A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors

Jack Dongarra Orlie Brewer
James Arthur Kohl Samuel Fineberg

CRPC-TR90021
1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 9, 185–202 (1990)

A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors*

JACK DONGARRA

Computer Science Department, University of Tennessee, Knoxville, Tennessee 37996-1300; and Mathematical Sciences, Oak Ridge National Laboratory, P.O. Box Y, Building 9207 A, Oak Ridge, Tennessee 37831-8083

ORLIE BREWER

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439-4801

JAMES ARTHUR KOHL

Department of Electrical Engineering, Box 142, Purdue University, West Lafayette, Indiana 47907

AND

SAMUEL FINEBERG

Department of Electrical and Computer Engineering, The University of Iowa, Iowa City, Iowa 52242

This paper discusses a tool that aids in the design, development, and understanding of parallel algorithms for high-performance computers. The tool provides a vehicle for studying memory access patterns, different cache strategies, and the effects of multiprocessors on matrix algorithms in a Fortran setting. Such a tool puts the user in a better position to understand where performance problems may occur and enhances the likelihood of increasing the program's performance before actual execution on a high-performance computer. © 1990 Academic Press, Inc.

1. INTRODUCTION

The emergence of a wide variety of commercially available parallel computers has created a software dilemma. Will it be possible to design general-purpose software that is both efficient and portable across a wide variety of these new parallel computers? Moreover, will it be possible to provide programming environments sophisticated enough to make explicit parallel programming a viable means to exploit the performance of these new machines? For many computational problems, the design, implementation, and understanding of efficient parallel algorithms can be a formidable challenge. Efficient parallel programs are more difficult to write and understand than efficient sequential programs, because the behavior of parallel programs can be nondeterministic. Moreover, they are generally less portable than serial codes, because their structure may depend critically on specific architectural features on the underlying hardware (such as the way in which data sharing, memory access, synchronization, and process creation are handled).

We have implemented a tool—called the Shared-Memory Access Pattern (SHMAP) program—to aid in the development of high-performance algorithms that are portable across a range of high-performance computers. SHMAP is useful in understanding how various algorithms access memory, the effects of multiprocessors sharing data, and the interaction of cache in a more complicated memory hierarchy. This tool provides a graphical display of memory access patterns in algorithms. Such patterns can be important in understanding memory bottlenecks in computationally intensive algorithms.

Section 2 discusses some of the motivation behind the SHMAP tool. Section 3 describes the preprocessor and postprocessor components of SHMAP in general, and Section 4 describes the actions of SHMAP in particular. Section 5 describes in detail the implementation of SHMAP windows. Section 6 illustrates the parallel-processing capability of SHMAP. Section 7 briefly outlines the tracing functions used to control the tracing operations. Section 8 gives information about how to obtain the new tools. Finally, Section 9 summarizes our efforts in understanding parallel algorithms.

2. MOTIVATION FOR THE SHMAP TOOL

Several factors motivated our developing the tool described here: the benefits of graphical representation, the

* This work was supported by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

0743-7315/90 $3.00

problems raised by memory hierarchy, and the success we had with earlier tool development.

2.1. *Graphical Representation*

In developing an algorithm, the initial definitions and specifications are often done graphically on a blackboard or at a desk with paper. A human can visualize the overall structure of the algorithm far more easily from these graphical representations than from words and formulae. Unfortunately, of course, the computer cannot. These drawings and figures must eventually be translated to a computer language in order to write a computer program. After the program is written, one does not usually go back and view the algorithm in execution in order to understand the flow of the working program and data. Indeed, until recently, such a task would have been impossible.

With a modern workstation environment, however, one can envision going far beyond the notion of numerical output from an algorithm. Today's workstations make it possible to obtain a picture of how the algorithm proceeds and enable the programmer to improve the implementation before it actually runs on high-performance computers.

2.2. *Memory Hierarchy*

The notion of visual aids to programming is certainly not new. Indeed, the entire August 1985 issue of *IEEE Computer* was devoted to this topic, and several of the articles appearing in this issue are germane to this article. A number of efforts are under way to provide parallel programming tools [3, 4, 6, 7, 12].

Our major objective has been to provide a common interface that will allow researchers to exploit existing hardware in the near term. Initially, we developed a tool that traces the flow of execution and processor use within a uniprocessor environment [5]. However, we soon realized that for shared-memory systems involving memory hierarchy, more detail was required.

2.3. *Memory Hierarchy Issues*

On modern high-performance computers, memory is organized in a hierarchy according to access time. 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. In order to be successful, algorithms must effectively utilize the memory hierarchy of the underlying computer architecture on which they are implemented.

The key 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. This situation provides considerable motivation to restructure existing algorithms and to devise new algorithms that minimize data movement.

For computers with memory hierarchy or for true parallel-processing computers, it is often preferable to partition the 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 data movement to arithmetic operations, i.e., $O(n^2)$ data movement to $O(n^3)$ arithmetic operations. 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 beneficial to select the blocking strategy for each target machine, 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. We plan to be able 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.

The tool we developed to meet this objective is the program SHMAP, which provides a visualization of the memory access patterns of a parallel program in a multiprocessing, shared-memory environment.

3. FUNCTIONAL DESCRIPTION OF SHMAP

The program SHMAP involves two distinct entites: (1) preprocessor instrumentation, accomplished by the Shared-Memory Access Pattern Instrumentation (SHMAPI) program; and (2) postprocessor display graphics, accomplished by the Shared-Memory Access Pattern Animation (SHMAPA) program.

3.1. *SHMAPI*

The SHMAPI preprocessor analyzes an arbitrary Fortran program and, for each reference to a matrix element, generates a Fortran statement that calls a SHMAPI routine which records the reference to the matrix element. Moreover, since many programs dealing with matrices reference the Basic Linear Algebra Subprograms (BLAS) [8, 9, 11], SHMAPI translates those calls into calls to SHMAPI routines

that understand the BLAS operations and records the appropriate array references. The replaced routine records the memory access to be made, as well as the number of floating-point operations to be performed, and then calls the Level 1, 2, or 3 BLAS originally intended. This approach allows us to reduce the size of the trace file by recording a range of values referenced per trace line, rather than a trace line per matrix element.

The output of SHMAPI is a Fortran module that, when compiled and linked with a SHMAPI library, executes the original code and generates a readable ASCII file that contains an encoded description of how the arrays in the program have been referenced. Three types of trace lines are generated: array definition, read access, and write access. If a call to one of the BLAS has been made, the trace file may contain information about a row or column access or both. The name of the BLAS is recorded, and during playback the name of the BLAS executed will be displayed. We also record the amount of floating-point work that has taken place for a given memory reference.

3.2. *SHMAPA*

Once a trace file is created, it may be used repeatedly in different ways by SHMAPA to visualize the actions of algorithms. In addition to simple trace events, such as loads and stores, SHMAPA also can project access patterns for various parallel events. These parallel events may themselves contain subgroups of several sequential events that, although executable in parallel with other events, must be executed sequentially in order among themselves. Figure 1 displays the output of SHMAPA for a view of LU decomposition.

4. FEATURES OF SHMAPA

SHMAPA can analyze a given algorithm by using several different system configurations, thus providing insight into

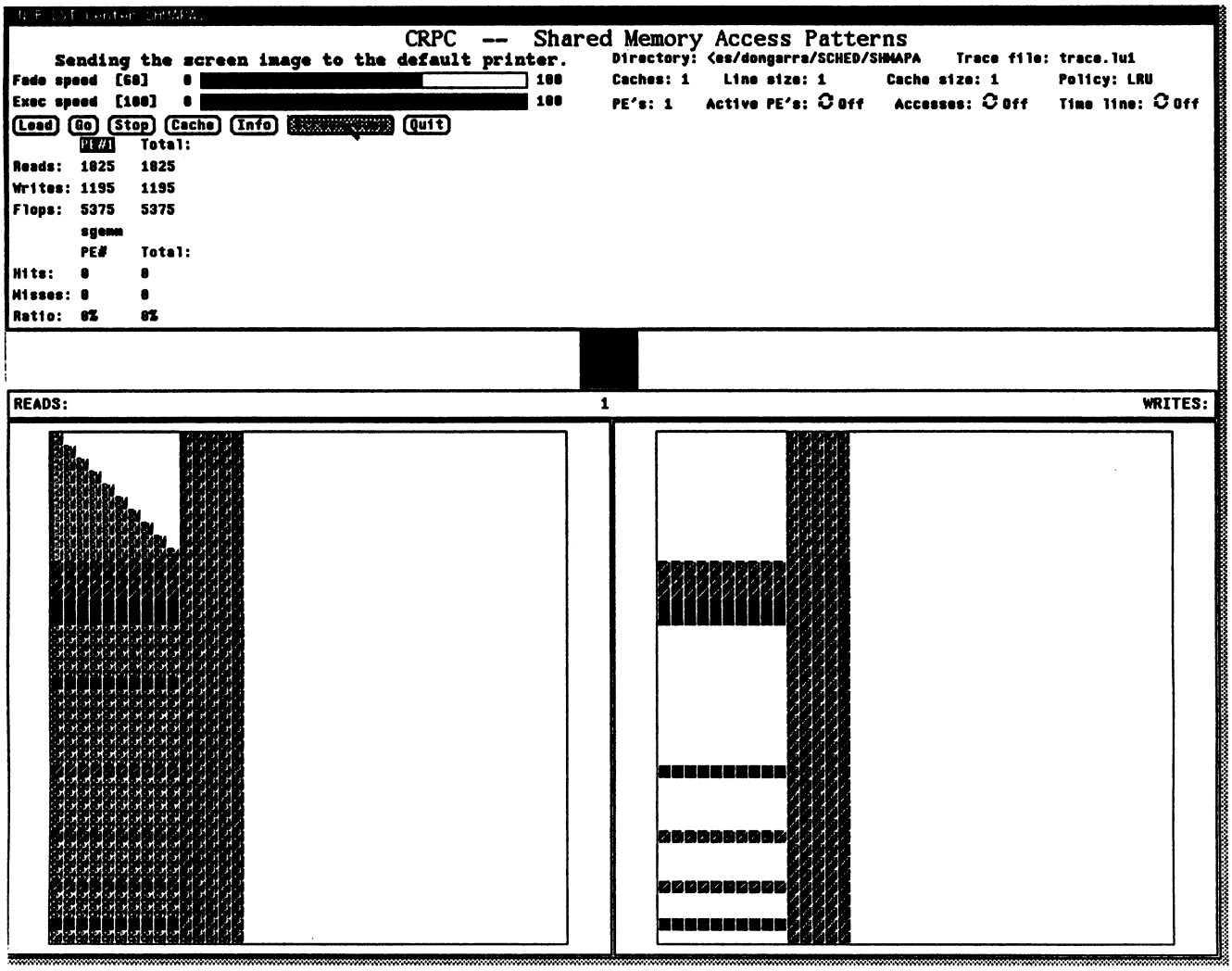

FIG. 1. LU decomposition.

the operation of the algorithm on various systems. The speed of the animation is variable to allow the user to closely examine the more critical aspects of the algorithm, yet move quickly through the less interesting portions. The animation may be adjusted in other ways to emphasize locality of reference and to reveal patterns of memory accesses. In addition, SHMAPA displays a running tabulation of numerical statistics as well as animated histograms. See Fig. 2 for a display of three algorithms for performing LU decomposition. (The graphical representations for Figs. 1 and 2 are explained later in the paper.)

4.1. *Multiple-Processor Configuration*

The current SHMAPA tool models shared-memory parallel machines of up to 16 separate processing elements (PEs). These PEs share one main memory of unconstrained size and up to 16 separate cache memories. A cache memory is a high-speed buffer inserted between the processors and the main memory to capture those portions of the contents of main memory that are currently in use. Since cache memories are typically five to ten times faster than main memory, they can reduce the effective memory access time if carefully designed and implemented [2]. In the current tool, only one layer of cache is supported between main memory and the PEs. Hence, a PE may have only a single cache between it and main memory. Although each PE is assigned to at most one cache, several PEs may share a single cache. For consistency, each PE is assigned a specific cache to use when the cache is turned on.

The purpose of the tool is to display the patterns of accesses for algorithms. Thus the actual value of the data in memory is inconsequential; the only characteristic necessary for analysis is the location in memory. Hence, the PEs are not actually modeled but merely represent the origin of load and store events in a trace. A given PE may load or store a memory location, but the data themselves are not taken into consideration.

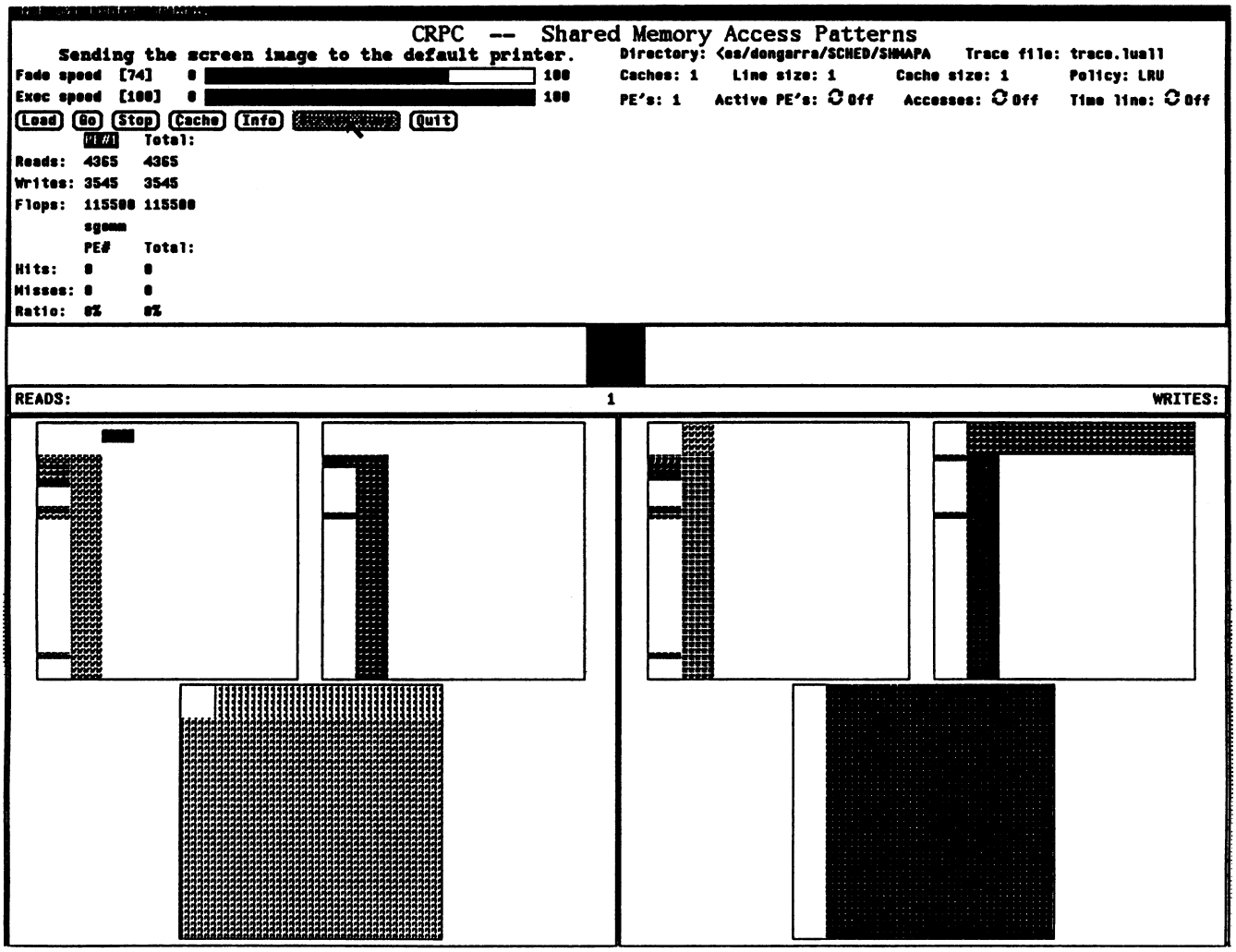

FIG. 2. Three algorithms performing LU decomposition.

Each PE has a number and a distinguishable color to identify it when a memory access is made. The color is also used in identifying a given PEs actions in the animated histograms. It should be noted, however, that the PEs as well as the caches are all considered to be identical, and are distinguished only for consistency. A window is used to display the various sets of PE colors and their corresponding numberings. The colors vary depending on the number of PEs configured for the system (see Fig. 3).

The number of PEs and the number of caches are set with separate pull-down menus. The number of PEs must be a power of 2, but the number of caches needs only to be an integer less than or equal to the number of PEs. Upon changing either of these parameters, the cache-to-PE assignments are recalculated, and the system is automatically reset and ready to continue where tracing was interrupted. Note that the tracing must be temporarily stopped in order to change the system configuration in any way.

4.2. *Cache Configuration*

As previously stated, all caches for any given system configuration are identical. The characteristics of the caches are quite flexible, however. A number of parameters can be modified via pull-down menus, including cache size, line size within the cache, and cache replacement policy. Cache can also be turned off, so that the memory accesses proceed directly to and from main memory without the intermediate cache layer.

Cache size is defined here as the number of words of memory available within a given cache. The cache size can be varied from 1 word to 65,536 words in powers of 2.

Cache line size is the smallest number of words of memory that may be loaded from or stored to main memory at a given time. The relation of this parameter to cache size affects the efficiency of the cache [2]. The cache line size can also be varied from 1 to 65,536 words by powers of 2, provided that the line size selected is less than the cache size.

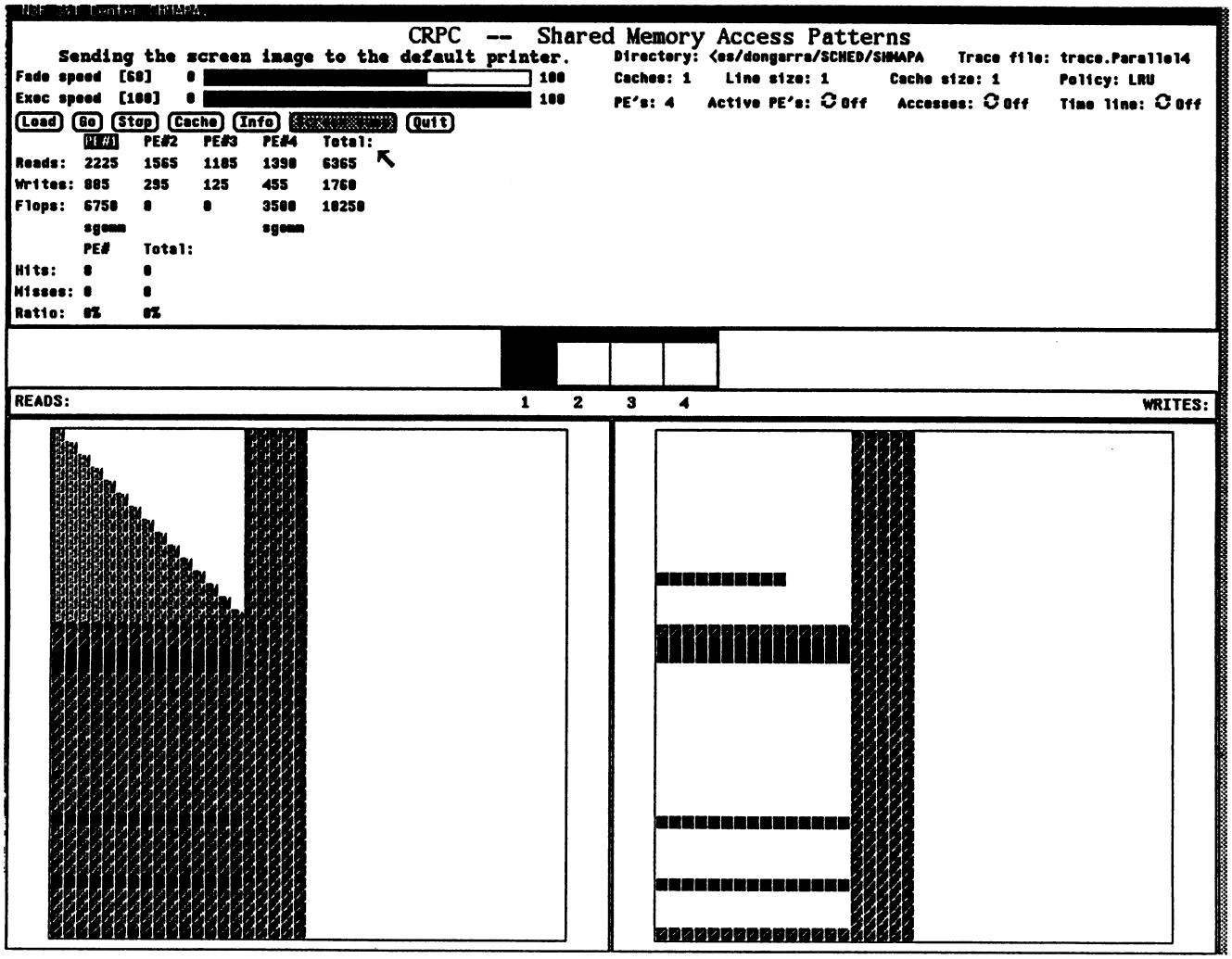

FIG. 3. Multiple processors.

For simplicity, each cache in SHMAPA is initialized with a *cache replacement policy* which consists of placing new cache lines into the next free sequential location. When a cache miss occurs (in other words, when a desired cache line is not available within the cache), the new cache line is loaded into the next unused position in the cache. In this way the cache fills up sequentially starting from the first position and continues until all positions are occupied. When the cache becomes full, the selected cache replacement policy takes effect, and normal cache operation commences.

The cache replacement policies currently supported by the tool include Least Recently Used (LRU), Least Frequently Used (LFU), First-In First-Out (FIFO), Clock, Last-In First-Out (LIFO), and Random. The LFU mechanism chooses the line that has been used the fewest number of times since it was loaded into the cache. The FIFO mechanism simply keeps track of the order in which lines were loaded into cache and replaces lines in the same order. The Clock mechanism is an approximation to LRU using FIFO, but as lines are used, a usage bit is set so the line will be skipped over as the FIFO queue is traversed. The LIFO mechanism is similar to the FIFO except that instead of replacing the lines in the order they were loaded, the lines are replaced in reverse order. The Random mechanism simply generates a random line number and replaces that line. These are the more commonly used cache replacement policies, but SHMAPA has provisions for installation of a wide variety of other policies.

4.3. *Memory Access Animation*

SHMAPA displays the activity in memory as a result of the actions of an algorithm. Specifically, SHMAPA graphically represents loads and stores dynamically over time. Although memory can logically be considered one-dimensional, SHMAPA displays memory as being two-dimensional. Among other reasons, the visual space available on graphics devices encourages use of two dimensions. In addition, the algorithms of interest deal with two-dimensional matrices.

To simplify the image displayed to the user, two separate windows or canvases are used to visualize the single shared main memory. On one canvas, all loads from main memory are shown; on the other canvas, all stores to main memory are shown. To further clarify the meaning of the display, the main memory canvases are sectioned into separate rectangular regions that represent the different matrices. These regions are scaled and arranged to fit into a given canvas. Each of the "loads" and "stores" canvases displays the same arrangement of matrix regions to provide consistent observation of the activity in each matrix.

Within a given matrix region, an actual location or word of memory is represented by a small square area. All squares in all matrix regions in both main memory canvases are uniform in size. These squares are arranged in the two-dimensional regions in rows and columns corresponding to those of the given matrices. Chosen arbitrarily yet appropriate to Fortran column-major convention, the rows of a matrix are displayed from the top of a region to the bottom, and columns from left to right.

An access to a particular location in memory, or matrix element, is represented by the illumination of the corresponding square on one of the canvases. This illumination will occur appropriately on either the load canvas or the store canvas, in the appropriate matrix region, and at the row and column location of the matrix element accessed, and will be drawn in the color of the PE that made the access.

These illuminations occur in succession over time to provide an animation of the memory accesses to the various matrices. The time stamps from the trace file provide an ordering of memory accesses only; hence, the animation reflects memory accesses only, with no breaks for computation. The "exec speed" determines the degree to which the animation is paused between successive trace events. There can be no pause at all, or a pause of up to a few seconds between events.

To avoid having the canvas become a confusing blur of color even at slow execution speeds, it is necessary to do more than simply color elements on and off as accessed. To provide a more fluid and continuous animation, each memory access is gradually faded with time after its initial illumination. The initial color fades through a number of small discrete jumps which gradually approach black. Eventually, the memory access will reach black and then default back to the canvas background color. This approach provides the user with enough visual information to see when things have not been referenced in a while.

Each individual memory access is faded separately in relation to the amount of time that has expired since it was illuminated. Note that "time" here refers to trace time, which is examined at each trace event time stamp to determine whether a given access should be faded. The amount of time between fade steps determines the fade speed. By reducing the fade speed, accesses will be visible long enough to be compared with subsequent accesses, thus providing more understanding of the locality of reference. Patterns of accesses over large time periods can all be seen simultaneously and their ages identified by color brightness. By adjusting the fade speed properly, the memory access patterns will show various characteristics of the given algorithm.

Theoretically, some accesses might be hidden. For example, if a PE were to access the same location a second time before previous accesses (by itself or by a different PE) had faded, the access could go undetected. To avoid this situation, if the color of the accessed location is still fully illuminated from a previous access and has not yet been faded, accesses are flashed to black before illuminating.

4.4. *Cache Animation*

The cache window is animated in the same way as main memory, with similar fading of accesses over time. Cache lines are displayed vertically from top to bottom as sets of square elements. The cache canvases are sized to fit an integer number of cache lines vertically, sometimes resulting in leftover space in the last row. The cache canvases are also shaped to have the overall cache window result in an aesthetically proportioned rectangle.

SHMAPA uses a main memory update policy called write-through-with-no-write-allocate (WTNWA). This policy requires no manipulation of the cache on a store. Rather, the updated value is stored directly to the main memory location independent of whether the affected cache line is loaded into cache. Thus, there is only one canvas per cache memory, since only loads are displayed. On a load, the accessed element is illuminated individually except in the case of a miss, where the entire cache line is illuminated (as well as the corresponding locations on the main memory load canvas). On a store, the cache canvas is not illuminated; only the proper location on the main memory store canvas is colored (see Fig. 4).

4.5. *Statistics Displayed*

In addition to the animated memory access canvases, three other canvases animate several of the statistics gathered while tracing. The remaining statistics are displayed numerically on the main panel.

The first of the three animated canvases is a sliding graph which illustrates the number of actively executing PEs at any given time. Since work may not always be available in the parallel pool, some PEs will remain inactive for periods of time. The graph outlines a vertical histogram that sums the number of active PEs. The graph slides with time pro-

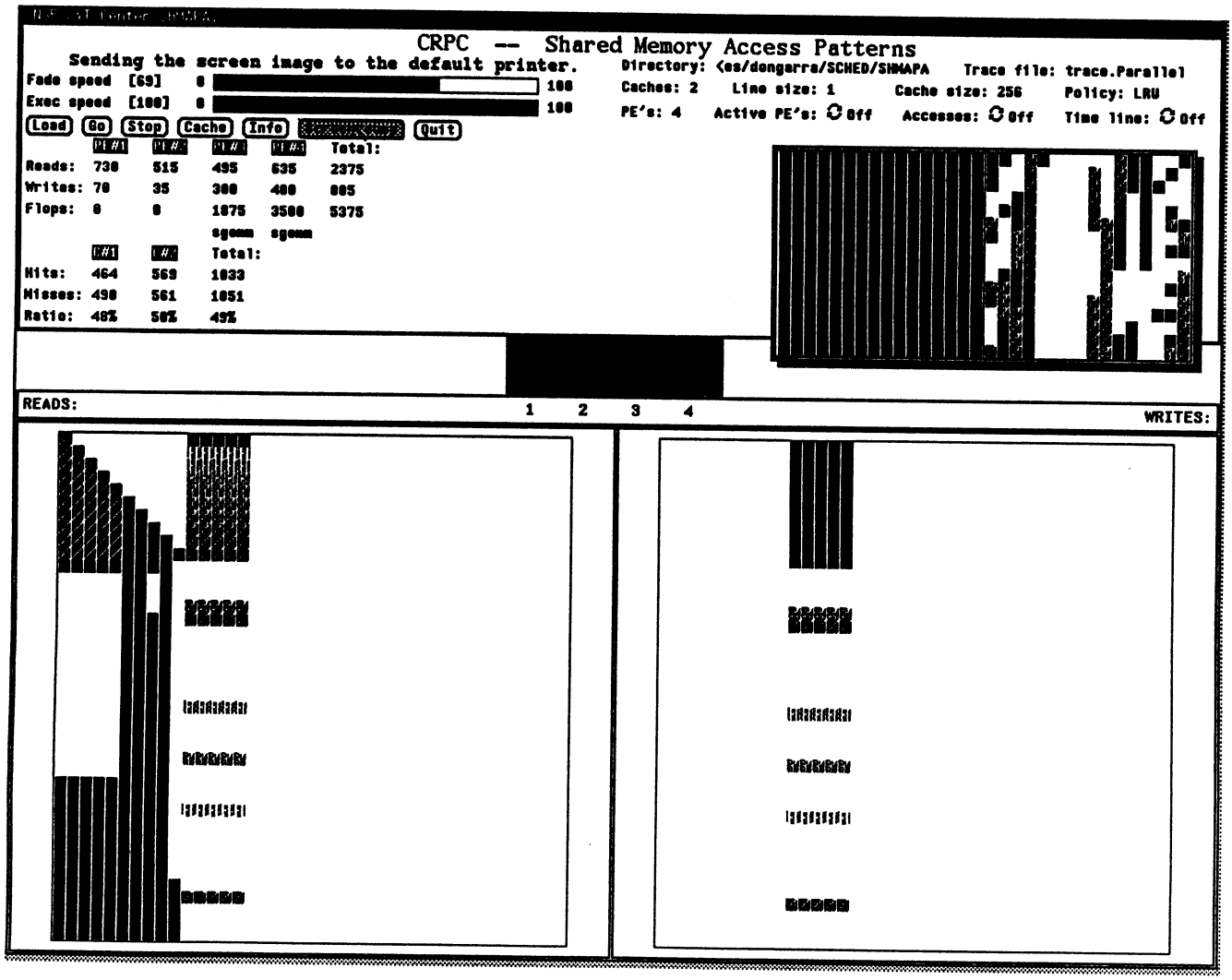

FIG. 4. Cache animation.

portional to the time elapsed according to the trace event time stamps. Hence, this time reflects actual execution time including calculation, as opposed to the memory canvas animation which does not. The graph is appropriately drawn with the colors of those PEs that are active (see Fig. 5, the canvas labeled "Number of active PE's").

The second animated canvas displays the total number of accesses made to each memory location or matrix element in the different matrices. This canvas is a horizontal histogram that builds from left to right as the elements are accessed. These matrix elements are arranged sequentially from top to bottom in a one-dimensional sequence. Arbitrarily, this sequence is generated by concatenating one row after another in the matrices. Each access added to a given location is shown as a colored dot stacked to the right of the previous access dots. The color of each dot reflects which PE made each reference to that particular location (see Fig. 6, the canvas labeled "Number of references").

The third and final animated canvas shows a time-line history of accesses similar to the access histogram. Unlike the access histogram which is stacked tightly against the left edge, the time-line histogram is spread over the width of the canvas. Each access is marked in a similar fashion as on the access histogram, with each dot's color representing the PE making the access, but on the time-line histogram the horizontal positioning has meaning. The entire width of the canvas represents the entire length of time of execution of the given trace. Each dot is placed between the left and right edges of the canvas proportionate to the relation of its time stamp to the total execution time (see Fig. 6, the canvas labeled "Time").

All of these animated canvases are turned on and off with cycle switches on the main panel. To save system overhead, the animated canvases are completely inactive when not displayed. They are initialized when turned on, and only then are statistics gathered and displayed.

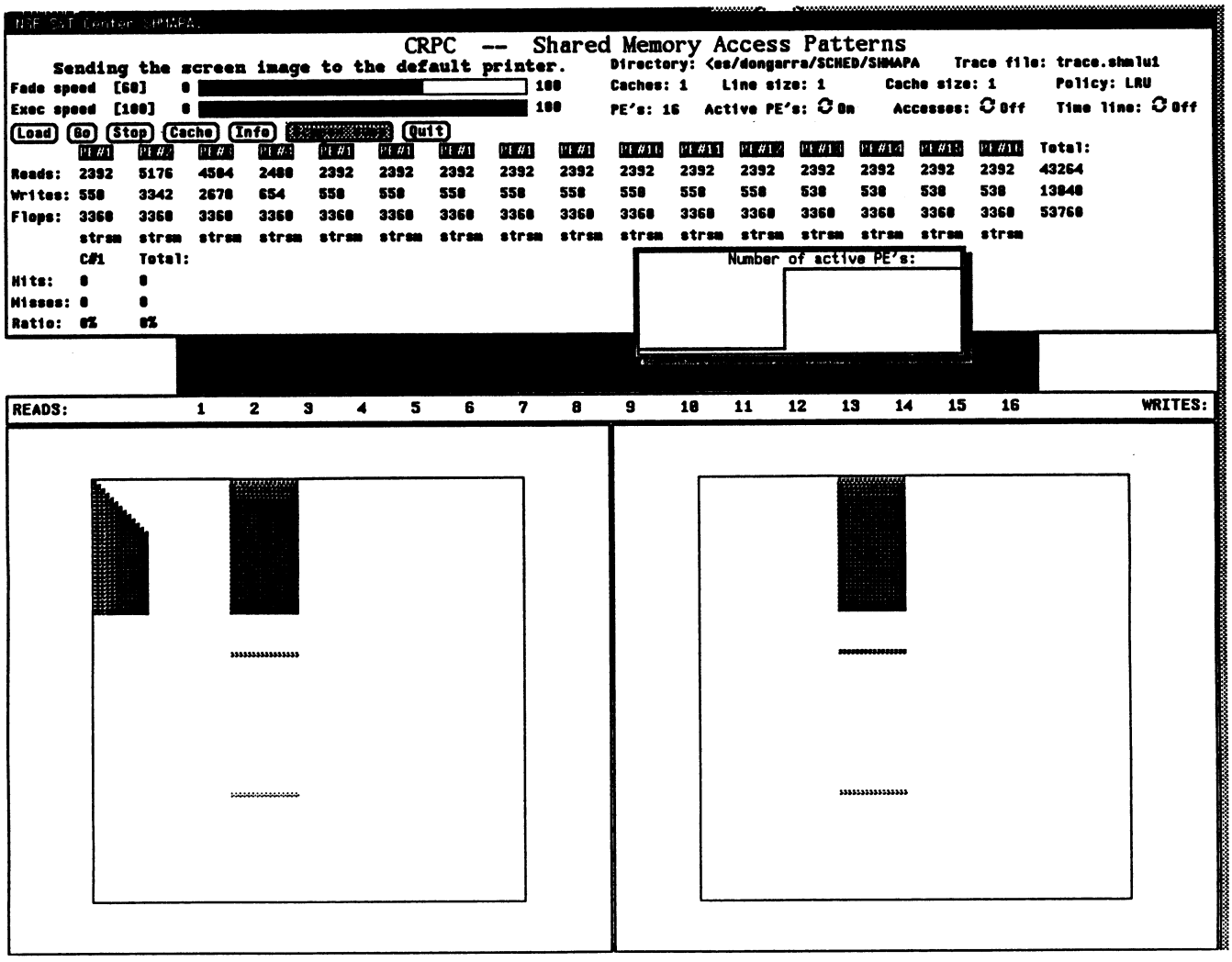

FIG. 5. Multiple processors.

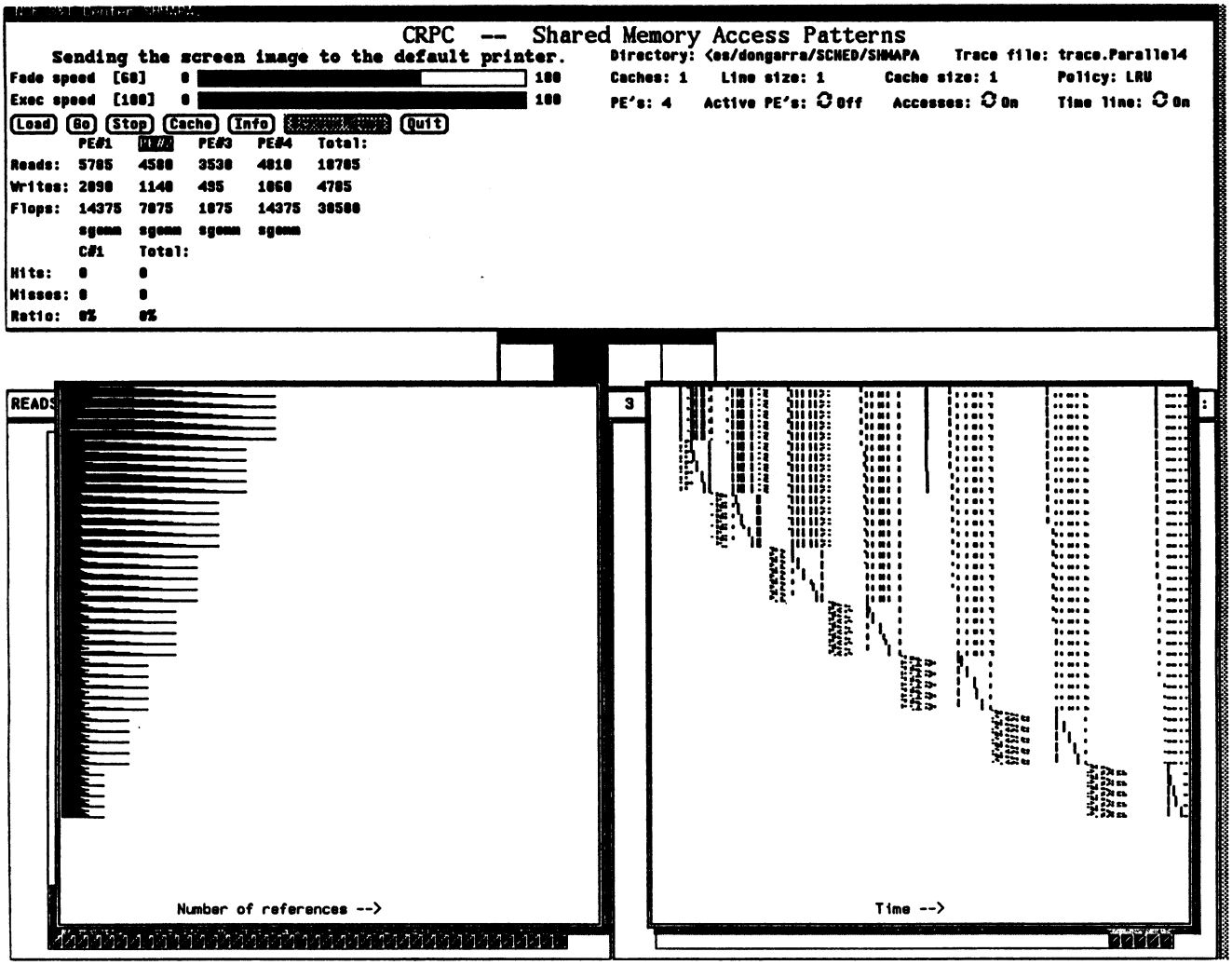

FIG. 6. References over time.

In contrast, across the bottom of the main panel, various statistics for the PEs and caches are always displayed. Running totals are kept for each statistic and displayed dynamically during tracing. Overall running totals for each statistic across all PEs or all caches are also displayed dynamically. The displays are automatically adjusted to display statistics for the proper number of PEs and caches.

The information displayed for the PEs consists of a tabulation of loads, stores, and floating-point operations (*flops*) executed during tracing. The flops are determined by special trace events read in from the trace files during tracing.

The information displayed for the caches consists of a tabulation of cache hits, cache misses, and the cache hit ratio. Cache hits reflect the number of load accesses that were satisfied through cache with previously loaded locations. Cache misses reflect the number of load accesses in which a cache line had to be loaded into the cache from main memory to satisfy the access. The cache hit ratio is the percentage of all accesses that resulted in cache hits. This ratio describes the efficiency of the cache in satisfying the memory accesses.

5. IMPLEMENTATION AND PORTABILITY OF SHMAPA

The Shared Memory Access Patterns tool was engineered in C language [1] through the use of the SunView windowing system. The tool consists of a number of windows, panels, and canvases that together animate the memory accesses as described in the previous section. This section will explain the operation of the more vital and unusual software for SHMAPA. The trace file format, as well as the method of efficiently processing the trace files, will be discussed. The matrix region mapping, cache mechanism and window mapping, the color maps, and the memory access flashing and fading will also be covered.

5.1. *Trace File Format*

As previously stated, the trace files are actually files containing output dumped from modifier user applications. The trace data consist mostly of numerical and some character information. These data represent various types of trace events which are subsequently interpreted by SHMAPA. An event begins with an integer event type which determines the syntax of the remainder of the event. All data are separated by white space, and new lines are not interpreted. The currently defined event types include matrix definition, load, store, information message, flops count and subroutine name, parallel start, parallel end, sequential group start, and sequential group end.

The matrix definition event defines a new matrix to be referenced by SHMAPA. The arguments to this event include a matrix ID and the number of rows and number of columns in the matrix. The matrix ID is used by other events to reference the given matrix. The ID is a unique integer from 1 to the maximum allowable number of matrices, which is a predefined constant. The number of rows and number of columns are integers greater than 0.

The load and store events represent memory access requests for particular sets of matrix elements. Both events have as arguments a matrix ID, a starting row and column, an ending row and column, and a time stamp. The matrix ID must be in the valid range and for a matrix that has been defined. Also the rows and columns referenced must exist within the matrix size specified by the given matrix definition. The time stamp is a floating-point value representing the program execution time elapsed when the event was logged by the user application.

The information message event allows the user to specify some comment about a particular matrix or operation occurring in the application. The arguments for this event are a matrix ID and a character string, including white space, concluded by a new line. The matrix ID must be for a valid, defined matrix. The information message is displayed as a special text item on the main panel during tracing.

The flops and subroutine name event provides information concerning the number of flops that have occurred during the previous memory accesses, as well as the name of the subroutine currently executing. The arguments to this event are a matrix ID, an integer representing the flops count, and a character string subroutine name. The matrix ID must be for a valid, defined matrix. The flops count is added to the running total kept for the particular PE executing this event, and a special text item for that PE displays the subroutine name. The subroutine name can contain no white space.

The last four events (the parallel start, parallel end, group start, and group end events) are related and are used only to control the combining or separation of events as sequential or parallel groups. These events have no real arguments, except for character string labels that are discarded. These labels serve only to clarify the trace for human observers.

5.2. *Trace File Processing*

Before a given trace file can be "loaded" for tracing, certain statistics about the file must be obtained. When the LOAD button is clicked, the interactively selected file is actually preprocessed once completely before positioning at the start for tracing. All the events for the entire trace are loaded in, and information is saved about all matrices, the length of time of execution, and the minimum time between event time stamps.

All matrix definitions are processed in this initial step. In this way, all information is known from the beginning about how many matrices will be used, as well as their sizes. This information will be used to map all matrix regions into the available space on the main memory canvas in one complete step. It is not then necessary to repeat the time-consuming operation of rearranging the matrix regions because another matrix is encountered.

By monitoring the time stamps of the trace events as they are loaded in, the total trace execution time and the minimum time between time stamps can be determined. Knowledge of the total execution time makes possible the time-line histogram. This histogram displays accesses proportionate to the relation of elapsed time to total trace execution time. The minimum time between time stamps provides a granularity number for use with the active PEs sliding graph. The graph uses this minimum time to scale the distance that the graph slides with the passage of time, as measured by the trace event time stamps.

Once the preprocessing step has completed, the matrix region mapping operation is done, and all displays are reset and initialized. At this time, the beginning of the trace file is loaded in and converted to an internal representation. There are several reasons for converting to this internal representation.

The main reason is performance, considered first as throughput. In an animation tool, it is best to have fast execution to provide fluid graphics. After converting to internal representation in one intensive I/OP operation, all subsequent references to the trace information can be made from memory instead of secondary storage, thus increasing the overall speed of processing the trace events. Rather than repeatedly waiting for I/O devices, information can simply be fetched immediately from memory. For the same reason, response time performance is also increased. Since user interface interrupts do have to wait for I/O operations to finish aside from the initial loading period, response time will be decreased.

Another reason for loading the data into memory is flexibility. The trace information in memory is randomly accessible, as opposed to the trace file which provides sequential

access only. Hence, having trace events in memory allows jumping and reverse tracing. Although neither of these options has been implemented, the capability exists and is valuable.

Of course, considering for a moment the potential size of trace files, it becomes immediately obvious that it is infeasible to load the entire file. The time taken to load a large trace would be substantial and annoying. The memory required to store all of the trace event information would also be quite extensive, even if stored in a condensed format. These drawbacks actually lead to a more desirable solution, however.

By loading small portions of the trace file at a time, only a few thousand trace events, the benefits are still gained without an unacceptable amount of time or memory. In fact, we have observed in testing the tool that when sets of only a few thousand events are used at a time, the time delay required to load and convert the trace data is nearly imperceptible. Rather, the break in animation to load the next portion of the trace data is lost in the midst of the graphic activity on the screen.

5.3. *Internal Trace Information Representation*

The trace information is converted internally into a hierarchal structure of the work to be done. On the top level of this hierarchy is the designation of *pool* entry. Pool entries consist of one or more *task* entries, which are entries of the next level down in the hierarchy. These task entries consist of one or more entries of the lowest level, which are appropriately called *trace* entries since they store the original trace data. The purpose of this hierarchy is to provide a clear abstraction of the work and to expedite division of work between multiple PEs.

The relation of the trace, task, and pool levels is stored in arrays of appropriate trace, task, and pool structure entries. The lowest trace level array is actually a sequential list of the trace events, except for the parallel/group starts/ends, in the order read from the trace file. The trace entry structure holds data only, with no pool hierarchy interconnection information. This structure includes an event type, or opcode, and a matrix ID. The structure also has elements that can be used to store the various different types of data needed by trace events of all types. Several integer values, a floating-point value for time stamps, and a character buffer for strings and messages are all included.

The task and pool structures contain all the information necessary for maintaining the pool hierarchy interconnections. This information is derived from the parallel/group start/end trace events. The task and pool structures contain similar information, each structure referencing the structures in the level below it.

The task structure contains a type that identifies a particular task as consisting of a *single* trace event or a *group* of trace events. The structure also contains two index markers which point to the specific trace array elements contained in the task. These markers are simply integers that store the trace structure array indices of the first and last trace array elements of the task. The task structure also stores one more marker to keep track of which trace event is currently awaiting processing in the task.

The arrangement of trace events into tasks is generated from the group start and end trace events. If a trace event occurs by itself in the trace file and not between group start and end events, it is considered a single task. In this case, the task consists of just the one trace event. Otherwise, all trace events that occur between a particular group start event and its matching group end event will be considered members of a group task.

The pool structure is identical to the task structure, but represents a piece of work rather than a task. The piece of work, in fact, contains tasks. The pool structure contains a type value, begin and end pointers, and a current task marker. Here the type may be either *parallel* or *sequential* to designate whether the particular piece of work in the pool can be processed by many or only one PE, respectively. The begin and end pointers reference into the task structure array to identify which specific tasks are a part of the piece of work. The marker points to the task currently awaiting assignment.

The arrangement of task events into pieces of work is generated from the parallel start and end trace events. If a task is generated from any number of trace events that exist independently outside a parallel start and end events in the trace file, then that one task constitutes a single sequential piece of work in the pool. Otherwise, any number of tasks generated from trace events that occur between a parallel start event and a parallel end event will be combined into one parallel piece of work.

The method of depleting the pool of work amounts to processing pieces of work one by one in order. Each piece of work is processed by assigning the tasks making up the piece of work to PEs, which then execute the individual trace events contained in those tasks. The tasks making up a particular piece of work may be executed by several different PEs, but all trace events in a given task, single or group, must be processed by a single PE. Hence, only one PE may process a sequential piece of work, which only contains one task.

From the PE perspective, a given PE requests work and can then receive a task containing one or more trace events. The PE will execute these trace events in order until the task is completed. When the task completes, the PE will request another task from the current piece of work in the pool. Meanwhile, other PEs are requesting tasks from the current piece of work. All PEs with assigned tasks execute their trace events in parallel. When there are no more tasks to assign, the unassigned PEs wait for the PEs with task assignments

remaining to finish the execution of their tasks. Only when all tasks within a given piece of work are completed can the PEs receive tasks from the next piece of work. Hence, the completion of a piece of work can be considered a type of synchronization point, where all PEs must wait until all are ready to proceed.

To keep track of which tasks and trace events have been processed, the markers in the pool and task structures are used. Receipt of a task will increment the pool structure task marker to the next available task. A PE increments the task structure trace marker through the trace events as it proceeds.

With the hierarchy defined in this way, all valid trace files can be broken into pieces of work which are easily processed by SHMAPA. Parallel and sequential work can be assigned to any number of PEs requesting work while preserving algorithm function and efficiently exploiting the parallelism present.

5.4. *Cache Mechanism*

The cache mechanism for SHMAPA is internally represented by several data structures that store the current cache states for each cache. An indexing structure provides quick access to the structure which stimulates mapping of the cache locations to main memory. The implementations of cache replacement policies have been specially designed for flexibility so that many diverse policies may be added easily.

The main data structure in the cache mechanism maintains the mapping information between cache and main memory. This array of cache elements contains references to the particular matrix ID and the address of the matrix element's data which it stores. Each cache element also holds an index to the first cache element in each cache line to expedite handling of cache misses.

To reduce searching time, an indexing array is arranged to easily determine whether a desired matrix element is in cache. Each index entry references the main cache structure and points to the particular cache element that stores the desired matrix element. If the matrix element is not in cache, the index will point nowhere.

The cache replacement policies that determine which cache line is replaced on a cache miss are carefully implemented to allow flexibility and easy addition of more policies. Each cache line is provided with a status word for use in identifying the desired replacement characteristic. When a cache line is referenced, its status is updated as specified by the particular policy chosen. Also, when a miss occurs and a cache line must be replaced, the replacement is done under the control of that policy.

Two distinct methods are used. Many of the standard policies implemented by SHMAPA share similar characteristics and can be implemented by general routines that operate slightly differently depending on the precise policy in effect. For these policies, the status updating is handled in such a way as to allow replacement choices to be made identically for all such policies. For example, LRU and LFU policies may share the same replacement choice routine. The status update routine places a time stamp in the status words for LRU and stores total references for LFU. Now, a single routine that searches for the status word of smallest magnitude can be used to choose the next cache line to be replaced for both policies.

The other possible method requires separate routines for each policy since they perform unusual and unique operations for cache replacement. To combine these two methods, global high-level status update and replacement choice routines are created. These master routines pair each policy specifically with its appropriate routines, and then calls these particular routines depending on the currently active policy. Policies that share the same routines will all be paired with the same set of status update and replacement choice routines and can take advantage of this overlap. Meanwhile, other more unusual policies will use their special unique routines. This method is implemented through simple conditional blocks that choose which routines handle the status and replacement manipulation for the current policy. It is trivial to add a new policy by including another case in the conditional block that references the new policy's routines.

5.5. *Matrix Region Mapping*

During the preprocessing step in loading a trace file, all information regarding matrices is obtained. With this information, regions can be defined on the main memory canvases for displaying accesses to the given matrices. The assignment of a matrix to a canvas region is consistent for each of the of the two main memory canvases. In assigning these regions, efficient use of the display space is the main concern. The goal is to scale and arrange the regions in such a way as to maximize the size of an individual matrix element while minimizing vacant space between regions.

To avoid confusion and to simplify the mapping problem, the matrix regions are arranged in order by their matrix IDs, as declared by the user in the trace file definitions. To minimize vacant space, the rectangular regions are arranged side by side with adjacent sides parallel. The regions are placed starting from the upper left corner of the canvas and proceeding horizontally to the right. When the region placement reaches the right edge of the canvas, it wraps around and begins underneath, starting again from the left. The number of regions per row depends on the individual matrix sizes as well as the size of a matrix element square. The size of a matrix element is specified by a *matrix factor* which determines the number of pixels per side in each element square displayed. This matrix factor is consistent across all matrix regions. Setting the matrix factor deter-

mines the number of regions that will fit in a particular row of regions on a canvas. Since the matrix sizes are static for a given trace, and the number of regions per row depends on the matrix factor, this matrix factor is the only independent variable parameter.

Hence, to find the most efficient arrangement given the above constraints and the size of the main memory canvases, the matrix factor is varied. The initial solution tried is with a matrix factor of 1, meaning that each matrix element will be represented by a 1×1 pixel square. This is the smallest possible matrix factor. If the matrix regions cannot be arranged to fit when scaled by this factor, then no possible solution for matrix region arrangement exists, the matrices cannot be mapped, and that particular trace file cannot be used. Otherwise, if a solution is found for the initial factor, then other arrangements are tried with larger factors to search for the most efficient solution. The positive integer factors increase to approach the *best* factor, or the factor that will provide the most efficient solution. The largest possible matrix factor is desired because it scales the matrix regions to the largest dimensions, thus filling more of the available space on the canvases. When a factor is found that scales the matrix regions to a size too large to accommodate a solution, then the previously tried factor is known to be the best possible factor. The matrix region mapping for that factor is then used for the trace.

Once the most efficient mapping is found, the placement of the matrix regions is refined slightly to provide a more aesthetic display. The vacant regions around the matrix regions are adjusted to leave even borders and spaces between matrix regions. This adjustment is done by separating and centering the matrix regions evenly across the canvases both vertically and horizontally.

5.6. *Cache Window Mapping*

Some amount of region mapping is done for the cache display. This problem is simpler than the matrix region mapping for several reasons. Since every cache is of identical characteristics, all individual cache regions should have the same dimensions. The cache regions are displayed in a dynamically created special window, so there are fewer restraints on the overall canvas size. The regions are arranged in only one row, so the mapping actually requires no placement but only size adjustment. The regions need only be scaled, stretched, or compressed to fit into the desired cache window size.

This fitting can be done in a similar way to the matrix regions by gradually increasing a *cache factor* until the cache window no longer fits on the display. The resulting individual cache region dimensions for the given cache factor can be produced by a single calculation. This calculation relates the number of cache elements, the cache line size, and the number of caches to the desired cache window proportions and maximum size. These proportions are derived from the overall proportions of the pair of currently defined SHMAPA main memory canvases. An aesthetic combination results when the cache window is placed over the main SHMAPA window.

5.7. *Color Map Manipulation*

The colors used by SHMAPA to distinguish the actions of different PEs are the result of extensive color map manipulation. In fact, one of the more complex elements of the SHMAPA software involves the generation of the color map. A color map is a data structure containing the list of colors available for use at any given time on a color display. This structure is an array where each entry stores information concerning a given color's characteristics, specifically the red, green, and blue (RGB) mixture which produces that particular color on the display. The maximum color map size is fixed by the window system, thus limiting the number of colors available at any particular time. This also restricts the number of allowable PEs configurable for SHMAPA, since many color entries are needed for the various PE's fading colors.

The complexity in generating the color map for SHMAPA exists because the colors used must be easily recognized as distinct when mixed and displayed together. Generating a color map is in general a simple problem; however, generating large sets of distinguishable colors requires careful planning. Especially with SHMAPA, the colors used are automatically generated depending on the number of PEs represented. The color maps could certainly be statically defined with reasonable effort if only a few small sets of PEs were needed, but this would be infeasible for many large sets.

Rather than deal with repeated complete generation of different-sized color maps by hand, we constructed an automated method for color map generation. This automated approach, given proper fine-tuning of parameters, creates color maps of any desired size with a distinguishable set of colors.

The method consists of combining a set of numerical functions to generate the desired color map values. Since a color is presented by its RGB mixture, these functions need only set up the appropriate ratio between the red, green, and blue intensities for each color. These intensities are actually maintained as three discrete integer values which lie in the possible range of intensities for the window system.

The functions used to generate these values are created and combined in relation to the colors they produce. Given three different base component colors (red, green, and blue), there are only four interesting combinations. Since a color need not be combined with itself, these combinations are blue with red, red with green, green with blue, and the combination of all three. For any given combination of col-

ors, many shades can be generated by the various intensity ratios. Before a distinguishable set of colors is chosen from these possible shades, they are collected to form an aesthetic progression.

To produce this pleasing smooth arrangement of colors, the shades generated for each combination as well as the order of the combinations themselves are considered. The order is set by the order of the generating functions as well as the characteristics of those functions. In each of the two-color combinations, the shades generated by the functions start with one of the colors at maximum intensity and the other at some low, but not minimum, intensity (see Fig. 7). (Colors tend to have an unnoticeable effect for most of the lower intensity levels, so it is not necessary to set the intensity to the absolute minimum.) To generate the first half of the shades, the function gradually increases the second color in intensity until both colors are at maximum intensity. The third unused color is turned off at minimum intensity for all of these shades. When both colors are at the maximum, the first color is then gradually decreased to a minimum intensity in generating the second half of the shades. In this way, the resulting starting and ending shades are approximately pure base component colors.

If the color combinations are arranged properly with matching pure component colors adjacent, the collection of colors flows pleasingly from one shade to the next without a sharp contrast. The order used for SHMAPA is as follows: pure blue to purple (blue and red) to pure red, pure red to orange (red and green) to pure green, pure green to aqua (green and blue), and then to white. Note that the aqua does not return back to pure blue, which was already used, but instead adds red to produce white with all three colors at maximum intensity.

For each of these color transitions, simple linear functions are used to either increase or decrease the intensity levels through the necessary ranges. These functions can be

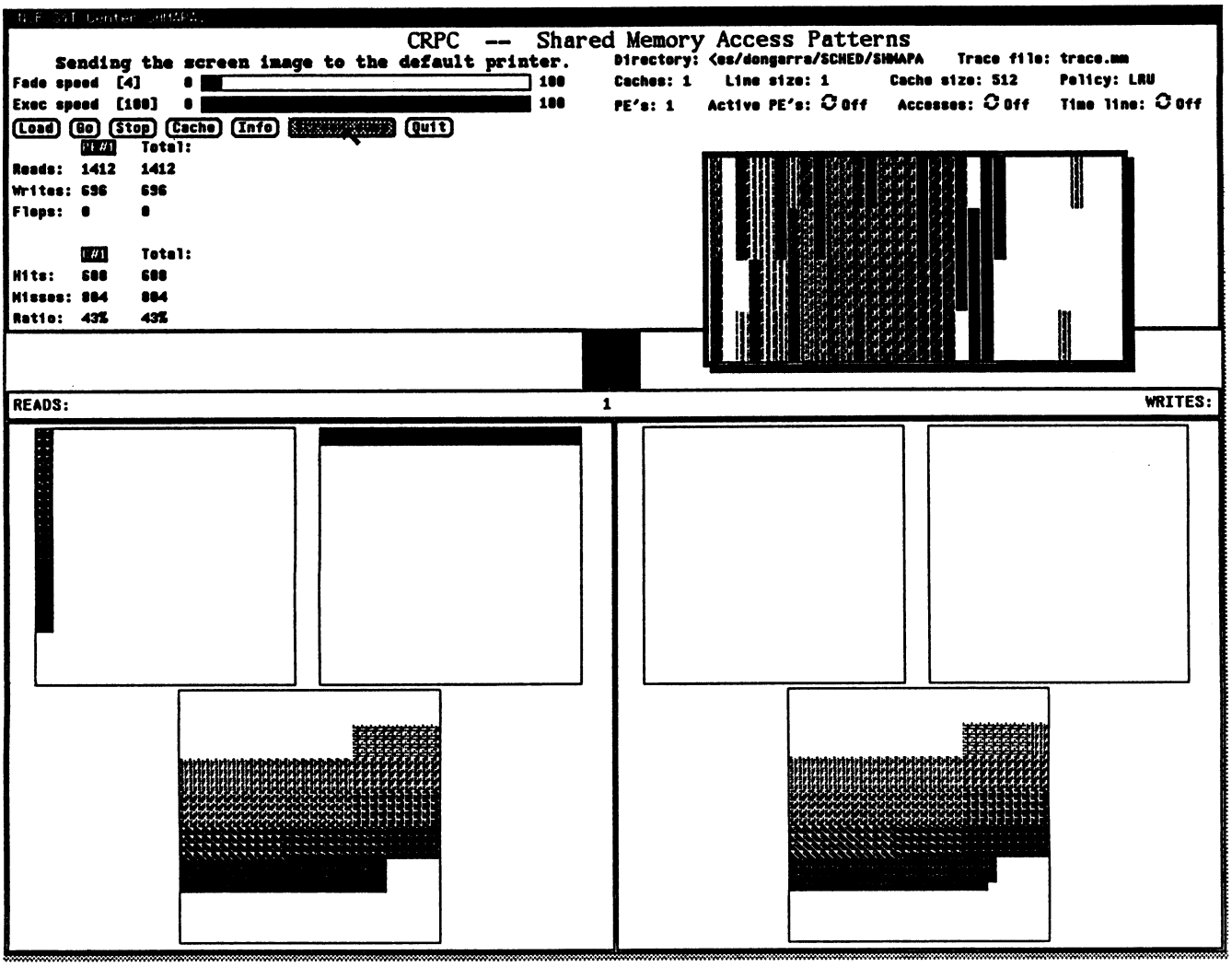

FIG. 7. Matrix multiply example.

fine-tuned by adjusting the rate which intensity is increased or decreased. Not all possible shades are actually needed, so the functions only generate a particular number of shades for a given color combination. By generating colors in even spacing across the entire set of all possible shades, a color map is generated with as many distinct colors as desired. The quality of distinction between two colors is determined by the space between the two shades, meaning the number of other shades which could be generated between them.

Finally, once the number of desired colors has been created, these colors are placed into a color map array, but more than just these pure colors are necessary. All of the colors need to be faded with time on the memory displays. To accomplish this, the desired number of faded shades for each color are included with the pure colors and placed into the color map array in subsequent locations. These faded shades are created by repeatedly lowering the intensities of the base components that make up the pure color. This decrease occurs at a predefined constant *decay rate.* No attempt to preserve the ratio of the three base components is necessary, so this decay rate only represents a constant value to be subtracted from each base component intensity value. The magnitude of this decay rate determines the extent to which the faded shades eventually approach black.

5.8. *Memory Access Flashing and Fading*

For each load and store access event animated by SHMAPA, the information for the event is processed by a special routine that appropriately illuminates the memory displays and fades the previously illuminated accesses. This routine is also responsible for updating the animated graphs and histograms, if activated.

The routine directs the drawing of the memory accesses on one of the two main memory canvases, as determined by the event opcode. Load accesses are drawn appropriately on the load canvas, stores on the store canvas. Since trace events actually reference blocks of several matrix elements, all of these elements must be drawn at the same time in the appropriate color of the PE making the access. When the cache system is active, the elements are drawn one at a time in order as they are searched for in the cache, since each given element may be part of a separate cache line requiring individual attention. Each element may need to be drawn not only on the cache window but also on the main memory load canvas depending on whether the element is found in cache. If a cache miss occurs, all the elements in the cache line must be drawn along with the originally requested element in both the cache window and the main memory load canvas. This is to reflect the loading of the line into cache. When the cache system is turned off, all accesses for a given trace event are displayed as a single block access. As an optimization, the entire rectangular region is drawn all at once on the main memory canvas.

The fading of colors is accomplished by storing information for each matrix element while animating. Each element has its own data structure which maintains a time stamp and a color map index. The time stamp is set to that of the trace event that last accessed that matrix element. The color map index simply keeps track of which color was last drawn to that element.

As the trace time passes, the current time stamp is compared to that for each matrix element to decide when its present color should be faded. With each new trace event, each individual matrix element is considered separately for fading. The fading occurs at a period determined by the fade speed setting. If the matrix element time stamp has aged by at least this period since the last fade, then the element is redrawn in the next faded shade. The color map index for that element is then adjusted to point to the new shade. The faded shades for each color are located together in the color map to allow a simple decrement for this adjustment.

Separate fading for each matrix element provides a more accurate view to the user. Rather than fade all elements on the display simultaneously, hence fading some elements too soon or too late, each element is faded on its own time. There is no blurring of the ordering of events by such a simplification of the fading activity.

5.9. *Portability*

SHMAPA was implemented to simplify its porting to different window systems. Currently it operates under either SunView or X Version 11.

To facilitate portability, the amount of codes dependent on the window system in use was minimized. Specifically, generic routines were created for basic window system functions. The window system is specified at compile time. Macros are included to make data type names equivalent (e.g., Panel_item in SunView is aliased to Widget when compiling for X); to specify the program code to be compiled (e.g., X or Sunview code); and to draw lines, rectangles, and highlight labels. In addition, some of the more complicated routines were provided, such as those for copying pixels and sliding window contents.

Separate files were created for the initialization and event handling functions of SHMAPA. The initialization routines include the routines to set up the screens, initialize windows, and install the color map. The event handlers are the functions that process events generated by the user interface. These are also significantly different for X Version 11 and SunView. Because of their length and dissimilarity, these programs were separated into different files for each of the window systems provided.

6. PARALLEL PROCESSING

Since our matrix algorithms use the BLAS, we simulate the use of multiprocessors in the calls to the BLAS. (This is

indeed the way the LAPACK project is exploiting parallel processing on shared-memory machines.) As an example, we describe the additions that we made to the Level 3 BLAS routine SGEMM.

SGEMM performs a matrix–matrix multiplication to update a third matrix, $C \leftarrow \alpha AB + \beta C$. If we look closely, we can see that each column of the resultant matrix can be computed in parallel. In others words, each of the following matrix–vector operations, $c_{*j} \leftarrow \alpha Ab_{*j} + \beta c_{*j}$, for $j = 1, \ldots, n$ where n is the number of columns of B and C, can be executed in parallel.

The main loop of SGEMM is

```
      DO 50, J = 1, N
         CALL SGEMV (TRANSA, NROWA, NCOLA,
     $                ALPHA, A, LDA, B(1, J), 1,
     $                BETA, C(1, J), 1)
50    CONTINUE
```

which performs the matrix–vector operations described above. To indicate the parallelism, we added calls to output the desired information to the trace file. The main loop became

```
      call par(0)
      DO 50, J = 1, N
         call group(0)
         call ops(idc,2*m*k,'sgemv')
         call r(ida,ia,ia+nrowa-1,ja,ja+ncola-1)
         call r(idb,ib,ib+nrowb-1,jb+j-1,jb+j-1)
         call r(idc,ic,ic+m-1,jc+j-1,jc+j-1)
         call w(idc,ic,ic+m-1,jc+j-1,jc+j-1)
         call group(1)
         CALL SGEMV (TRANSA, NROWA, NCOLA,
     $                ALPHA, A, LDA, B(1, J), 1,
     $                BETA, C(1, J), 1)
50    CONTINUE
      call par(1)
```

The call to *par(0)* indicates the beginning of a parallel section. A parallel section is divided into a set of groups each of which represents one unit of computation and is independent of all other groups within that section. Thus, all groups in a parallel section can be executed concurrently. The call to *group(0)* indicates the beginning of group. The call to *ops* records the number of floating-point operations performed during the computation and the name of the subroutine performing the computation. The calls to *r* and *w* record the load and write memory accesses of the subroutine. The call to *group(1)* ends the current group, and the call to *par(1)* ends the parallel section.

The resultant trace file, produced by the instrumented Fortran program, has the same format as our original memory analysis tool, but with the following additions:

- Indicate the beginning of a parallel section:
  6 Start parallel
- Indicate the end of a parallel section:
  7 End parallel
- Indicate the beginning of a unit of computation:
  8 Start group
- Indicate the end of a unit of computation:
  9 End group

An example of the trace output is displayed below:

```
0 1 100 100
6 Start parallel
8 Start group
5 1   16 strsv
1 1 1 16 1 1   0.93333
1 1 17 32 1 1   0.93333
2 1 17 32 1 1   0.95000
9 End group
8 Start group
5 1   16 strsv
1 1 1 16 1 1   0.95000
1 1 17 32 2 2   0.95000
2 1 17 32 2 2   0.95000
9 End group
8 Start group
5 1   16 strsv
1 1 1 16 1 1   0.95000
1 1 17 32 3 3   0.95000
2 1 17 32 3 3   0.95000
9 End group
.
.
.
7 End parallel
.
.
.
```

7. SHMAPA IN OPERATION

A number of simple functions are used in controlling the tracing operation of SHMAPA. A row of buttons on the main panel of the tool constitutes the central point of control. These buttons are labeled LOAD, GO, STOP, CACHE, INFO, SCREEN DUMP, and QUIT. There are also two sliders that control characteristics of the animation.

The LOAD, GO, and STOP buttons control the actual tracing. The LOAD button commences processing and loading of the trace file to be used to drive the tracing. The file selected at the time the LOAD button is clicked is considered to be the file to load. The GO and STOP buttons start and stop the tracing animation, respectively. After loading a trace file, clicking GO starts the tracing, which will continue until the entire file has been traced or until the

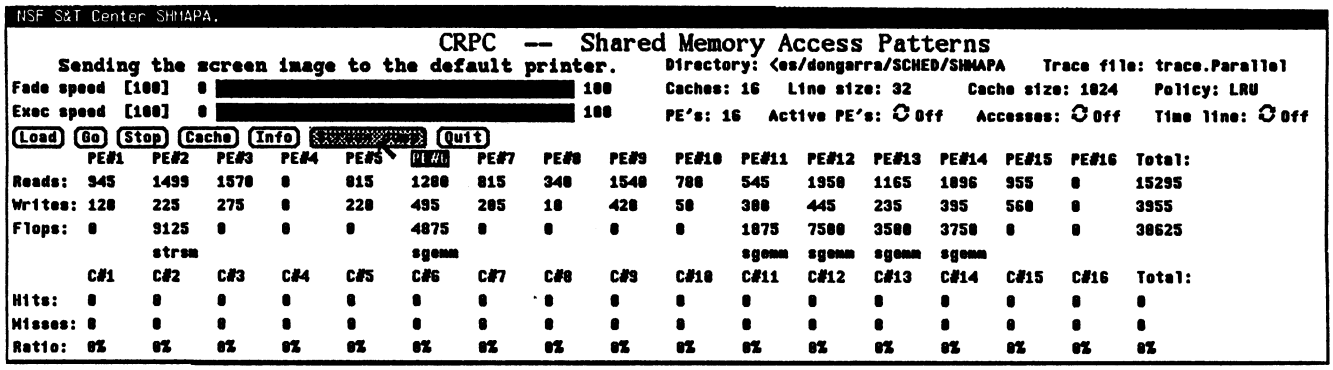

FIG. 8. Control panel.

STOP button is clicked. To change the system configuration during tracing, the STOP button may be clicked to temporarily interrupt the trace. Then after configuration changes have been made, GO may be clicked again to continue tracing where interrupted. Depending on the configuration change, some trace information may be lost at the point of interruption.

The CACHE button turns the cache system on and off. Cache is off by default when the tool is initiated, and with cache off no cache window is displayed. Upon clicking the CACHE button, the cache will be turned on and the current interactively set cache configuration will be initialized. The special cache animation window will then be displayed. This window consists of a rectangle divided into the proper number of identical smaller rectangles which each represent an individual cache memory. Repeated clicking of the CACHE button will successively turn the cache system off and on. Cache configuration may be altered with the cache on or off, but changes will take effect only if the cache is turned on.

The INFO button is used for producing text output from SHMAPA. If INFO is clicked instead of GO when a file has been loaded, then no animation will occur and only a text description of the tracing will be produced. This description can be controlled by various command line arguments to SHMAPA which produce output of the desired length and detail.

The SCREEN DUMP button is used to capture the current raster image of SHMAPA to be output to laser printers. This feature works with color or black-and-white printers.

The QUIT button exits the tool.

The two sliders on the main panel control the two animation characteristics manipulated by the tool. The Update speed slider controls the length of time the memory reference is held on the screen before fading away. The Execution speed slider controls the speed at which the trace file is processed; this slider expresses the event display speed as a percentage of the fastest possible speed.

Figure 8 shows SHMAPA's main user control interface.

8. AVAILABILITY OF OUR TOOLS

The software described in this report is available electronically via *netlib* [10]. To retrieve a copy, one should send electronic mail to netlib@ornl.gov. In the mail message, one should type

send index from tools
send shmap from tools

UNIX *shar* files will be sent back. To build the parts, one need only ship the mail file (after removing the mail header) into an empty directory and type "make".

9. CONCLUSIONS

SHMAP is intended to provide an animated view of a program's memory activity during execution. By playing back a program's execution, we are able to study how an algorithm uses memory, to experiment with different numbers of processors and different memory hierarchy schemes, and to observe their effects on the program's flow of data. Using such tools thus provides insight into algorithm behavior and potential bottlenecks in computationally intensive parallel algorithms.

REFERENCES

1. Kernighan, B., and Ritchie, D. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1988.
2. Hwang, K., and Briggs, F. *Computer Architecture and Parallel Processing,* McGraw-Hill, New York, 1986.
3. Ahuja, S., Carriero, N., and Gelernter, D. Linda and friends. *IEEE Comput.* **19,** 8 (Aug. 1986), 26–34.
4. Bershad, B. N., Lazowska, E. D., and Levy, H. M. PRESTO: A system for object-oriented parallel programming. Tech. Rep. 87-09-01, Department of Computer Science, University of Washington, Sept. 1987.
5. Brewer, O., Dongarra, J., and Sorensen, D. Tools to aid in the analysis of memory access patterns for Fortran programs, *Parallel Comput.* **9,** (1988/1989), 25–35.

6. Browne, J. C., Azam, M., and Sobek, S. Architectural and language independent parallel programming: A feasibility demonstration. Tech. Rep., Department of Computer Science, University of Texas, Austin, Feb. 15, 1988.
7. Carle, A., Cooper, K., Hood, R., Kennedy, K., Torczon, L., and Warren, S. A practical environment for Fortran programming. *IEEE Comput.* **20,** 11 (Nov. 1987), 75–89.
8. Dongarra, J. J., DuCroz, J., Duff, I., and Hammarling, S. A set of level 3 Basic Linear Algebra Subprograms. Argonne National Laboratory Report, MCS-P1-0888, Aug. 1988.
9. Dongarra, J. J., DuCroz, J., Hammarling, S., and Hanson, R. An extended set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Software* **14,** 1 (Mar. 1988), 1–17.
10. Dongarra, J. J., and Grosse, E. Distribution of mathematical software via electronic mail. *Comm. ACM* **30,** 5 (May 1987), 403–407.
11. Lawson, C., Hanson, R., Kincaid, D., and Krogh, F. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Software* **5,** (1979), 308–323.
12. Snyder, L. Parallel programming and the poker programming environment. *IEEE Comput.* **17,** 7 (July 1984), 27–36.

JACK DONGARRA is a Distinguished Scientist specializing in numerical algorithms in linear algebra at the University of Tennessee's computer science department and Oak Ridge National Laboratory's mathematical sciences section. Dongarra received a Ph.D. in applied mathematics from the University of New Mexico in 1980, a M.S. in computer science from the Illinois Institute of Technology in 1973, and a B.S. in mathematics from Chicago State University in 1972. Professional activities include membership in the Society for Industrial and Applied Mathematics and also in the Association for Computing Machinery (ACM). He is also an editor for *Communications of the ACM, ACM Transaction on Mathematical Software, Journal of Distributed and Parallel Computing, International Journal of Supercomputer Applications, Journal of Supercomputing, Parallel Computing,* and *Research Monographs on Parallel and Distributed Computing.* He has published numerous articles, papers, reports, and technical memoranda and has given many presentations on his research interests.

ORLIE BREWER is a scientific assistant in the mathematics and computer science department at Argonne National Laboratory. His interests include parallel processing, graphics interfaces, and visualization in scientific computing. He received the B.S. degree in mathematics from the University of Oklahoma in 1978 and the M.S. degree in computer science from the University of Oklahoma in 1986 and is a member of ACM, IEEE Computer Society, and SIAM.

JAMES ARTHUR KOHL received the B.S.C.E.E. degree in 1988 and the M.S.E.E. degree in 1989 from the School of Electrical Engineering of Purdue University in West Lafayette, Indiana. He is currently enrolled in the Ph.D. program in the electrical and computer engineering department of the University of Iowa. His research interests include graphic analysis of parallel systems, and parallel computer architecture and software development.

SAMUEL A. FINEBERG received the B.S.C.E.E. degree in 1988 and the M.S.E.E. degree in 1989 from the School of Electrical Engineering of Purdue University in West Lafayette, Indiana. He is currently enrolled in the Ph.D. program in the electrical and computer engineering department of the University of Iowa. His research interests include parallel computer architecture and performance evaluation.

Received June 1, 1989; revised January 10, 1990