Introduction in Efficient SAS® Programming via Incremental Refinement by Example

Paul M. Dorfman
Citibank Universal Card, Jacksonville, Fl

Abstract

How to write efficient SAS programs? Should you allow performance considerations to be a hurdle on the way of coding a workable program by suspiciously looking at each statement as a potential resource hog? Or should you keep efficiency in the background until a working prototype has been devised, and then refine its performance incrementally?

As the title suggests, not only the author is rather inclined to accept the latter notion, but also he has the audacity to use real-life SAS examples to prove that this is the way to go! He also dares to assert that by applying this process repeatedly and accumulating the knowledge about high-performance techniques, a SAS programmer acquires the ability to look ahead and program efficiently from onset.

Introduction

SAS programs are written to deliver accurate information on time. That is why writing efficient code is important. A program unable to complete because it runs out of resources or keeps running past the time its results can be used, is one extreme. From this standpoint, writing efficient code is not just the question of programming aesthetics, but a vital, real-world concern.

However, an efficient program may take longer to write than a quick-and-dirty one. A program finishing in a millisecond and spending zero resources but taking eternity to develop is as useless as one that runs forever. This is the opposite extreme.

A useful SAS program resides somewhere in the middle between the two extremes, not really close to either one. Developing an efficient program is important, but almost as important is to spend programming resources judiciously. The question is, if there is a good approach that would allow to strike the balance.

Unfortunately, attempts to write an ultimately fast, efficient program at once rarely lead to the goal. This is because too much time gets spent optimizing things that afterwards prove to be insignificant. The irony is that usually only quite insignificant part of code governs the speed of a program in any significant way. However, such part, also known as the inner loop, can only be identified after the skeleton of the program is ready. Therefore, it might make sense to attack the efficiency issue in two stages:

1. Write a workable running prototype trying to avoid major and obvious inefficiencies.
2. Run the prototype and establish a tentative benchmark.
3. Identify the inner loop.
4. Make every endeavor to optimize the inner loop comparing the run-times with the previous benchmark after each improvement.
5. See if any other performance enhancements can be made with little effort.

In this tutorial, this approach is illustrated by taking a simple real-life problem and following the steps as outlined above.

The Sample Problem

You have a large “production” SAS file MST.M (master) intended to keep track of customers logging on the company’s site. It contains many variables, but only several of them are of interest: K (the key, numeric), E (current e-mail address, $24), and P (current phone, numeric). Note that even though the file is sorted, it is sorted by account number, while K represents an encrypted key used for communication with vendors. Therefore, with respect to the key, file MST.M is disordered. The fields in the file might resemble something like this:

K	E	P
1849625698220742	qpwoei@hartz.com	2593987194
9216025778658700	mqiwuey@hotmail.com	5316917697
0497940262080143	vbdjhf6@jvlnet.com	5238705691
. . . . .		

You receive an e-mail (from whomever has a right to send you such e-mails) containing a short list like this:

8533943108531620	afyq7q@epicor.com	2971940344
2726117890666294	q387qh@aol.com	2265075959
6882365502827971	ljgfdsp@ix.netcom.com	2872256820
4757893050442400	qwertymew@pacbell.net	5903647102
5825815264054488	gwqb7t9q@gateway.net	5066035788

The list is followed by a request to extract the variables K, E, P from the master file and create an updated file UPD.U, where the fields in the records matching the keys on the list are overwritten with the new values of the address and phone. The sender also wants to know how many records have actually been updated. Pure and simple, and of course, the updated file is needed *yesterday* for some kind of urgent marketing initiative.

"Wallpaper" Approach

The UPDATE statement seems to be banging on the door, but since the file is not sorted, no good use can be made of it. On the other hand, being against the negative deadline, you do not feel like getting fancy. So, you cut-and-paste the data from the e-mail to your editor and insert them into the code, coming up with the following, say:

```
data upd.u;
   set mst.m (keep=k e p) end=eof;
   select (k);
   when (8533943108531620) do;
      e = 'afyq7q@epicor.com';
      p = 2971940344         ;
      u ++ 1;
   end;
   when (2726117890666294) do;
      e = 'q387qh@aol.com   ';
      p =  2265075959        ;
      u ++ 1;
   end;
   when (6882365502827971) do;
      e = 'ljgfdsp@ix.netcom.com';
      p =  2872256820            ;
      u ++ 1;
   end;
   <... rest of the case structure ...>
      otherwise;
   end;
   if eof then put u = comma.;
run;
```

The variable U is intended to print the number of updated records in the log. You submit the program, it runs just OK first time in, the customer is happy, and so are you. All the more that the program runs quite fast.

But the happiness is short-lived, because in a couple of days, you receive another e-mail of the same nature. This time, however, it has some 100 plus lines of data. You still manage to change your program in a matter of an hour, run it, and create the much needed file on time. However, this time, not everything goes as smoothly. First, there are those pesky typing errors, and the program has to be resubmitted several times. Secondly, it runs noticeably slower than the first variant. This is unpleasant, but at least logically justifiable: Finding a key among 100 entries takes longer than among five.

The real problem arises the next time you receive a similar request, because now it does not contain any data per se. Instead, it has a reference to a SAS data file with upwards of 1,000 records. The perspective of printing it out and doing the same boring typing thing all over again is not inviting at all! Much more importantly, with this many records, the probability of making a typo – and hence that of creating a corrupted file – is very high, virtually guaranteed, no matter how good and attentive a typist you are.

Finally, if you have attended Ian Whitlock's SUGI 24 lecture about the relationship between code and data, you understand that the code above qualifies for Ian cleverly calls *"wallpaper"*. Your data (transaction records) already reside separately, and embedding them in your code is a bad habit and principal mistake.

Increment 1: Automate

The entire purpose of computers is to automate mundane, error-prone processes. This one seems just like a perfect candidate. Why not read the attached file into an array in memory and use it as a lookup table? This way, the data will stay where they belong – in a separate file, while the program will only tell the computer what to do with the data, and this is exactly what and only what any program should do. The plan is simple:

1. Store all the information from TRN.T in parallel temporary arrays.
2. Read a record from MST.M, and search the array containing the keys for K.
3. If the current key read from MST.M is found in TRN.T, overwrite the variables E and P by moving data from the corresponding array cells.
4. Write the record out and go to step 2.

Naturally, the array has to be populated before the very first observation from data set M has been read. A "standard" way of making it happen is to test if the automatic variable _N_=1 and take the action only if the condition is true.

How large should the arrays be? Obviously, they should have at least as many entries as there are observations in TRN.T. You have already run CONTENTS and know that the file has 1009 records, so you may decide to size the arrays accordingly. However, thinking again in terms of code and data (you are trying to get rid of the wallpaper!), every time the size of the transaction file changes, the manual process of running CONTENTS and retyping the upper array bound in the program will be required. If you have decided to automate, then automate! The simple step below extracts the number of observations from TRN.T and populates the macro variable DIM:

```
data _null_;
   call symput('dim',compress(put(dim,best.)));
   stop;
   set trn.t nobs=dim;
run;
```

If the macro variable DIM is used to size the arrays, the program will adjust itself without the need of manual intervention. Finally, there is no need to output the array elements as variables, so temporary arrays can be used. It will save a lot of compilation time and relieve the compiler from keeping track of hundreds of unneeded variables. Now the plan of attack can be executed:

```
data upd.u (drop=j u);
   array ka (1: &dim)     _temporary_;
   array ea (1: &dim) $24 _temporary_;
   array pa (1: &dim)     _temporary_;
   if _n_ = 1 then do j=1 to &dim;
      set trn.t;
      ka(j) = k;
      ea(j) = e;
      pa(j) = p;
   end;
   set mst.m (keep=k e p) end=eof;
   do j=1 to &dim;
      if k ne ka(j) then continue;
      e = ea(j);
      p = pa(j);
      u ++ 1;
      leave;
   end;
   if eof then put u = comma.;
run;
```

This piece looks and feels like a program rather than a wallpaper. Actually, it can be considered a viable prototype, because:

1. It is automated. You no longer need to type anything taking care of the program. Instead, the program takes care of itself – and you.
2. It works, and it produces the correct output.
3. It completes before the deadline.

Creating a simple workable prototype similar to the above is very important not only because it is a correct program executing on time and giving you some peace of mind (for now). It is also important because it can serve as a pad, from which incremental performance improvements can be launched.

One of the concerns you may have running the code is that with 1,000 transaction records, the program takes almost 3 hours to run against 20 million records in the master file. This concern might be somewhat muffled by the fact that if the code is submitted in the morning, and the file is due the next morning, there is nothing to worry about. Yet there is:

1. There is no guarantee that the next time you do not have 10,000 transaction records, in which case the deadline will definitely be missed. And 10,000 records is still not a whole lot, since having 1 million transaction records is not unusual at all under such circumstances. What will happen then? 3,000 hours is definitely not the amount of time you want any business program to run, let alone one whose output is needed yesterday.
2. Intuitively, it is apparent that something must be not quite right, for you know that without updating, it takes only minutes to read the master file and write the necessary variables out. So there has got to be a better way.

Since, for the time being, the request is off your back, there is some time to take a fresh look at the code and try to find ways of speeding it up. But to be able to improve performance of the program, you have to identify its *inner loop* first.

The Inner Loop

The inner loop is a block of instructions executing much more frequently than any other part of the program. Typically, the difference between the number of iterations of the inner loop and other pieces of code constitutes an order of magnitude or more. Because of that, the speed of the inner loop governs the speed of the entire program. Any extra instruction, utterly insignificant when it stands alone, acquires a huge weight when executed as part of an inner loop.

It suggests two principal ways of making a program run faster:

1. Reduce the number of instructions in the inner loop to the bare minimum.
2. Reduce the number of times the inner loop iterates.

It is easy to identify the inner loop in the prototype above directly from its definition. We have three loops in the program. The loop loading the array cannot be the inner loop, for it executes only once. The outer loop - the implicit observation loop iterating each time to read a record from the master file – is a better candidate, since it executes as many times as there are observations in MST.M. However, the loop

```
do j=1 to &dim;
  if k ne ka(j) then continue;
  e = ea(j);
  p = pa(j);
  u ++ 1;
  leave;
end;
```

is an even better candidate. Why? Because for each record read from the master file, this loop goes over all

array items N times (assuming that KA() has N items), if the current master key K is not present in the array. And if it is present, the loop iterates, on the average, N/2 times, for it is equally likely, on the average, to locate the key at the beginning and at the end. So, with 1,000 transaction records and 20 million master records, the loop executes, in the average, 30 billion times! Since there is no other loop nested inside it, the "DO J=" is the inner loop of the program.

Identifying the inner loop is a program as simple as this is not difficult. With more convoluted real-life programs, it may not be as easy. However, there is almost a sure-fire way of pinpointing the most frequently executing block by inserting counters in likely locations, running the program against a fairly large test file, and printing the counters. Effort spent on diagnostics like that is usually richly rewarded thereafter.

Increment 2: Making Inner Loop Leaner

Now that you know where the likely culprit is, it makes sense to concentrate intellectual efforts on making the inner loop as lean as possible, for this is where the greatest performance-improving potential is lurking. Because of the CONTINUE, the block of instructions

```
e = ea(j);
p = pa(j);
u ++ 1;
```

is performed only if and when the master key K is actually found, and therefore only once per master record. If the key is not in the transaction file, this instruction block is not performed at all. So, effectively, the instructions are not part of the inner loop, and then none of them can be left out, anyway. The instructions executing each time the inner loop iterates are:

1. Explicit comparison IF K NE KA(J) at the top.
2. Implicit comparison IF J > &DIM at the bottom.
3. Implicit increment J=J+1 at the bottom.

At first glance, both comparisons seem to be necessary. In reality, the second of them is actually excessive! At every iteration, the instruction asks the computer: Is it the end of array yet?

The only reason the second comparison is in the loop is preventing the index J from running past the upper array bound. But the same thing can be accomplished simply by placing the search key itself as stopper (sentinel) to the right of the upper bound of the array. Then all we have to ask is: Does the current array item KA(J) equal the search key K? If the answer is 'yes', the pointer is either at the end of array (and so the key is not found), or it is still within the array bounds (and so the key is found, and the loop can be terminated). To implement the improvement, another element in array KA() is needed. Without changing anything else, it can be accommodated by redefining the array as

```
array ka (1: %eval(&dim+1)) _temporary_;
```

Now the shrunk inner loop can be written as follows:

```
ka(hbound(ka)) = k;
do j=1 by 1 until (k = ka(j));
end;
if j <= &dim then do;
   e = ea(j);
   p = pa(j);
   u ++ 1;
end;
```

The first assignment moves the stopper value to the last item of KA(). Even though this instruction is added to the previous code, it is added outside the inner loop. For 1 such extra instruction executed outside the loop, 1.5*N comparisons "IF J > &DIM" are eliminated. The same is true about the comparison "IF J <= &DIM". Note that the inner loop has become a null loop – its body is empty, but it does not mean it does nothing! Its only purpose now is to locate the slot whose content equals K and return its index J.

Benchmarking the program with the change discussed above results in 20 to 30 percent shorter run-times. And this is achieved just by eradicating a single implicit comparison from the inner loop. However, the largest performance-improving potential lies in a radical reduction of the number of times the inner loop iterates.

Increment 3: Making Inner Loop Iterate Fewer Times

The algorithm used in the prototype program is nothing else but plain sequential search. The scheme improved by identifying the inner loop and making it leaner is sometimes called quick sequential search. Although the latter is more efficient, both are similar in that they rely on a 2-way decision:

1. K = KA(J) .
2. K NOT= KA(J).

And it is the only way of going about the searching business if the array is not organized in any particular way but simply searched as is. However, organizing a table in an intelligent manner ahead of the time can (and does) make a gigantic difference.

Out of great many ways items can be arranged to facilitate efficient searching, the simplest is sorting. Having the elements ordered, we possess enough knowledge about them beforehand to make a judgement about the location of the search key in the

lookup array without comparing it to all or the majority of items. Hence, it should be expected that the order relation, if exploited properly, would result in a major performance gain.

Rearranging the arrays into order by key is as simple as sorting the transaction file by K:

```
proc sort data=trn.t; by K; run;
```

But how exactly can we take advantage of the order relation between the keys? Suppose we have chosen some array element KA(J), sometimes called *a pivot*. That will divide the array into three parts: the items in the locations lower than KA(J), KA(J) itself, and the items in the locations higher than KA(J). Accordingly, three mutually exclusive outcomes are possible:

1. K = KA(J). The key has been found.
2. K > KA(J). All elements with indices lower than J are eliminated from consideration.
3. K < KA(J). All elements with indices higher than J are eliminated from consideration.

Thus, whilst the sequential search is limited to a *2-way decision* (equal or unequal), ordering enables us to continue search based on a 3-*way decision*. Regardless of the way it branches, substantial advance has been made. However, it is important to choose J correctly. If the only fact known about the keys is that they are sorted, the choice suggesting itself naturally is to start searching by comparing K to the *middle* item in the array. The result of the comparison will either locate the right key or tell which half to search next, and the same process can be used again. As a result, after at most about log(N) iterations, we will have either found the key or established that the array does not have it.

The basic idea of this widely known *binary search* seems to be transparent. However, it is surprisingly easy to code it wrong! One of the most popular correct ways of implementing binary search is to maintain three indices: L pointing to the lower limit of the current search interval, J pointing to its middle, and H pointing to its upper limit. Then you can proceed as follows:

1. Initially, set L and H to the lower and upper bounds.
2. Compute J as the half-midpoint between L and H.
3. If K < KA(J), set H to (J-1) and go to 6.
4. If K > KA(J), set L to (J+1) and go to 6.
5. If K = KA(J), the key has been found. Terminate.
6. If the pointers L and H cross then stop, else go to step 2.

Expressed in terms of the SAS Language, the binary-search-based inner loop may look as follows:

```
l =    1;
h = &dim;
do until (l > h);
   j = floor((l + h) * .5);
   if      k < ka(j) then h = j - 1;
   else if k > ka(j) then l = j + 1;
   else do;
      e = ea(j);
      p = pa(j);
      u ++ 1;
      leave;
   end;
end;
```

Note that binary search contains more instructions in the inner loop than even plain sequential search. Every time the inner loop iterates, two comparisons are made: one inside the loop, and one (L > H) at its bottom, plus some extra time is spent calculating the midpoints. But it is more than alleviated by the fact that to either find or reject a key, binary search iterates, on the average, only log(N)+1 times, not N. Effectively, it cuts the number of inner loop iterations to 11 times for N=10,000 and to 21 times for N=1,000,000.

The performance consequences of this relationship are hard to overestimate. The update job, that used to run for 2 hours with quick sequential search, now takes only several minutes. But the most important advantage of the method is its scalability. Because of its O(log(N)) nature, the run-time increases just twice when the size of the transaction file grows full 3 orders of magnitude. That is why you do not have to fear that the next file that might be thrown at you may have 10,000 or more transactions. Benchmarking shows that even with 100,000 observations in TRN.T, the binary search program takes only about 12 minutes to run against a 20-million-record master file.

Increment 4: Buy None, Get One Free

Let us sum up what has been done so far. You started with a wallpaper-type program, then created a decent working prototype based on sequential search. Taking a single comparison instruction out of the inner loop allowed to cut the run-time 20 to 30 per cent. Replacing sequential search with binary search reduced the number of times the inner loop iterates orders of magnitude. As a result, the run time, especially with large transaction files, is reduced orders of magnitude as well.

After the giant leap the last incremental improvement offers, it is, generally speaking, almost impossible to significantly improve performance remaining within the domain of lookup methods based on comparisons between the search key and keys in the table. Direct-addressing techniques, discussed elsewhere at this meeting, are able to cut run-times several times even compared to binary search.

However, the threshold we have achieved so far is high enough for the purposes of the original task. The program is automated, it runs correctly, scales well, so let us see how the last, fastest so far, version looks like (last step only, trn.t being sorted by K is assumed):

```
data upd.u (drop=j u);
   array ka (1: &dim)     _temporary_;
   array ea (1: &dim) $24 _temporary_;
   array pa (1: &dim)     _temporary_;
   if _n_=1 then do j=1 to &dim;
      set trn.t;
      ka(j) = k;
      ea(j) = e;
      pa(j) = p;
   end;
   set mst.m (keep=k e p) end=eof;
   l =    1;
   h = &dim;
   do until (l > h);
      j = floor((l + h) * .5);
      if      k < ka(j) then h = j - 1;
      else if k > ka(j) then l = j + 1;
      else do;
         e = ea(j);
         p = pa(j);
         u ++ 1;
         leave;
      end;
   end;
   if eof then put u = comma.;
run;
```

In principle, the code can be left alone this way. But let us look at the program with a critical eye just one more time and see if there is anything else that can be saved without huge sacrifices in brainpower and time.

Now with binary search in place, the inner loop iterates only about 10 times per record read from the master file, so some modest performance improvement can be achieved by going after extra instructions outside of the inner loop, if any such instructions exists. They may be worth something: The outer loop, after all, iterates 20 million times, and with the 10:1 ratio between the number of inner and outer iterations, there should be, speaking intuitively, a chance to save another 10 per cent of the total run-time.

There are only two suspicious outer-loop instructions: “IF _N_=1” and “IF EOF”. The first comparison can be indicted on the ground that we only need to load the array once, so why should the computer be compelled to ask the question “Have I already started doing anything?” every single time a master record is read? Likewise, if the number of updates U must be printed only once after the entire file has been processed, why should the computer evaluate the condition “IF EOF” 20 million times? Straightforward logic tells that both actions – loading the array and printing the final value of the number of updates – should be done no matter what, so why do we need those IFs?!

The answer is: We do not. Because of the force of inertia, it is customary in SAS to leave the automatic observation loop intact, trying to do whatever has to be done inside it. However, this practice flies right in the face of the most fundamental programming doctrine: All those instructions that can be performed outside a loop, do not belong inside it. What IF _N_=1 et al. amounts to from the standpoint of common-sense programming can be easily understood from the following example. Imagine that you have an array A() with 1 million, elements and you need to: a) print the first element; b) increase each element twice; c) print the last element. Which translates into the SAS Language directly as:

```
put a(lbound(a))=;
do j=lbound(a) to hbound(a);
   a(j) = a(j) * 2;
end;
put a(hbound(a))=;
```

That is it. Now imagine you reaction if someone suggested that the right way of doing this were

```
do j=lbound(a) to hbound(a);
   if j=lbound(a) then put a(j)=;
   a(j) = a(j) * 2;
   if j=hbound(a) then put a(j)=;
end;
```

You would perhaps say: “Wait a minute! Why should I test those IFs 1 million times each? Why not do the prints outside the loop?” And be exactly right, because the IF clauses above are not modified at each iteration of the loop, and therefore they need not be there. Yet testing for _N_=1 and EOF follow precisely the same pattern. To place the unconditional before- and after-loop actions where they belong, it is necessary to make the automatic observation loop explicit:

```
data upd.u (drop=j u);
   array ka (1: &dim)     _temporary_;
   array ea (1: &dim) $24 _temporary_;
   array pa (1: &dim)     _temporary_;
   do j=1 to &dim;
      set trn.t;
      ka(j) = k;
      ea(j) = e;
      pa(j) = p;
   end;
   do until (eof);
      set mst.m (keep=k e p) end=eof;
      l =    1;
      h = &dim;
      do until (l > h);
         j = floor((l + h) * .5);
         if      k < ka(j) then h = j - 1;
         else if k > ka(j) then l = j + 1;
         else do;
            e = ea(j);
            p = pa(j);
            u ++ 1;
```

```
            leave;
         end;
      end;
      output;
   end;
   put u = comma.;
run;
```

But do not we eliminate just one condition instead of two by still testing for EOF at the bottom of the explicit outer loop? No, because in the previous version, EOF was actually tested twice: Once implicitly, once explicitly, and in this version, it is tested just once.

Rounding off this seemingly small rough edge does not take any more programming effort than the "standard" coding, but yields 2-3 per cent shorter run-time. Is it worth bothering? Well, why forfeit performance improvement, even several per cent of it, if it is free? More importantly, the structure of the last version is vastly superior to that using the automatic loop, because it a) perfectly aligns with the logic of the pseudocode, and b) is much more extendible. For example, if, after the master file has been processed, you would like to read and process yet another file, it can be done in the same step without breaking any logic – another explicit file-reading loop simply follows the first one, and so on.

Conclusion

Donald E. Knuth once quipped in writing that "premature optimization is the root of all evil". It is a waste of time to spend it trying to find small efficiencies and looking at every statement suspiciously before the most critical part of the program, its inner loop, has been identified. A program approached like that will be, most likely, never finished on time.

Instead, it makes sense to write a good working prototype avoiding performance blunders (such as not dropping 100 unnecessary variables from input) and locate its inner loop. That is where most of the performance gain can be obtained from. Without the prototype, it is difficult, and often impossible to tell a priori where the inner loop resides. Additionally, since the prototype is an already correctly working program in terms of its output, it can fulfil immediate needs.

Each improved working version of the program can then serve as a basis for further incremental improvements. In our simple example, taking a single comparison outside the inner loop immediately led to the version running 20 to 30 per cent faster than its predecessor. Over the course of the next increment, deploying a new strategy to cut the number iterations the inner loop goes through, improved performance an order of magnitude or better.

After the inner loop has been taken care of, other, more subtle, performance improvements can be considered. Those should be weighed against the time and mental effort they require to be implemented. And, most certainly, they should not be passed up if they are free!

Finally, incremental approach to enhancing the efficiency of coding is a good self-teaching tool. The knowledge garnered at different stages of making a program run faster using fewer resources accumulates into an experience letting a programmer skip the entire stages of the optimization process and thus reduce the time both the programmer and machine spend to accomplish the task.

References

1. D. E. Knuth, Literate Programming, CSLI, Stanford.
2. D. E. Khuth, The Art of Computer Programming, v. 3.
3. SAS Language Reference, Concepts, SAS Institute Inc., Cary, NC.
4. I. Whitlock, Code or Data? Proceedings of SUGI 24, Miami Beach, Fl, SAS Institute Inc., Cary, NC.
5. P. M. Dorfman. Array Lookup Techniques: from Sequential Search to Key-Indexing, Proceedings of SESUG'99, Mobile, Al.
6. P. M. Dorfman. Table Lookup via Direct Addressing, Proceedings of SESUG'00, Charlotte, NC.
7. T. A. Standish. Data Structures, Algorithms & Sorfware Principles in C, Addison-Wesley, 1995.

Acknowledgements

Thanks to Ian Whitlock for suggesting a topic and inviting the author to present it at SESUG'00. Thanks to Ian Whitlock and Sig Hermansen for discussing a variety of SAS programming efficiency issues both tete-a-tete and on-line. Thanks to all SAS-L contributors who, by asking questions and replying to posts, keep the interest to efficient SAS programming alive.

Author Contact Information

Paul M. Dorfman
10023 Belle Rive Blvd. 817
Jacksonville, FL 32256
(904) 564-1931 (h)
(904) 954-8533 (o)
sashole@mediaone.net
paul.dorfman@citicorp.com