A technique for measuring and comparing the performance of existing computer systems is to devise a synthetic job that is simple enough to be programmed with a modest effort in different languages and on dissimilar machines, so as to be run and timed on each of the systems.

The job dercribed here is a greatly simplified file maintenance procedure, which exercises both the central processing unit and major input/output devices, with activity parameters being specified in a manner independent of the system. A complete PL/I *version is shown as an example. It is conjectured that such a synthetic job may evolve into a practical standard of performance.*

A synthetic job for measuring system performance

by W. Buchholz

In the absence of a theoretical definition of performance, we will here describe performance quantitatively in terms of the running time of a given job, or rather the reciprocal of this time so as to associate the faster machine with the larger number. To make performance a reproducible measure, we will, as far as possible, exclude hard-to-control variables, such as manual set-up and error recovery times. These important aspects of overall performance must be stated as factors reducing the ideal performance of a perfect system.

Thus performance is defined as a measurable quantity, but how do we select a job and how do we measure it? Compiling and executing a selection of well-defined mathematical routines, such as matrix inversion and polynomial evaluation, may be a reasonable work load to characterize a computing center running mostly small mathematical programs. In a data processing environment, it may be found that sorting occupies a significant fraction of the execution time, and programmed sorting routines may be available for comparing different systems. For most other applications, the effort of programming a nontrivial job and converting its data files to compare dissimilar systems is prohibitively expensive. High-level languages help to reduce the cost, but imperfect compatibility and dependence on particular hardware or software configurations still present substantial reprogramming problems. Consequently, the use of substantial "benchmark" jobs has been limited in practice to evaluating successor systems for a particular

application environment. Even large benchmark jobs must be recognized as models simulating the real world.

A greater degree of abstraction is necessary to make intersystem comparisons practical. Stripped-down but carefully specified file maintenance and mathematical routines have been described,[1-3] which permit skilled analysts to calculate from published timing data their comparative running times on different systems. Alternatively, a simulation program may be employed[4,5] to model the major actions of a particular job in a specific application environment while suppressing details that would unnecessarily extend the model writing and execution times.

The technique to be described here is to imitate the real application by a simple but complete synthetic program. For existing hardware and software, the execution time can be measured objectively, thus avoiding all assumptions regarding the behavior of complex hardware and software systems. For proposed hardware or software, evaluation by analysis or simulation of an executable program has the considerable advantage that its accuracy can be verified by direct measurement once the system is completed. To keep the program simple enough so that it can be readily reprogrammed, details of the application being imitated are intentionally suppressed. Hence, performance of a system on a synthetic program cannot be used directly to predict the running time of a specific application accurately. The relative performance of two systems on a synthetic job should, however, yield a reasonable first approximation to their relative performance on a specific job using the same system facilities.

A measurement standard

A synthetic job can serve as a standard measure of computer performance in the same sense that a yardstick measures length. The yard, or the meter, is a completely arbitrary but standardized measure of the length of an object. It does not measure other important characteristics, such as weight or power. A synthetic job can only serve to measure those characteristics of a system to which it is sensitive. Several different computer yardsticks may be needed; then one can choose that which will best measure the characteristics of interest. The objective would be to have only a few different but well-controlled yardsticks in general use, so that a measurement by one group may prove useful to many others and its significance can be more reliably interpreted.

One requirement of such a job is that it can be stated as a machine-independent procedure. Another is that it be meaningful over quite a wide range of computer systems, being neither too trivial for the larger ones nor too complex to be run on the smaller ones. It should not be so short that it cannot be measured accurately nor so long that measuring becomes burdensome. Consequently, the procedure should be cyclic with the running time directly proportional to the number of repetitions.

The yardstick job

The synthetic *yardstick* program, used here as an illustration, is modeled after the file maintenance procedure that is central to many data processing jobs. It makes heavy use of input/output or external storage devices. By providing a compute kernel of variable length, it is possible to simulate both input/output- and processor-bound situations, the latter being more representative of mathematical computing. The amount of storage required can be changed.

The input is a file of detail records. For each detail record, a file of master records is searched. When the matching master record is found, the compute kernel is executed, after which an updated master record and a detail output record are written. For sequential record organization on magnetic tape, the detail records must be in the same sequence as the master records, and all master records must be read and written during the search, whether active or not. Direct-access devices, such as magnetic disks, are not restricted to sequential processing. Randomly selected detail records can be processed directly against the active master records without searching through the inactive ones. After locating the desired master record and executing the kernel, the updated master record is written, replacing the old one. With random processing, there is only one copy of the master file.

The cyclic requirement is satisfied by designing a program whose running time is directly proportional to the number of records processed. Initiation and termination procedures, such as tape rewinding, are excluded from the measured time as far as possible.

The compute kernel is likewise a cyclic procedure, so that the ratio of compute time to input/output time can be defined in terms of a machine-independent parameter: the number of repetitions of the compute cycle. To stay on common ground among simple fixed-point machines, data processing machines with a good repertoire of alphanumeric and data-moving operations, and larger computers with specialized floating-point arithmetic units, the procedure chosen uses integer arithmetic. Thus the kernel does not attempt to measure the effectiveness of more specialized operations. After all, why use the speed of floating-point arithmetic subroutines as a basis for comparing a computing system designed for applications where such a feature is not important?

the kernel

The kernel is a simple mathematical exercise of summing n values looked up in a table in main storage. The table is set up beforehand to contain at least n^3 consecutive integers starting with an arbitrary integer A. To make the exercise nontrivial, the lookup intervals are varied: successive values to be summed are the first, eighth, twenty-seventh, etc., value in the table, that is, the kth value in the summation is the k^3th entry in the table. The summation result is then verified by an independent computation.

In mathematical notation, the table entries are

$$T_{i+1} = T_i + 1 \qquad (i = 1 \text{ to } n^3)$$

with $T_1 = A$. The procedure consists of computing

$$S = \sum_{k=1}^{n} T_{i_k}$$

where

$$j_k = j_{k-1} + 6\,u_{k-1} + 1$$

and

$$u_k = u_{k-1} + k$$

with

$$j_0 = u_0 = 0.$$

It will be seen that $j_k = k^3$, so that

$$S = \sum_{k=1}^{n} (A - 1 + k^3) = n(A - 1) + \sum_{k=1}^{n} k^3$$

Since

$$\sum_{k=1}^{n} k^3 = \left(\sum_{k=1}^{n} k\right)^2 = [n(n+1)/2]^2$$

the arithmetic can be verified by computing

$$B = \frac{1}{n}\left[S - \left(\frac{n(n+1)}{2}\right)^2\right] + 1$$

and testing for $A = B$.

A self-testing program is desirable to ensure accurate programming and proper operation of the system if the program is allowed to run for an extended time. Checks are also provided to verify input/output operation.

Note that A and n are parameters that affect the word length and storage space needed to carry out the computation successfully. If A or n are large, smaller computers would require multiple-precision arithmetic and fetching of data from secondary storage at a large sacrifice in speed.

the PL/I program

As a specific illustration, a PL/I version of the program is given in the Appendix. Only sequential organization of records on magnetic tape is shown for simplicity. The program is complete, including the generation of suitable master and detail files. It has an option to bypass tape and execute only the compute kernel. The program is self-timing on any system with an internal timer where the timing function of PL/I is implemented.

A corresponding program for direct-access devices, such as disks, needs some additional features. Random processing requires a random number generator for the detail keys. Even when sequential processing is desired, the spacing between successive detail keys should be irregular to smooth out the effect

of the fixed revolution time of disks and drums; otherwise, the running time would change only in quantum jumps of the revolution time as, for example, the kernel repetitions are increased.

An important part of random processing is the conversion of real record keys to addresses. It is suggested that for simplicity this problem be ignored here and small integers be used as keys, as is done in the sequential example in the Appendix. Such integers are easily generated in any desired sequence and converted to whatever addressing scheme is required by a specific device.

Note that for sequential processing the running time is proportional to the number of *master* records; a secondary parameter is the ratio of master to detail records, or its reciprocal, the activity ratio. For random processing, the time is proportional to the number of *detail* records, given a sufficient number to average out the random fluctuations; the number of master records is a secondary parameter that affects average access time but not in direct proportion. A time comparison between the two organizations can be made but only if the number of detail and master records are both specified.

Hardware and software comparisons

The yardstick program is readily programmed in different languages. As long as the procedure remains the same—that is, each program does the same thing in the same way—the running time of each version on the same hardware system provides a valid performance measure of the corresponding software.

If the program is written in machine language to be run independently of any software facilities, the execution time measures only the hardware facilities used. If an equivalent program is written in a higher-level language (FORTRAN, COBOL, PL/I, etc.), its execution time measures the efficiency of the object code produced by the compiler and of other software features, as well as the performance of the hardware. Indeed, the difference in execution time of two equivalent programs run in precisely the same way, one compiled from a higher-level language and one written in machine language, is a measure of the software efficiency alone.

It is possible, of course, to run a synthetic program as a combination compile-load-go job to bring in other software factors including job scheduling time, compile time, and file opening and closing time. Such a combination run lacks the cyclic nature of the execution phase and is, therefore, less easily controlled.

Multiprogramming

It has been assumed so far that the yardstick program is timed while running alone. In a batch processing system that can overlap the execution of more than one job, two additional types of measurement are of interest. One is the increase in total system

throughput resulting from multiprogramming, and the other is the slowdown of any single job arising from interference by other jobs being processed concurrently. For both measurements it is valuable to have a well-controlled job stream consisting of synthetic, parameterized jobs with known properties.

The number of jobs that can be processed simultaneously depends in part on the input/output, compute-time, and storage-space requirements, all of which can be varied in the yardstick program. A low value of performance is obtained by measuring a number of identical copies of the program running concurrently and competing for the same facilities. A higher performance is obtained by setting the parameters differently for each copy, so that a compute-bound program can run together with an input/output-bound program, a large program (in terms of storage space) with a small one, and so on. In each case, the measure of multiprogramming capability is the comparison of the total running time against the same job stream run one job at a time.

A time-sharing system responding to a demand for processing a task—such as compiling or executing a program or answering an inquiry concerning a record in a file—may be regarded as a multiprogramming system; the same type of job is appropriate for measurement as in a batch system if the corresponding facilities are provided. Some time-sharing systems, however, are restricted to specialized functions or languages, so that a more specialized workload must be designed.[6] Measurement does not start until an appropriate number of jobs have been entered from different terminals or possibly through a central facility. The terminals and communication network of a time-sharing system do not really enter into this measurement.

Response to a nonprocessing demand—such as editing a statement or returning a message—does involve terminals and network and requires quite different measuring techniques, which are not the subject of this paper.

Conclusions

The synthetic program described here can serve several functions. It may be simply a well-behaved exerciser of system features or a tool for comparing the speed of dissimilar systems. By writing the program in a procedure-oriented high-level language, it is easier to define the procedure precisely and to transfer the program to different machines, but the performance measure then includes language and software as well as hardware functions. The procedure is simple enough, however, that it can be reprogrammed readily in machine language for a particular system, so that hardware performance can be separated from software.

The particular program chosen for discussion is a highly stylized file maintenance procedure. No claim is made that the program is in any way representative of a real file maintenance application so that it could be used to predict the time on a particular job. Given a particular application, however, it may

be possible to use the synthetic program repeatedly with differently adjusted parameters in such a way that it approximates the sequence of steps of the real program. Whether such modeling of a real program can be made to track its performance on different systems remains to be determined, but the approach holds promise of greatly simplifying the technique of benchmark testing that is widely used in evaluating complex systems.

The objection may be raised that a synthetic program cannot represent all the complexities of a real program. It should be remembered, however, that any benchmark program separated from its original environment is but a model of the real job. When reprogramming a benchmark, there are many uncertainties of interpretation of the original intent and of human and other factors, which may introduce as much error into the comparison as the artificiality of a less costly synthetic program. The synthetic program can be much better controlled. At least the user can understand in detail what he is measuring and what the limitations are.

A synthetic file maintenance program alone may not be able to model all the steps of a real application. A single measure of performance also leads to the danger of designing a system that is "tuned" to this job. But if several dissimilar programs are used, a system that does well on all of them is likely to do well on real jobs. A small collection of parameterized procedures, imitating such operations as sorting and matrix computations, may well prove to be adequate standards of comparison from which a user can select those most appropriate for his application.

CITED REFERENCES AND FOOTNOTE

1. J. A. Gosden and R. L. Sisson, "Standardized comparisons of computer performance," *Information Processings 62* (Proceedings of the IFIP Congress 62), 57–61.
2. N. Statland, "Methods of evaluating computer systems performance," *Computers and Automation* **13**, No. 2, 18–23 (February 1964).
3. J. B. Totaro, "Real-time processing power: a standardized evaluation," *Computers and Automation* **17**, No. 4, 16–19 (April 1967).
4. P. S. Cheng, "Trace-driven modeling," in this issue.
5. P. H. Seaman and R. C. Soucy, "Simulating operating systems," in this issue.
6. For example, the computing kernel alone of the yardstick program has been used to measure a conversational system using the APL language, which does not make large file storage facilities available to the user. Hence the file processing part of the program was not applicable.

Appendix: PL/I version of yardstick program

The following listing shows a PL/I version of the yardstick program using magnetic tape for file storage. Four tape drives are needed, one each for the old master file (MASTER), new master file (NEWMAS), detail input (DETIN), and detail output (DETOUT)

tapes. If two tape channels are available, the two input tapes should be on one channel and the output tapes on the other. A card reader for entering parameters and a printer for indicating the results are incidental and do not have any effect on the measurement.

The first DECLARE statement, specifying file attributes, is implementation-dependent and may have to be modified for different systems. This may, or may not, be the place to specify the record lengths (200 characters), blocking factors (10 records per block for master files, no blocking for detail files), and degree of buffering desired.

Whenever a new tape processing run is specified, the program first generates master and detail input tapes before entering a full tape processing pass. During generation, these tapes are defined as output tapes (MASGEN and DETGEN); during processing, the same tapes must be defined as input tapes (MASTER and DETIN).

The program uses only features available in the PL/I subset as well as the full language. The program prints the elapsed time automatically using the built-in PL/I TIME function. (If an internal timer is not available, the DISPLAY statement may be used to stop the program for manual timing.)

For multiprogramming measurements, the printed elapsed time would include the slowdown caused by interference from other concurrent jobs. The program could be modified to print the values of START_TIME and END_TIME as well, so as to show the sequence in which the jobs were executed and to provide the complete time interval for all jobs, from the earliest value of START_TIME to the latest value of END_TIME.

Typical parameter cards are shown following the program. The cards are read during program execution to specify the number of master (NMAS) and detail (NDET) records to be generated. Master records are numbered consecutively. Detail record numbers jump by an increment equal to the ratio between NMAS and NDET. For each matching master and detail record, the compute kernel is repeated NREP times. NMAS = 0 specifies a compute-only pass, using NREP number of kernel repetitions, which bypasses all tape operations. Any number of passes can be specified by successive parameter cards. The results produced with the sample cards on a particular system are also shown.

Several variations of the program are possible by making changes and recompiling. In the second DECLARE statement, specifying the variables START, SUM, and TABLE, the attributes BINARY FIXED (31) may be replaced by DECIMAL FIXED (7) to execute the kernel in fixed-point decimal arithmetic, or by DECIMAL FLOAT (7) to obtain a comparison with floating-point arithmetic. Increasing N will rapidly increase the main storage requirement N^3 for table storage, as well as lengthen the kernel time; the dimension of the array TABLE (J) must be increased correspondingly. Increasing the value of START increases the precision required.

PL/1 listing of yardstick program

```
YSTKP: PROCEDURE OPTIONS (MAIN);                                                1
   /* THE FOLLOWING DECLARE STATEMENT IS IMPLEMENTATION-DEPENDENT */            2
   DECLARE PARAMS FILE INPUT,                                                   3
         MASTER FILE RECORD INPUT,   DETIN  FILE RECORD INPUT,                  4
         NEWMAS FILE RECORD OUTPUT,  DETOUT FILE RECORD OUTPUT,                 5
         MASGEN FILE RECORD OUTPUT,  DETGEN FILE RECORD OUTPUT;                 6
   /* THE FOLLOWING STATEMENT DEFINES THE TYPE OF KERNEL ARITHMETIC */          7
   DECLARE (START,SUM,TABLE(1000)) BINARY FIXED(31) STATIC;                     8
   DECLARE ((I,J,K,N,U,CHECK,COUNT,LSUM,NMAS,NMAS1,NDET,NDET1,NREP)             9
         BINARY FIXED(31),CARD CHARACTER(80),TEMPC CHARACTER(6)) STATIC,        10
         (INTKEY PICTURE'(6)9',KRETURN LABEL,                                   11
         (START_TIME,END_TIME) CHARACTER(9)) STATIC,                            12
         1  MASTER_REC ALIGNED STATIC,                                          13
           2  MASTER_KEY CHARACTER(12),                                        14
           2  MASTER_SUM   BINARY FIXED(31),                                   15
           2  MASTER_CHECK BINARY FIXED(31),                                   16
           2  MASTER_DATA (15) CHARACTER(12),                                  17
         1  DETAIL_REC ALIGNED STATIC,                                          18
           2  DETAIL_KEY CHARACTER(12),                                        19
           2  DETAIL_SUM   BINARY FIXED(31),                                   20
           2  DETAIL_CHECK BINARY FIXED(31),                                   21
           2  DETAIL_DATA (15) CHARACTER(12);                                  22
                                                                               23
   OPEN FILE (PARAMS);  ON ENDFILE(PARAMS) GO TO EOF;                           24
   N = 10;  START = 100;  NMAS = 0;  NDET = 0;                                  25
   DO J = 1 TO N**3;  TABLE(J) = START + J - 1;  END;                           26
                                                                               27
START_PASS:                                                                    28
   GET FILE(PARAMS) EDIT (CARD) (A(80));                                        29
   PUT EDIT (CARD) (SKIP(2),A(80));                                             30
   IF SUBSTR(CARD,1,6) ¬= ' PASS ' THEN GO TO START_PASS;                       31
   GET STRING (CARD) EDIT (NMAS1,NDET1,NREP) (X(6),3(X(6),F(6)));              32
   IF NMAS1 < 0 THEN GO TO START_PASS;  COUNT = 0;  CHECK = 0;                  33
                                                                               34
/* MASTER GENERATION */                                                        35
   IF NMAS1 = NMAS | NMAS1 ¬> 0 THEN GO TO DETAIL_GENERATION;                   36
   NMAS = NMAS1;                                                               37
   OPEN FILE(MASGEN);  DO J = 1 TO NMAS;                                        38
         MASTER_SUM = 0; INTKEY = J; MASTER_KEY = '000000' || INTKEY;          39
         CHECK = CHECK + J;  MASTER_CHECK = CHECK;                             40
         TEMPC = INTKEY;  MASTER_DATA = 'MASTER' || TEMPC;                     41
         WRITE FILE(MASGEN) FROM (MASTER_REC);                                 42
   END;  CLOSE FILE(MASGEN); CHECK = 0;                                         43
                                                                               44
DETAIL_GENERATION:                                                             45
   IF NDET1 = NDET | NDET1 ¬> 0 THEN GO TO TAPE_PASS;                           46
   NDET = NDET1; RATIO = NMAS / NDET;                                          47
   OPEN FILE(DETGEN);  DO J = RATIO TO NMAS BY RATIO;                           48
         DETAIL_SUM = 0; INTKEY = J; DETAIL_KEY = '000000' || INTKEY;          49
         CHECK = CHECK + J;  DETAIL_CHECK = CHECK;                             50
         TEMPC = INTKEY;  DETAIL_DATA = 'DETAIL' || TEMPC;                     51
         WRITE FILE(DETGEN) FROM (DETAIL_REC);                                 52
   END;  CLOSE FILE(DETGEN); CHECK = 0;                                         53
                                                                               54
TAPE_PASS:                                                                     55
   IF NREP = 0 THEN GO TO START_PASS;                                          56
   IF NMAS1 ¬> 0 | NDET1 ¬> 0 THEN GO TO COMPUTE_PASS;                         57
   KRETURN = WRITE_DETAIL;                                                     58
   OPEN FILE(MASTER), FILE(NEWMAS), FILE(DETIN), FILE(DETOUT);                 59
   ON ENDFILE(MASTER) GO TO END_TPASS;                                         60
   ON ENDFILE(DETIN) GO TO RUNOUT;                                             61
   READ FILE(MASTER) INTO (MASTER_REC);                                        62
   READ FILE(DETIN) INTO (DETAIL_REC);                                         63
   START_TIME = TIME;                                                          64
                                                                               65
KEY_TEST:  IF MASTER_KEY < DETAIL_KEY THEN GO TO WRITE_MASTER;                 66
   IF MASTER_KEY > DETAIL_KEY THEN DO;  PUT EDIT                               67
         ('SEQUENCE ERROR HALTED RUN')(SKIP,A); GO TO CLOSE_FILES; END;        68
                                                                               69
KERNEL:                                                                        70
/* KERNEL SUMS  N  INTEGERS FROM A TABLE OF  N**3  CONSECUTIVE INTEGERS        71
   BEGINNING WITH 'START'.  K-TH INTEGER SUMMED IS 'START - 1 + K**3'.         72
   SUM IS CHECKED ALGEBRAICALLY.  KERNEL IS REPEATED  NREP  TIMES. */          73
   DO I = 1 TO NREP; SUM = 0; U = 0; J = 0;                                    74
         DO K = 1 TO N;  J = J + (6*U + 1);                                    75
         SUM = SUM + TABLE(J);  U = U + K;  END;                               76
      LSUM = (N * (N + 1)) / 2;                                                77
      IF START ¬= (SUM - LSUM * LSUM) / N + 1 THEN DO;  PUT EDIT               78
         ('COMPUTE ERROR HALTED PASS')(SKIP,A);  GO TO START_PASS;  END;       79
   END;  GO TO KRETURN; /* KRETURN IS EITHER WRITE_DETAIL OR CRETURN */         80
                                                                               81
WRITE_DETAIL:  MASTER_SUM = SUM; DETAIL_SUM = SUM;                             82
         CHECK = DETAIL_CHECK; COUNT = COUNT + 1;                              83
   WRITE FILE(DETOUT) FROM (DETAIL_REC);                                       84
   READ FILE(DETIN) INTO (DETAIL_REC);                                         85
                                                                               86
WRITE_MASTER:  WRITE FILE(NEWMAS) FROM (MASTER_REC);                           87
   READ FILE(MASTER) INTO (MASTER_REC);  GO TO KEY_TEST;                       88
                                                                               89
RUNOUT:  DETAIL_KEY = HIGH(12);  GO TO WRITE_MASTER;                           90
                                                                               91
END_TPASS:  END_TIME = TIME;                                                   92
   IF CHECK = (COUNT * (COUNT + 1) * RATIO) / 2 THEN GO TO CLOSE_FILES;        93
   PUT EDIT (' CHECKSUM ERROR HALTED PASS')(SKIP,A);                           94
CLOSE_FILES:  CLOSE FILE(MASTER),FILE(NEWMAS),FILE(DETIN),FILE(DETOUT);        95
                                                                               96
```

PL/I listing of yardstick program (cont'd)

```
PRTEND:  GET STRING (END_TIME) EDIT (HRS,MINS,SECS) (2 F(2),F(5,3));          97
   ELAPSED_TIME = HRS*3600 + MINS*60 + SECS;                                  98
   GET STRING (START_TIME) EDIT (HRS,MINS,SECS) (2 F(2),F(5,3));              99
   ELAPSED_TIME = ELAPSED_TIME - HRS*3600 - MINS*60 - SECS;                   100
   PUT EDIT (' ELAPSED TIME = ',ELAPSED_TIME,' SECONDS')                      101
        (SKIP,A,F(7,1),A);                                                    102
   PUT EDIT(' END OF PASS.  REPETITIONS = ',I-1,', SUM = ',SUM,               103
        ', ACTIVE RECORDS = ',COUNT,', CHECK SUM = ',CHECK)                   104
        (SKIP,2(A,F(6),A,F(9)));                                              105
   GO TO START_PASS;                                                          106
                                                                              107
COMPUTE_PASS:  KRETURN = CRETURN;                                             108
   START_TIME = TIME;  GO TO KERNEL;                                          109
CRETURN:  END_TIME = TIME;                                                    110
   GO TO PRTEND;                                                              111
                                                                              112
EOF:  CLOSE FILE(PARAMS);                                                     113
   END YSTKP;                                                                 114
```

Sample data cards

```
          1         2         3         4          CARD
12345678901234567890123456789012345678901234      COLUMN

 PASS  NMAS=000000 NDET=000000 NREP=010000        COMPUTE PASS.
 PASS  NMAS=001000 NDET=000200 NREP=000001        TAPE PASS.
 PASS  NMAS=001000 NDET=000200 NREP=000050        TAPE PASS.
```

Sample result printout

```
PASS  NMAS=000000 NDET=000000 NREP=010000  COMPUTE PASS.
ELAPSED TIME =    21.2 SECONDS
END OF PASS.  REPETITIONS =  10000, SUM = 4015, ACTIVE RECORDS =   0, CHECK SUM =      0

PASS  NMAS=001000 NDET=000200 NREP=000001  TAPE PASS.
ELAPSED TIME =    16.8 SECONDS
END OF PASS.  REPETITIONS =      1, SUM = 4015, ACTIVE RECORDS = 200, CHECK SUM = 100500

PASS  NMAS=001000 NDET=000200 NREP=000050  TAPE PASS.
ELAPSED TIME =    22.7 SECONDS
END OF PASS.  REPETITIONS =     50, SUM = 4015, ACTIVE RECORDS = 200, CHECK SUM = 100500
```