data exchange while splitting data to multiple cores; but we do not have time to work out the necessary communication details between adjacent grids here. We see in this example, unlike in *poisson*, compiled and parallelized APL outperform MATLAB; and this is likely due to the fact the matrix operations involved here have fewer opportunities for optimization and fine-tuning.

4. ***rprime*** The fourth example is to find prime numbers less than or equal to *N*, using a recursive sieve method starting with the set of primes less than 100. It uses a Boolean vector to indicate prime-ness of an integer at its corresponding bit-position. The data types used are integer and Boolean.

```
[0]  P←RPRIME N;I
[1]  P←2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
[2]  →(~N≤100)/L0
[3]  P←(N≥P)/P
[4]  →0
[5] L0:PL←⍴P←RPRIME⌊N*0.5
[6]  B←N⍴0
[7]  I←1
[8] L2:B←B∨N⍴(-P[I])↑1
[9]  →(PL≥I←I+1)/L2
[10] P←P,1↓(~B)/⍳N
```

Finding prime number < 88000

rprime(88000)	interpreter	compiled	speedup	parallelled	speedup
APL	16 ms	10 ms	1.6	10ms	1
MATLAB	56 ms	61 ms	0.918	*	*
ratio	0.2857	0.1639			

In this example, we see that APL handles bits-vectors and recursion quite efficiently compared with MATLAB; and compiled APL also has efficient bits-vector implementation. As in the last example, compiled MATLAB incurred some additional setup cost without giving any speedup. The automatic parallelism capability implemented in *PARAC* does not apply to operations in *rprime*. Hence, it has no parallel speedup at all. For MATLAB, the limitation that indices in a *parfor* loop must be contiguous precludes its use for parallelization of *rprime*.

5. ***morgan*** The fifth example is some calculation from a financial application; it is the only non-recursive example here with a function call; and we note that MATLAB compiler generates two m-files for the two functions. The input data type used is double-precision floating point.

```
[0]  R←N MORGAN A;X;Y;SX;SX2;SY;SY2;SXY ⍝ N is a number, A a 3-dim array
[1]  X←A[1;;]                           ⍝ X the 1st hyperplane of A
[2]  Y←A[2;;]                           ⍝ Y the 2nd hyperplane of A
[3]  SX←N MSUM X
[4]  SY←N MSUM Y
[5]  SX2←N MSUM X*2
[6]  SY2←N MSUM Y*2
[7]  SXY←N MSUM X×Y
[8]  R←((SXY÷N)-(SX×SY)÷N*2)÷(|((SX2÷N)+(SX÷N)*2)*0.5)×(|(SY2÷N)-(SY÷N)*2)*0.5

[0]  R←N MSUM A                    ⍝ A is a matrix. (0,-N)↓T drop last columns of T
[1]  R←((0,N-1)↓T)-0,(0,-N)↓T←+\A  ⍝ T is the partial sum of A on each row
```

morgan(2×1510×2010)	interpreter	compiled	speedup	parallelled	speedup
APL	2578 ms	516 ms	4.996	182 ms	2.835
MATLAB	1345 ms	1343 ms	1.0014	*	*
ratio	1.9167	0.3842			

In this last example, run-time is also dominated by large operations; we see that MATLAB is twice as fast as APL, but after compilation, APL is 3 times faster. The difference with *poisson* is that the matrix operations here do not have ready counterparts in *BLAS*. We also see that in this example, parallelized APL code achieved very good result due to *virtual operations* and *streaming* [2]. For MATLAB, we cannot use the distributed array facility from Parallel Toolbox here due to the limitation that it cannot contain function calls.

4. Implications of the comparative performance data

From the timing data in the previous section, we have three general observations: First MATLAB interpreter is much faster than that of APL if computation is dominated by nested for-loops (*floyd*) or can directly call highly optimized linear algebra routines either provided by Intel's MKL (Math Kernel Library) or packages based on FORTRAN (*poisson*) which most likely are already parallelized to take advantage of multi-core hardware. It is also twice faster than APL on *morgan* where there are also large array computations. APL interpreter is only faster than MATLAB when there is recursion or bits-operations (*rprime*).

Second, compiled MATLAB code executes only modest faster than in its interpreter, if at all, while in APL that performance improvement can be significant, especially on inherently sequential code like *floyd*. Indeed, *COMPC* provides speedup to all programs in our samples, but the amount of speedup gained by compilation greatly depends on the nature of the program. With the exception of *poisson* where matrix multiplications dominated computation time, compiled APL substantially outperforms MATLAB counterpart. As we mentioned earlier, *COMPC*'s strategy of compiling APL is to translate APL into an equivalent C program after extensive type-shape analysis of a program's variables. There is also substantial work on compiling MATLAB such as de Rose [5]. We do not have technical information on the implementation of the MATLAB product compiler. But it seems that the product adopted a rather conservative strategy and generally geared more towards providing a tool for producing independently executable module than for faster execution.

Third, MATLAB thru Parallel Toolbox provides very comprehensive solutions for parallelism: it has *distributed arrays* to parallelize vector operations, *parfor*, which is equivalent to the OpenMP *parallel*

for directive, and *spmd*, which is a more general and more powerful tool than *parfor* but requires more effort to write a parallel program. Nevertheless, there are quite number of limitations in utilizing these set of facilities as mentioned separately in the previous section. More importantly, the whole aim of the Toolbox is to speedup interpretation in parallel, not parallelizing compiled code by a compiler. We can see then a two prongs strategy of MATLAB to give a user the benefit of parallelism: one explicit, i.e. to let a user do some work using the toolbox, the other implicit, i.e. linking highly optimized and parallelized underlying routines in the interpreter. In case of APL we championed the approach of parallelizing compiled code by a parallelizing component of the compiler, i.e. parallelism is *automatically* provided by the compiler, at least for data parallelism. Not only this lessens the burden of studying parallelism on user's part but it is also shown to be effective in examples 2, 3 and 5 where the programs are written in an array-oriented style. The possibility of *automatic* parallelism is one advantage a very high level language such as MATLAB or APL has over C. We note that there is research work on parallel MATLAB compiler such as that in [4].

Finally, from the performance data in example 2, the only example where compiled/parallelized APL executes slower than MATLAB, we learned the importance of utilizing highly optimized routines of frequently used linear algebra operations which may even be hand-crafted by hardware vendors' developers with particular knowledge of the underlying machine architecture. Highly efficient routines for some common functions, when their computation dominates run-time, can easily compensate inefficiency incurred by an interpreter. However, other than linking to ultra-efficient routines, fine-tuning the performance of an interpreter, or parallelizing its execution, is not necessary a wise approach. For instance, had we called the efficient package BLAS or MKL in our APL compiler, our compiled code would have performed at least on par with that of MATLAB. We believe to optimize and/or parallelize compiled code is a more effective approach to enhance execution performance of very high level programs as it will cover more general type of programs, and this is supported by the performance data from our sample programs.

5. Conclusion

We presented a study on the execution performance of MATLAB and APL. We measured and compared the run-time of 5 example programs of various characteristics executed under interpreter, after compilation or parallelization. We gained valuable insights into some important issues in program language implementation. We believe that cross-fertilization in language implementation research is quite desirable, especially for languages which are not too dissimilar. Works done in APL compiler and parallelization can be profitably adopted in MATLAB; and conversely APL and its compiler implementation can learn from the MATLAB interpreter's efficient call to existing highly optimized math routines.

References

[1] W.-M. Ching and D. Ju, An APL-to-C Compiler for the IBM RS/6000: Compilation, Performance and Limitations, APL Quote Quad, Vol.23, No.3, 15-21, 1993.

[2] W.-M. Ching and Da Zheng, Automatic Parallelization of Array-oriented Programs for a Multi-core Machine, submitted for publication.

[3] Ken. Iverson, Notation as a Tool of Thought, Comm. ACM, vol.23, no.8, 444-465, 1980.

[4] M. Quinn, A.Malishevsky, N. Seelam and Y. Zhao, Preliminary Results from a Parallel MATLAB Compiler, Proc. Int'l Parallel Processing Symp., 81-87, 1998.

[5] Luiz A. De Rose, Compiler Techniques for MATLAB Programs, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1996.

[6] The MathWorks, MATLAB Compiler, 6th ed., 2002.

[7] The MathWorks, Parallel Computing Toolbox 4, User's Guide 2010.

Appendix (timing of different mode of coding matrix assignment in MATLB in milliseconds)

data size		100x100 time/ratio		500x500 time/ratio		1000x1000time/ratio	
1.	C = A	8.263	1	9.626	1	10.160	1
2.	C(i,:) = A(i,:)	10.782	1.30	22.858	2.37	54.378	5.35
3.	C(i,j) = A(i,j)	13.176	1.59	38.879	4.04	67.510	6.64
4.	parfor i; C(i,:) = A(i,:)	177.791	21.5	185.138	19.2	395.315	38.9
5.	parfor ; cell use	196.321	23.8	762.237	79.2	2907.148	286

Initialization:

```
n = 100; %n = 100/500/1000
A = rand(n,n);                  //A is of type double-precision floating point
C = zeros(n,n);
```

Note: *Initialization* not included in timing; parfor uses two labs with command "matlabpool open"

2.

```
for i=1:n
    C(i,:) = A(i,:);
end
```

3.

```
for i=1:n
    for j=1:n
        C(i,j) = A(i,j);
    end
end
```

4.

```
parfor i=1:n
    C(i,:) = A(i,:);
end
```

5.

```
n = 100;
A = rand(n,n);
C = cell(n,1);
parfor i = 1:n
    for j = 1:n
        C{i}(j) = A(i,j);
    end
end
C = cell2mat(C); %this step is necessary
```