.

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

O. Brewer, J. Dongarra, and D. Sorensen

Tools to Aid in the Analysis of Memory Access Patterns for Fortran Programs

Mathematics and Computer Science Division

Technical Memorandum No. ??

June 1988

Tools to Aid in the Analysis of Memory Access Patterns for Fortran Programs

O. Brewer, J. Dongarra, and D. Sorensen

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439-4844

1. Introduction

The development of efficient algorithms on todays high-performance computers can be a challenging undertaking, with the efficient use of memory being a critical factor. Memory is organized in a hierarchy according to access time. High-performance computers rely on effective management of memory hierarchy when carrying out floating-point computations. This hierarchy takes the form of main memory, cache, local memory, and vector registers. The basic objective of this organization is to attempt to match the imbalance between the fast processing speed of the floating-point units and the slow latency time of main memory. Successful algorithms must effectively utilize the memory hierarchy of the underlying computer architecture on which they are implemented.

Cache memory, local memory, and vector registers are really high-speed buffers Cache Memory Design: An Evolving Art Cache memory is usually controlled by hardware, while local memory and vector registers are controlled by software. The purpose of this hierarchy is to capture those portions of the main memory that are currently in use, and to reduce the time for subsequent accesses. Since these high-speed buffers are often 5 to 10 times faster than main memory, they can substantially reduce the effective memory access time if they can be used. The success of hierarchy is then attributed to locality of reference and reuse of data in a users program.

Thus, in order to improve the performance of algorithms implemented on high-performance computers, we must consider not only the total number of memory references, but also the pattern of memory references Linear Algebra on High-Performance Computers dongarra Sorensen germany The Use of BLAS3 in Linear Algebra on a Parallel Processor with a Heirarchical Memory We would like our algorithms to observe the principle of locality of reference, so that the data can be effectively utilized. Our new tool provides an aid in understanding a program's locality of reference.

We have designed and built two tools that will help in understanding how a specific Fortran program references memory. The first tool, called the Memory Access Pattern Instrumentation program (MAPI), instruments a user's program and, when the instrumented program is run, produces a trace file. The trace file is a detailed ASCII file giving the individual memory references that were made to the one-

---

† Work supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38.

Typeset on November 24, 1988.

and two-dimensional arrays in the program. The second tool, called the Memory Access Pattern Animation (MAPA) program, allows the trace file to be viewed. This program runs on a Sun workstation (running UNIX and SunView) Sun Microsystems

In this paper Section 2 examines the motivation for efficient use of memory hierarchy, Section 3 discusses the goals for the tools, Section 4 presents a detailed description of the tools, and Section 5 shows how to instrument a program using MAPI and how it is tied into the BLAS. Sections 6, 7, and 8 discuss the user interface to the animation part of the tools and give an example of its use. Section 9 states the availability of the tools over *netlib* and Section 10 summarizes our efforts.

2. Motivation

The goal of this work is to assist in formulating correct algorithms for high-performance computers and to aid as much as possible the process of translating an algorithm into an efficient implementation on a specific machine. Over the past five years we have developed approaches in the design of certain numerical algorithms that allow both efficient and portability Linear Algebra on High-Performance Computers dongarra Sorensen germany Our current efforts emphasize three areas: environments for algorithm development, parallel programming methodologies, and advanced algorithm development.

For most computational problems, the design and implementation of an efficient parallel solution are formidable challenges. Since parallel computation is still in its infancy, we often do not understand what algorithms to use, much less how to implement them efficiently on specific architectures. With existing technology, the construction of a parallel program is a laborious, largely manual enterprise that forces the programmer to assume responsibility for determining a suitable mathematical algorithm and translating it into an intricately coordinated set of instructions tuned to a particular parallel machine.

Efficient parallel programs are much more difficult to write than efficient sequential programs, because the behavior of parallel programs is nondeterministic. They are also much less portable, because the structure critically depends on specific architectural features of the underlying hardware (such as the structure of the memory hierarchy). To use parallel machines efficiently in scientific research, we must develop high-level languages and environments for producing efficient parallel solutions to scientific problems.

The key to using a high-performance computer effectively is to avoid unnecessary memory references. In most computers, data flows from memory into and out of registers and from registers into and out of functional units, which perform the given instructions on the data. Algorithm performance can be dominated by the amount of memory traffic rather than by the number of floating-point operations involved. The movement of data between memory and registers can be as costly as arithmetic operations on the data.

This situation provides considerable motivation to restructure existing algorithms and to devise new algorithms that minimize data movement. A number of researchers have demonstrated the effectiveness of block algorithms on a variety of modern computer architectures with vector-processing or parallel-processing capabilities Demmel Prospectus on which potentially high performance can easily be degraded by excessive transfer of data between different levels of memory (vector registers, cache, local memory, main memory, or solid-state disks).

In particular, for computers with memory hierarchy or for true parallel processing computers, it is often preferable to partition the matrix or matrices into blocks and to perform the computation by matrix-matrix operations on the blocks. This approach provides for full reuse of data while the block is held in cache or local memory. It avoids excessive movement of data to and from memory and gives a surface-to-volume effect for the ratio of arithmetic operations to data movement, i.e., $O(n^3)$ arithmetic operations to $O(n^2)$ data movement. In addition, on architectures that provide for parallel processing, parallelism can be exploited in two ways: (1) operations on distinct blocks may be performed in parallel; and (2) within the operations on each block, scalar or vector operations may be performed in parallel.

The performance of these block algorithms depends on the dimensions chosen for the blocks. It is important to select the blocking strategy for each of our target machines, and then develop a mechanism whereby the routines can determine good block dimensions automatically.

Since most memory accesses for data in scientific programs are for matrix elements, which are usually stored in two-dimensional arrays (column-major in FORTRAN), knowing the order of array references is important in determining the amount of memory traffic. To get an idea of how arrays are accessed for a particular implementation of an algorithm and for a particular data set, we could add instructions to our code to output the name of the array and the indices, whenever an array element is accessed. However, the coding would be tedious and error prone, and looking at page after page of indices is a difficult way of visualizing the memory access patterns. Notice the use of the word "visualizing." We would like to take an arbitrary linear algebra program, have its matrices mapped to a graphics screen, and have a matrix element flash on the screen whenever its corresponding array element was accessed in memory. This type would of tool would be beneficial in many ways:

1. It would help show that the implementation of the algorithm is correct, or at least doing what the developer thinks the algorithm should be doing.
2. It would provide insight into the algorithm's behavior.
3. It would enable the programmer to compare the memory access patterns of different algorithms.
4. Being easy to use, it would be used more often than a tedious method such as examining pages of indices.

3. Goals

The MAP tools are intended to provide an ‘‘animated’’ view of the memory activity during execution. Our objective in providing these tools was threefold: (1) We wished to easily play back a previous execution trace over and over again to study how an algorithm uses memory, (2) we would wished to experiment with different memory hierarchy schemes and observe their effects on the program's flow of information; and (3) we wished to use what was available from Sun Microsystems in the way of creating a SunView application.

4. Description of Tools

There are two basic aspects to accomplishing our goals: preprocessor instrumentation and postprocessing display graphics. Our first tool, MAPI, is applied to the user's program before it is executed. This tool instruments the program so that trace information can be produced. MAPA is a postprocessing tool which displays the output of the instrumented program, permitting a user to visualize the output of the instrumented program and study how the program is referencing memory.

We have developed a simple preprocessor (written in C) that analyzes a FORTRAN module and, for each reference to a matrix element, generates a FORTRAN statement that calls a MAPI routine which in turn records the reference to matrix element. In addition, if calls are made to Level 1, 2, or 3 BLAS lawson hanson paper An Extended Set of Fortran Basic Linear Algebra Subprograms paper define A Proposal for a Set of Level 3 Basic Linear Algebra Subprograms MAPI translates those calls into calls to MAPI routines which understand the BLAS operations and record the appropriate array references. The output of this tool is a FORTRAN module that, when compiled and linked with a MAPI library, executes the original code and produces a trace file. This trace file is used as input to MAPA in order to display the memory accesses on the arrays in the FORTRAN code.

By default, the preprocessor looks for references to array A and assumes that all arrays that are parameters in calls to the BLAS subroutines are array A. However, it also has a run-time option to search for up to three different arrays. Thus, it can be directed to look for references to arrays A, B, and C.

An example of how the program is instrumented is as follows. The original code is

```
         DO 30 K = 1,J-1
            DO 20 I = K+1,N
               A(I,J) = A(I,J) + A(I,K)*A(K,J)
   20       CONTINUE
   30    CONTINUE
```

which is transformed into

```
         DO 30 K = 1,J-1
            DO 20 I = K+1,N
               CALL R( 1, I, I, J, J )
               CALL R( 1, I, I, K, K )
               CALL R( 1, K, K, J, J )
               CALL W( 1, I, I, J, J )
               A(I,J) = A(I,J) + A(I,K)*A(K,J)
   20       CONTINUE
   30    CONTINUE
```

Subroutines R and W record access to storage. The calling sequence has the following meaning:

R(<array id>, <start of row>, <end of row>, <start of column>, <end of column>)

W(<array id>, <start of row>, <end of row>, <start of column>, <end of column>)

where <array id> is the number given to reference the array, (<start of row>,<start of column>) is the starting point in the array for the operation, and (<end of row>,<end of column>) is the ending point in the array for the operation.

Each call to subroutine R records the element of the array. In this case, array A has been given the identifier 1, the first argument to subroutines R and W. Arguments 2 and 3 give the range of row accesses, and arguments 4 and 5 give the range of column accesses. Thus ''CALL R(1, I, I, J, J)'' translates to a read of array A for element I, J. In addition to this information, subroutines R and W also time stamp the event.

The subroutines R and W record the information in a trace file. MAPA can then read the information in the trace file and produce a simple animation simulating the memory accesses. Figure 1 displays the output of MAPA for a view of LU decomposition.

Figure 1. MAPA output for LU decompostion

5. MAPI: The Preprocessor

The preprocessor is very simple and makes many assumptions about the FORTRAN code. Most are assumptions about styles that, although syntacticly correct, are not in common usage. The idea here was not to spend a great deal of time producing a complete FORTRAN lexical analyzer to cover every possible statement, but to produce something quick and easy that would recognize common FORTRAN statements and pick out the array references. Others assumptions are so the preprocessor is aware of certain information such as array size and array names. In addition, there two types of array references that the preprocessor does not yet catch. These are valid array references but not as common. They should be added as needed in the future. All of the assumptions and exceptions are listed below.

At the moment, the preprocessor makes the following assumptions about the input FORTRAN file:

1. It is syntactically correct and compiles.

2. Statements do not extend beyond column 72; in fact, nothing should be beyond column 72, such as old dusty decks that used columns 73-80 as ordering information for the cards.

3. A single parameter is not split across lines in a call to a BLAS routine:

```
   CALL SUB (n1, n2, a(i,
  $               j), ..., b
  $     (j,i), ..., nlast).)
```

4. A single term is not split across lines in an assignment statement:

```
   A(I,J) = T*A(J,
  $               I)/(A(J,
  $     I)+B(J,I))).
```

5. Comments are not interwoven with continuation lines:

```
   CALL SUB (n1, n2, ...,
C  comment
  $     ni, nj, ..., nlast)
```

6. There are no variable names like DO30I (i.e DO30I = 1.3 vs DO30I = 1,3) or function names like IF.

7. There is a variable named N in the file being preprocessed which is set to the array size.

8. The designated arrays are global throughout the file. In other words, there should not be two subroutines each with local arrays using the same variable name; the preprocessor would not know the difference. Also, the main driver program should be in a separate file so that array references, such as array initialization, are not listed in the metafile.

9. In calls to subroutine, the sequence "CALL SUB(parameter1,", should be on the same line.

The preprocessor does not yet catch the following references:

1. IF(...A(I,J)...) ... array reference in logical expression of IF.
2. ...z(A(I,J))... array reference used as an array index.

To use the MAPI program one merely types

```
mapi < foo.f > outfoo.f
```

at the UNIX shell level. The MAPI program will take as input a Fortran program and generate a new program on standard output containing the instrumented version. When compiling and loading outfoo.f, the user should reference the mapilib.a file to resolve calls to MAPI routines.

5.1 Calls to the BLAS

Since the BLAS form such an important part of software for linear algebra problems, we have provided an interface for them to our package. During the preprocessing phase, if a call to a Level 1, 2, or 3 BLAS is present, it is replaced by a call to one of our MAPI routines. The replaced routine will record the memory access to be made, as well as the number of floating point operations to be performed, and then call the Level 1, 2, or 3 BLAS originally intended.

For example, a call such as

```
CALL  SGEMV (  . . .  )
```

will be replaced by a call to

```
CALL  MSGEMV (  . . .  ) .
```

The call will be modified by additional parameters to resolve the two-dimensional array references in the call.

The next example shows how calls to the BLAS are translated. The original code looks like

```
*
*          Compute superdiagonal block of U.
*
           CALL STRSM( 'Left', 'Lower', 'No transpose', 'Unit', J - 1,
     $                 JB, A, LDA, A( 1, J ), LDA )
*
*          Update diagonal and subdiagonal blocks.
*
           CALL SGEMM( 'No transpose', 'No transpose', M - J + 1, JB,
     $                 J - 1, -ONE, A( J, 1 ), LDA, A( 1, J ), LDA, ONE,
     $                 A( J, J ), LDA )
```

After the preprocessor executes, it is transformed to

```
*
*          Compute superdiagonal block of U.
*
           CALL MSTRSM( 'Left', 'Lower', 'No transpose', 'Unit', J - 1,
     $                 JB, A, LDA, A( 1, J ), LDA ,
     $                 1, 1, 1, 1, 1, J )
*
*          Update diagonal and subdiagonal blocks.
*
           CALL MSGEMM( 'No transpose', 'No transpose', M - J + 1, JB,
     $                 J - 1, -ONE, A( J, 1 ), LDA, A( 1, J ), LDA, ONE,
     $                 A( J, J ), LDA ,
     $                 1, J, 1, 1, 1, J, 1, J, J )
```

In the instrumentation the name of the subroutine has been changed. Routines MSTRSM and MSGEMM are MAP routines that record the memory references and call the corresponding Level 3 BLAS. The calling sequence has been augmented to add the starting point of each array reference. Since internally the BLAS do not know what part of the original array the calling program has actually passed, we need to supply the starting index to correctly record each array reference. Therefore, in the call to MSTRSM, the last six arguments describe the starting point of the two arrays involved in the operation. The first arguement 1 involves the array A; the next two arguments, 1,1, provide the row and column index for the starting point of the first array; the last three arguments 1,1,J follow the same form. Within subroutine MSTRSM the appropriate calls to R and W are made to record the events, and then the call to the Level 3 BLAS takes place.

5.2 Execution of the instrumentated program

As the instrumentated program executes, it generates a trace file named *memory.<name>*. The trace file is a readable ASCII file which contains an encoded description of how the arrays in the program have been referenced. There are basically three types of trace lines generated: array definition, read access, and write access. For compactness not every element reference generates a trace line. If a call to one of the BLAS has been made, the trace line may contain the information about a row or column access or both. In addition, the events are time stamped, allowing the MAPA program to merge information with other trace files and have the relative order of operations preserved. We also record the amount of floating-point work that has taken place for a given memory reference. The name of the BLAS is recorded, and during playback the name of the BLAS executed will be displayed.

The trace file has the following format:

Matrix definition:
0 <array id> <number of rows> <number of columns>

Read access:
1 <array id> <start of row> <end of row> <start of column> <end of column> <time>

Write access:
2 <array id> <start of row> <end of row> <start of column> <end of column> <time>

Arithmetic operations and BLAS called:
5 <array id> <flops> <BLAS subroutine name>

An example of the trace file output is displayed below:

```
 0  1  40  40
5  1     0 strsm
5  1   125 strsm
1 1   1   5   1   1    0.05000
1 1   1   5   6  10    0.05000
2 1   1   5   6  10    0.05000
1 1   2   5   2   2    0.05000
1 1   2   5   6  10    0.05000
2 1   2   5   6  10    0.05000
1 1   3   5   3   3    0.05000
1 1   3   5   6  10    0.05000
2 1   3   5   6  10    0.05000
1 1   4   5   4   4    0.05000
1 1   4   5   6  10    0.05000
2 1   4   5   6  10    0.06667
```

```
1 1   5   5   5   5    0.06667
1 1   5   5   6  10    0.06667
2 1   5   5   6  10    0.06667
.
.
.
```

6. MAPA: The Control Panel

The MAPA program is written in C, using SunView, and runs on monochrome and color monitors. It displays the memory access patterns of the arrays by mapping the arrays to the graphics screen and highlighting the elements of the arrays when they are accessed.

The graphics window takes up most of the screen. It initially looks for trace files named *memory.<name>*, where *<name>* distinguishes the different trace files. If it does not find any, it prints an informational message to that effect.

The program can display up to four different arrays at one time. The top row displays the read accesses to the arrays, and the bottom displays the writes. The read accesses flash in blue, and the write accesses flash in red on a color monitor. On a monochrome monitor, the accesses flashed in black.

The panel subwindow (see Figure 2) is MAPA's main user control interface and contains several features:

Figure 2. MAPA Control Panel

• Directory: The user can step through various directories to locate the desired trace file. If the cursor is placed over the end of the directory string and the right mouse button is pressed, a menu listing of other directories will appear. To change to one of these directories, the user simply uses the cursor to highlight the directory and releases the right mouse button. One of the directories in the listing will have a check next to it (most probably the ‘‘.’’ directory). Depressing and releasing the left mouse button while the cursor is positioned on the directory string will cause a change to the checked directory. The user should remember, however, that while directory changes are supported to assist in locating trace files, this is fragile feature.

• Memory files: When files exist in the current directory that begin with the letters ‘‘*memory*’’, the first of these will appear in the memory file string. Positioning the cursor on the string and depressing the right mouse button will cause a menu of memory files to appear. Files can be selected from the menu by highlighting the file name and releasing the right mouse button. Depressing and releasing the left mouse button will cause the next file in the menu to be selected.

• Update speed: This slider controls the length of time the memory reference is held on the screen before fading away. The default value of the slider is set at 100% and can be easily changed by clicking the left button in the slider at the desired value (current value is displayed in brackets, See Figure 1).

• Execution speed: This slider controls the speed in which events are processed when ‘‘GO’’ has been chosen. The speed control slider expresses the event display speed as a percentage of the fastest possible speed. The default value of the slider is set at 100% and can be easily changed by clicking the left button in the slider at the desired value (current value is displayed in brackets, See Figure 1).

The ovals in the panel act as buttons. All of the buttons on the panel are activated by clicking the left button within the boundaries of the button. The button will remain gray as long as the action started by the button continues.

- Load: Initialize and reset MAPA for another trace file. Once a trace file has been chosen it must be loaded in before the animation can be started.

- Go: Process events from the trace file consecutively without stopping. The screen and counters in the control panel are updated appropriately. The only way the event process is halted is for you to hit the left mouse button while the mouse cursor is in the control panel or the end of the trace file is reached.

- Stop: Stops the tracing of events once the program realizes the button has been pressed. It sometimes requires a heavy finger. The activity can be restarted by hitting the ‘‘GO’’ button.

- Quit: This will completely exit the MAPA tool, first asking for confirmation.

- FLOP meter: This meter shows graphically the number of floating point operations over time. It needs work!

7. Execution of MAPA

The canvas subwindow occupies the lower two-thirds of the window. Graphics information is displayed here. The canvas is divided into two rows of four squares. The first row displays the load activities, and the second row displays the store activity.

Each of the four columns of squares across the canvas can be used to display an array. When the trace file is started, a load of a matrix element is denoted by a blackening of an area of the block used to represent the array. (With a color monitor, the area will turn dark blue.) As time evolves and if no further reference is made to that specific matrix element, the area will gradually become lighter, until at some time after the original access, it will return to its original color. If a subsequent reference to that element is made, the area representing the element will again become darken. In this way a user can observe the locality of reference the program is able to achieve.

The same situation is true for store operations. As a store is made, the area representing the element affected is darkened (on a color monitor, the area will turn dark red), and after time the area will return to its original color.

If the BLAS have been used, the whole area affected by the operation is changed at once. This results in considerable saving in terms of display time and in the amount of space the trace file occupies.

8. Example

We have been experimenting with three different organizations for the algorithm to factor a matrix in preparation to solving a system of linear equations via Gaussian elimination. Each method performs the same number of floating-point operations; the algorithms differ only in the way in which the data is accessed. The three methods are block jki, block Crout, and block rank update (see Dongarra karp gustavson Demmel prospectus for more details).

When MAPA displays the trace file produced by merging the trace files from the execution of the instrumented versions of the three different programs, we obtain the picture at shown in Figure 3.

Figure 3. Display of Fortran execution
matrix of order 40 blocksize of 5

The following table was generated on a matrix of order 100 and a blocksize 64.

As can be seen in this case, algorithm 1 (block jki) has fewer store operations over all and slightly more load operations. We would expect this algorithm to perform better than algorithm 3 (block rank update) and marginally faster than algorithm 2 (block Crout).

100 x 100 matrix

		LU1	LU2	LU3
Random	LOADS	102530	108515	99965
	STORES	33180	37455	90180
Diag	LOADS	84100	90085	81535
Dominant	STORES	14750	19025	71750

The row marked Diag Dominant reflects the fact that the matrix is diagonally dominant; thus, no pivoting is performed during the factorization resulting in fewer memory references. (For these results, it was assumed that the data would be held in the memory hierarchy once it was fetched for the operation, (i.e., fetched once for each block operation.)

9. Availability of the Tools

The software described in this report is available electronically via *netlib.* To retrieve a copy, one should send electronic mail to netlib@anl-mcs.apra. In the mail message type:

```
send map from anl-tools
```

A UNIX *shar* file will be sent back. To build the parts, one need only sh the mail file (after removing the mail header) into an empty directory and type ‘‘make’’. Two separate directories will be created, one with the MAPI tools and the other with the MAPA tools.

10. Summary

We have discussed a set of tools for the graphical analysis of memory accesses within a Fortran program. These tools allow users to view trace files generated by algorithms run on any computer.

Using such a tool provides insight into potential bottlenecks resulting from memory accesses. While these ideas are still in the formative stages, we believe approaches along these lines will greatly enhance the performance of programs and the underlying algorithm on shared-memory, high-performance computers.