Paper TU01

Considerations for improving program performance

Martin Gregory, Merck Serono, Darmstadt, Germany

1 Abstract

This paper discusses performance in the sense of efficient use of computing resources by a program. For short running or one-off programs performance is generally not an issue which needs to be considered. For long running or frequently used programs, especially if these frequently used programs may be called from within a loop, paying attention to efficient use of computing resources can cut down the amount of time spent waiting for results or avoid monopolising resources. Increasing volumes of data and complexity mean that statistical programmers need to be aware of the factors affecting performance of a program. In addition to showing some techniques and approaches which can contribute to efficiency, we also discuss the more general question of when investing in performance improvement pays off. Techniques are illustrated primarily using examples from SAS.

2 Why consider program performance?

In comparison with software developers in other industries statistical programmers have, up to the recent past, generally dealt with relatively small amounts of data with the result that performance issues were not particularly relevant. The volume of data involved in analysis of trials, and especially submissions has increased dramatically. The adoption of various CDISC standards have also led to an increase in the amount of space required to store the same amount of information. The increased standardisation associated with the adoption of CDISC also means that programs are more likely to be re-used and a program which uses an acceptable level of resources in one context may cause problems in another.

Let us consider some real life examples which illustrate such potential problems. The first case concerns a macro designed to compare outputs for validation purposes. This works by reading the text files into SAS datasets and using the COMPARE procedure to find differences. Each line of the text file is read into a character variable. The programmer[1] used a length of 32,767 in order to avoid the danger of ignoring a difference because of truncation. In general, the program works well. When used to compare two 50 MB files, each with 350,000 lines, the temporary datasets used by the macros are approximately 1.1GB in size, of itself not a large problem. However, after 53 minutes execution time, the program failed with a disk full condition on the WORK library. COMPARE itself generates further temporary files for the comparison which resulted in the disk full condition. The solution was quite simple - modify the macro to check the maximum record length and use this for the text variable length. This modification reduced the execution time to a few seconds, despite having to read the input files twice.

The second case concerns a macro for generating Kaplan-Meier graphs for various subgroups. The first version looped over the values of a single variable and performed in a reasonable time. Then the looping was extended to five nested variables and suddenly the program was taking 2.5 hours to run. Analysis of the individual steps showed that no single step was taking more than 0.02 seconds. The real problem was that several temporary datasets were being created each time the inner loop executed, even though the result was the same for given values of the outer loop. Moving the creation of these datasets into the appropriate loop reduced execution time to 5 minutes.

In both of these cases, particular program design decisions, or failure to make such decisions resulted in time being spent finding problems which could have been avoided. A little more time spent on design can help to avoid such problems, when one knows what needs to be taken into account. The second case of unnecessarily repeating actions in nested loops is a common one to all programming languages and can be easily avoided. The first case, however, shows that the intention to avoid one problem may inadvertently cause other problems.

So far we have discussed performance in relation to individual programs. Writing programs which make efficient use of resources also helps to avoid causing problems for other users. Resources on a computer system are finite and generally shared with others. Efficient programs not only deliver results faster for the person using them, they also minimize interference with other users programs. They do not fill up disks and do not monopolize the CPU.

[1] and author of this paper

3 What factors influence performance?

In this section we discuss the factors which influence performance: cpu time, I/O, memory and development time. These factors are not indepenent and like any optimization problem, the art lies in recognising which are the important factors for the problem at hand.

Computers provide essentially three different resources: calculation (CPU), storage (volatile or non-volatile) and input/output. Improving performance is a question of optimizing the use of these resources. These resources are not independent. Reducing the use of one may increase the use of one or both of the others. In particular it is possible to trade volatile storage against either computation or I/O. Since all three resources are finite there is a limit to the amount of optimization which can be done.

To these three resources we must add a fourth: the time spent by the developer improving the use of resources. Developer time is also a finite resource. As a result, in seeking to optimize the performance of a program we need to consider this fourth resources as well.

What are the factors which influence the use of these resources? The time a program takes to run depends primarily on the level of complexity of the algorithms used and the volume of data read or written. At a second level, precisely how the algorithms are implement may have a significant influence and one which is more under the control of the programmer than the availability of CPU time or I/O bandwidth.

4 Generally applicable rules

In this section we discuss general rules which, when followed, contribute to improving performance. In section 5 below we discuss how such general rules may or may not apply to a particular programming language.

4.1 CPU time

The most important rule may be stated quite simply as

> avoid unnecessary computations.

All other CPU optimizations are a corollory of this rule. The first corollary, and one where mistakes are easy to make and therefore where performance improvements are easy to obtain involves invariants and loops.

> if a variable does not change inside a loop, do not assign its value in the loop.

While some languages recognize loop invariants and optimize them during compilation, as DeVenezia [3] points out, SAS does not. For example in SAS the following code:

```
x=mean(1,2,3) ;
do i=1 to &iter ;
   end ;
```

runs between 3 and 10 times faster than

```
do i=1 to &iter ;
   x=mean(1,2,3) ;
   end ;
```

for values of *&iter* of 200 and 20,000, respectively. While very few programmers would write the loop in the second way, this was an example built to test the difference and in a realistic loop it may not be so obvious that a particular assignment is an invariant. The practice of keeping invariants outside the loop becomes all the more important for nested loops.

Another, less obvious, corollary is

> use less expensive operations when this makes sense

A well-known case of this is that addition is less expensive than multiplication as the following snippet of R code shows:

```
> start <- Sys.time();2*2;end <- Sys.time();print (end-start)
[1] 4
Time difference of 0.0001099110 secs
> start <- Sys.time();2+2;end <- Sys.time();print (end-start)
[1] 4
Time difference of 0.0001068115 secs
```

While it may not always be feasible to use addition instead of multiplication, this general rule extends to functions as well. String functions are often good candidates. For example comparing the start of a string variable with a literal value is less expensive than searching the string variable for the literal. In SAS terms, the *=:* operator is more efficient than the *index()* function.

4.2 Input/Output

As in the case of CPU time, the most important principle is to avoid unnecessary I/O. One should also be aware that the speed of storage devices and the access to them varies. The hierarchy starts at the fastest end with RAM disk follow by disks which are attached directly to the computer. Next comes network attached storage on the local network and, at the slowest disks which are accessed via a wide area network. Moving I/O operations to a faster device is often the most efficient way of achieving performance gains.

The discussion above on placing operations which are invariant in a loop applies also to I/O. This is a factor which can have a much more significan effect on performance in comparison to calculations.

5 Know your programming language

While the general rules described in the previous section apply to all languages, the circumstances in which they apply and the effect they produce may vary from language to language. One reason is that the compilation of code usually includes the application of optimization techniques but precisely which are applied vary from language to language. Furthermore for a given language, these may vary over time as that language is developed. As an example, consider the optimization for SAS suggested by Cody [2] of ordering the conditions of a series of *if/then/else* statements or the *when* clauses of a *select* statement according the frequency with which these conditions are encountered in the data. Since both *if/then/else* and *select* statements exit after the first true condition, this avoids testing conditions unnecessarily and therefore saves CPU time. While this was true at the time, a simple test shows that on current computers with current versions of SAS, this ordering has no appreciable effect. At the same time, this suggestion is also to be found with respect to the *switch* construct of the C language in Oualline [8]. Testing the same suggestion for C shows that it does make a difference.

A second reason for the variation is that languages are designed with different purposes in mind and contain unique features which provide performance advantages. An example is the hash or associative array found in some languages, e.g. Perl, which provide extremely efficient ways of performing table look-up. SAS has always had such a feature although it has often not been recognized as such: formats can be used for efficient table lookup[2].

For the C programming language placing loops with fewer iterations on the outside reduces the number of times the inner loop has to be initialised as described by Oualline [8]. The same is true for SAS as can be demonstrated by a simple program. Although the effect is minimal, it may be relevant in certain circumstances.

6 Optimization and developer time

In this section we discuss how the law of diminishing returns applies to performance improvement but suggest that programmers should learn to program for performance as second nature.

In his book on C programming [8] Oualline's first piece of advice on optimization is that one should not do it. His reasoning is that unless a program breaks one or more of the rules, optimization will be very difficult and since good programs will follow the rules, there is little scope for optimization. He suggests that a more fruitful approach is to add resources by upgrading the hardware and this is sometimes the only solution. If one of the rules is broken significant improvements can be achieved as described in examples in section 2. If such dramatic results can be achieved, however, it is almost certainly not worth spending development time of finding further improvements — once the broken rules are fixed, much effort is required to achieve better performance. As a programmer, one can easily learn to avoid after-the-fact performance improvement exercises by making techniques such as those described in this paper part of one's style of programming. None of the techniques actually require additional effort, only the awareness of the impact on performance of certain styles of programming.

[2]SAS 9 also has hashes but these require that the programmer pay a much higher price in terms of writing code in comparison to the implementation in other languages

7 SAS-specific considerations

In this section we discuss performance related topics which are specific to SAS. Before considering these in detail, it is worth pointing out that SAS is primarily an I/O intensive language, at least as far as most applications built with SAS are concerned. The I/O intensive nature of SAS results from the general sequence of SAS programs which consist of a series of steps, data or procedure, which read data or datasets and produce output or datasets, i.e. perform many I/O operations to disk. Exceptions to this rule are some statistics procedures, especially when exact calculations are specified.

7.1 DATA and PROC steps

7.1.1 Reducing the amount of data read or written

The easiest way to minimize I/O is to used *keep* or *drop* options to ensure that only necessary variables are read or written. This can have a significant effect. For example a dataset with 275,000 observations, two short character variables and five numeric variables took 0.25 seconds to read using the following data step:

```
data new ;
   set data ;
run ;
```

Adding a keep option to the input datasets retaining only three of the seven variables reduced the time to 0.18 seconds, a saving of approximately 25%.

A second easy to apply technique is to use *where* clauses or statements instead of subsetting *if* statement when only some observations are to be read. SAS can use where processing to read only the observations matching the criteria whereas a subsetting *if* reads every observation and then discards those which do not match. Both the *where* clause and statement allow complex expressions involving data step functions and are therefore quite flexible. The only point to note here is that, when combined with *keep* or *drop*, variables used in the *where* clause must be kept.

A final technique in this category is to avoid unnecessarily re-creating a dataset. For example, it is common to see code such as:

```
data a ;
   set a ;
   format var fmt. ;
run ;
```

This code reads every observation in the dataset and writes it out again in order to assign a format. The same can be achieved without re-creating the dataset as follows:

```
proc datasets lib=work nolist ;
   modify a ;
   format var fmt. ;
quit ;
```

A related technique is to avoid a proliferation of data steps when the result can be achieved in a smaller number of steps and reducing the number of steps does not obfuscate the code.

7.1.2 More efficient I/O

Indexes on SAS datasets represent another way to improve performance in two ways. Indexes can be simple (defined on a single variable) or composite (defined on multiple variables). The index is effectively a map of where in the dataset observations with particular values are located. First, they can be used by *where* to speed up finding observations when some of the variables in the clause are part of an index as described in Chapter 2 of Mason [7]. There is an initial cost in building the index which has to be balanced against the performance gain, but a typical case where the benefit outweighs the cost is when different subsets of a large dataset are read for analysis. Indexes should not be used blindly as there are cases where sequentially reading the data is faster. Generally, there should be a large number of distinct values for the variables in a simple index or combination of values for a composite index for a net benefit. A further consideration is that if the data occur randomly in the dataset, indexes are generally not useful. Further details on these conditions are described in Chapter 28 of the SAS Language Concepts manual. [5]

A second use of indexes is to avoid sorting. Reading a dataset using a *by* statement with variables of a composite index, in the same order as the index, obviates the need to re-sort the data.

7.1.3 Writing to LOG

Writing to the SAS log is also I/O and can become very expensive. A program which ran some simple macro code and generated a 280k log file and 700k lst file took 3 minutes to run. The same program, with macro debugging options turned on generated a 630k log file and took an additional 6 seconds to run. The effect is not necessary linear as a simulation program which processed twenty four scenarios in 6 hours ran in 3 hours when the notes option was turned off.

7.1.4 BUFSIZE, BUFNO and SASFILE

If memory is not a constraint, it is possible to increase the size and number of the I/O buffers that SAS uses in order to reduce the number of I/O operations and therefore improve performance. The general rule is that increasing the buffer size or the number of buffers will speed up both reading and writing. The BUFSIZE option, to have any effect, must be specified either as a system or dataset option when the dataset is being created and becomes a permanent attribute of the dataset which can be examined using the CONTENTS procedure or the properties window in interactive SAS. The number of buffers used is a transient attribute and may also be specified as either a system or dataset option. There is an interaction between the two parameters and the size of the observations, so increasing these requires testing to find the best values. The SAS Language Dictionary [6] recommends not setting the number of buffers to be greater than 10. Experiments with a datasets containing 5 numeric and two short character variables and a total observation length of 44 showed that setting the number of buffers to 4 achieved the best results except when the buffer size was increased above 64k. In general it is best to make the buffer size an integral multiple of the record length of the SAS dataset.

The *sasfile* statement allows a dataset to be loaded into memory and, according to the documentation [6] avoids repeated I/O when the dataset is repeatedly read. The statement may, however, have no effect if the system has sufficient memory and the disk cache is large enough. Here it is important to measure the effect in each case in order to determine if the technique provides a benefit.

7.2 SAS Macro

The first question to ask oneself when writing macros, as Carpenter [1] suggests in his chapter on efficiency issues, is whether to use the macro language at all. Reusable code which required no conditional execution can also be achieved using macro variables and the *%include* statement, and for such restricted applications is more efficient than using the full macro language as the macro compilation and macro execution steps are not necessary.

Given that the full macro language is required, there are a number of points to consider. First, one should avoid nesting macro definitions unless the code of the nested macro itself is dynamic. The reason is that nested macros are *compiled* every time the enclosing macro is executed. If the nested macro does not change between compilations we have a clear case of unnecessarily repeating operations.

A second feature to avoid is the command or statement style macros. These options allow macros to be called by naming them without a leading percent sign, without parentheses and with spaces instead of commas separating the parameter assignments, making them look like SAS statements. This adds an overhead as the SAS supervisor cannot easily recognize macro. It also has a potentially undesirable side-effect in that a macro with the same name as a SAS keyword will mask that keyword.

The SAS Macro Language Reference [4] in Chapter 11 points out that resolving a macro variable is always less expensive than executing a function so repeated calls to a function with the same arguments, e.g. if in the following loop the value of the macro variable text does not change:

```
%let x=1 ;
%do %while (&x < %length(&text)) ;
    ...
%end ;
```

it is more efficient to write:

```
%let x=1 ;
%let textlen=%length(&text);
%do %while (&x < &textlen) ;
    ...
%end ;
```

Finally, a limited number of macro variables are stored in memory. This limit is defined jointly by the SAS system options *msymtabmax* and *mvarsize*. The former specifies the total space available for storing macro variables in memory and defaults to 4MB on UNIX and Microsoft Windows systems. The latter parameter

specifies the maximum size of an individual variable which can be kept in memory and defaults to 32k. If memory is available, these can be increased to avoid I/O. Setting either to zero causes all macro variables to be stored on disk. A related tip is to use the *%symdel* statement to delete macro variables which are no longer needed.

8 Measuring performance

Some of the techniques described, such as avoiding invariant statements in loops or using *drop* and *keep* do not require any measurement - these are techniques which can always be applied. In some cases, however, it is necessary to make measurements. SAS provides such information using the *stimer* and *fullstimer* options. The former generally shows elapsed (real) and cpu time for a step while the latter also shows information on I/O and memory management. Details vary according to operating system so it is necessary to consult the relevant documentation. This information can be used when the techniques apply to complete steps and can be collected without needing to change the code.

Since programs are generally run on multi-tasking, multi-user operating systems, some of these measurements, and in particular elapsed time, vary according to other activity on the system. Since elapsed time is generally the most interesting measurement, this means that a single measurement is not sufficient to draw conclusions. It is necessary to repeat measurements and analyse the results statistically. The distributions of measurements tend to be non-normal, even after log transformations, so non-parametric tests are probably the best approach. It is relatively easy to write SAS code to parse the log files and collect the data for analysis.

For measuring effects within a data step, collecting the information involves inserting code to record start and end times for analysis. Here it is easier to use test programs designed for the purpose: for example, the following macro was used to test the assertion that SAS does not optimise invariants withing loops:

```
%macro test(n=30,iter=20000,obs=1000) ;
   data test ;
      test="Iter: &iter / Obs: &obs" ;
      do i=1 to &obs ;
         a=1 ;
         b=2 ;
         c=3 ;
         output ;
         end ;
      stop ;
   run ;

   %do k=1 %to &n ;
      data _measures_ ;
         set test ;
         start=time() ;
         do i=1 to &iter ;
            x=mean(1,2,3) ;
            end ;
         end=time();
         seconds=end-start ;
         order='inside loop ' ;
         output ;

         start=time() ;
         x=mean(1,2,3) ;
         do i=1 to &iter ;
            end ;
         end=time();
         seconds=end-start ;
         order='outside loop' ;
         output ;
      run ;

      proc append base=measures new=_measures_ ;
      run ;
```

```
      %end ;
%mend ;
```

9 Conclusion

Improving the performance of programs is both an art and a science. The art lies in knowing when to spend effort on performance improvements, the science in understanding the factors which influence performance and how they behave in given circumstances. It is possible to develop programming and program design styles which avoid most issues related to performance and which require only that the programmer learn both the art and the science. Efficient programs save time which can then be spent on more important activities.

References

[1] Art Carpenter. *Carpenter's Complete Guide to the SAS Macro Language*. SAS Institute, second edition, 2004.

[2] Ron Cody. *The SAS Workbook*. SAS Institute, 1996.

[3] Richard A DeVenezia. Demonstrates lack of optimization by data step compiler. advice: Don't use constant() inside a large loop. http://www.devenezia.com/downloads/sas/samples/constant.sas, September 2004.

[4] SAS Institute. *SAS 9.1 Macro Language: Reference*. SAS Institute, 2004.

[5] SAS Institute, editor. *SAS 9.1.3 Language Reference: Concepts*. SAS Institute, third edition, 2005.

[6] SAS Institute, editor. *SAS 9.1.3 Language Reference: Dictionary*. SAS Institute, fifth edition, 2006.

[7] Phil Mason. *In the Know... SAS Tips & Techniques from Around the Globe*. SAS Institute, 1996.

[8] Steve Oualline. *Practical C Programming*. O'Reilly & Associates, 1993.

Acknowledgements

Tom Keane, formerly of the Central Statisics Office, Ireland for encouraging me to think about and investigate performance.

Contact Information

Martin Gregory
Data Integration and Statistical Systems
Merck KGaA
Frankfurterstr 250
64293 Darmstadt
Germany
Martin.Gregory@merckgroup.com