Visualizing parallel execution of FORTRAN programs

by F. Szelényi
V. Zecca

As a first step toward the parallel execution analysis of FORTRAN programs, a tool called the Parallel Execution Profiler has been designed and implemented for the graphical postexecution analysis of parallel programs using the Parallel FORTRAN environment as a vehicle for both implementing parallel programs and tracing parallel events. The dynamic behavior of parallel execution is observed interactively in color graphs, which can be displayed concurrently with the source code, and in statistical summaries. This paper describes the implementation of our tool for parallel performance analysis with the aid of a parallelized application program from plasma physics.

Introduction

Complete performance analysis of parallel programs requires the analysis of both static and dynamic behavior. While static analysis can be done with optimizing compilers and code-restructuring tools that rely on estimated execution statistics to obtain well-optimized codes, dynamic execution analysis usually requires human comprehension and interaction. Static program analysis consists basically of dependency tests for parallelization [1] and estimates of program behavior (e.g., the number of iterations of a DO-loop). Unfortunately, the dynamic behavior of parallel programs is very complex, and inefficiencies and errors in these programs are difficult to detect. Starting from an optimized serial code, the goals of parallelization are to maximize parallelism inside the program, to minimize the overhead required by multitasking primitives, and to balance computational load among the parallel processors. Only well-balanced programs can provide optimal parallel execution. To achieve these goals, the capacity for fine tuning and debugging during parallel execution is desirable but difficult to implement. Parallel programming analysis presumes either that the parallel code can be simulated or that the parallel execution can be monitored. The overhead required by a monitor to produce execution traces generally results in some execution distortions (e.g., a different sequence of events), but monitoring does indicate the elements of program performance that should be analyzed more thoroughly by graphical analysis.

As a first step toward parallel execution analysis, a Parallel Execution Profiler (PEP) has been designed and implemented at the IBM European Center for Scientific and Engineering Computing (ECSEC), Rome, for the high-level postexecution analysis of parallel programs. This analysis is accomplished by examining performance data gathered during execution. Such performance-

history data might be recorded by tracing with a software or hardware monitor. Tracing requires cost-intensive monitoring of the parallel program, thus introducing execution distortions due to event recording, but with the advantage of portability. The monitoring data are collected in a trace file by storing the event type (e.g., "forking" a parallel work item to a processor), processor identification, timestamp, and certain additional information which must be filtered for important and significant data for the analysis. This situation leads to a complex task because of the increasing amount of information available from multiple processors and the multiplicity of possible combinations of events. As an alternative, techniques for parallel debugging of single processors under the control of a master [2] help in analyzing overall behavior, but monitoring seems more precise and efficient because of its simpler design. To interpret these parallel execution data, PEP visualizes the concurrent computation by referring to the source program and to previously recorded variable values. IBM Parallel FORTRAN [3] has been used as a vehicle both for implementing parallel programs and for tracking all events of a parallel program that are stored in the PEP database. The contents of the database can be visualized by color graphics and statistical charts using the Graphical Data Display Manager (GDDM) [4] on an IBM 3270 or 5080 color display. As a test and demonstration case, we have analyzed performance data obtained by dedicated runs of a magnetohydrodynamic parallel code on an IBM 3090[1] processor with Vector Facility (VF) and six processors.

The processor model of the PEP database is based on state diagrams, which show all the states of virtual processors. Virtual processors have been introduced because modern parallelizing tools such as Parallel FORTRAN rely on this concept, but it is also possible to show the status of the real (i.e., physical) processors. Several possible methods for storing parallel execution information have been found in the literature: The relational approach [5] or state charts [6] are thought to be more efficient because traditional queueing models [7] fail. *State diagrams* seem to be the most successful model because they lead to a clear and terse description of processor activity. State diagrams are simple directed graphs, with nodes denoting states and arrows (labeled with events and guarding conditions) denoting transitions between, for example, idle and active states. However, a more sophisticated technique is required to describe the more complex systems, because a complex system description will otherwise result in a chaotic state diagram. Therefore, a hierarchy was introduced into the PEP processor model to describe the dependency among the several processors during parallel execution (e.g., processor one must wait for the lock obtained by processor two) on a time- and statement-level basis.

The logical connections among different processor states are visualized with dependency graphs by displaying possible dynamic dependencies such as waiting for a lock or for the termination of a task. Parallel execution is shown (**Figure 1**) in a two-dimensional plane; the horizontal (x) axis shows the processors in numerical sequence from left to right and the vertical (y) axis shows time increasing from top to bottom. Blue lines indicate dependencies between processors. The time windows of parallel execution can be selected and customized by zooming, cutting, paging, or direct time and scaling input. Overall program efficiency can be summarized in charts by taking into account parallel inefficiencies such as waiting for locks, events, and tasks. These performance graphs can be saved as print files for page printers. A view of the source code for high-level analysis is provided by means of display windows. Source code statements can be displayed with respect to selected time and processor values (in a manner similar to the VS FORTRAN interactive debugger [8]) within the analysis graph, and conversely, a graphical display of the dynamic behavior of the parallel execution can be generated from the source code by pointing to statements in the code. Furthermore, the values of selected variables can be displayed as a first step toward a parallel debugger. A parallel debugger helps the programmer to write good parallel code by generating code for tracing, changing the priority of processes, suspending the execution of processes, simulating events and processes, forcing nondeterministic constructs, querying the program state, and eventually checking assertions [9].

Parallel performance analysis is completed by using execution statistics to summarize the use of parallel primitives with related overheads, the parallel loop analysis (e.g., chunk size, number of iterations assigned to a processor, and total number of iterations), the overhead due to inefficient parallel programming (e.g., waiting for events), the use of subroutines in parallel, and the efficiency of subroutines. These statistics can be displayed in simple tabular form or in colored charts.

Simple postprocessing tools for parallel programs, such as MTDUMP on Cray multiprocessors, can generate an execution graph in a postprocessing analysis step but do not allow a flexible on-line study. For the VM/EPEX environment, a speedup analyzer [10] has been implemented to examine algorithmic performance, parallel processing overhead, and the distribution of parallel work items by first monitoring the parallel program. Executing this program on a uniprocessor and then on a multiprocessor model allows the study of

[1] 3090 is a trademark of International Business Machines Corporation.

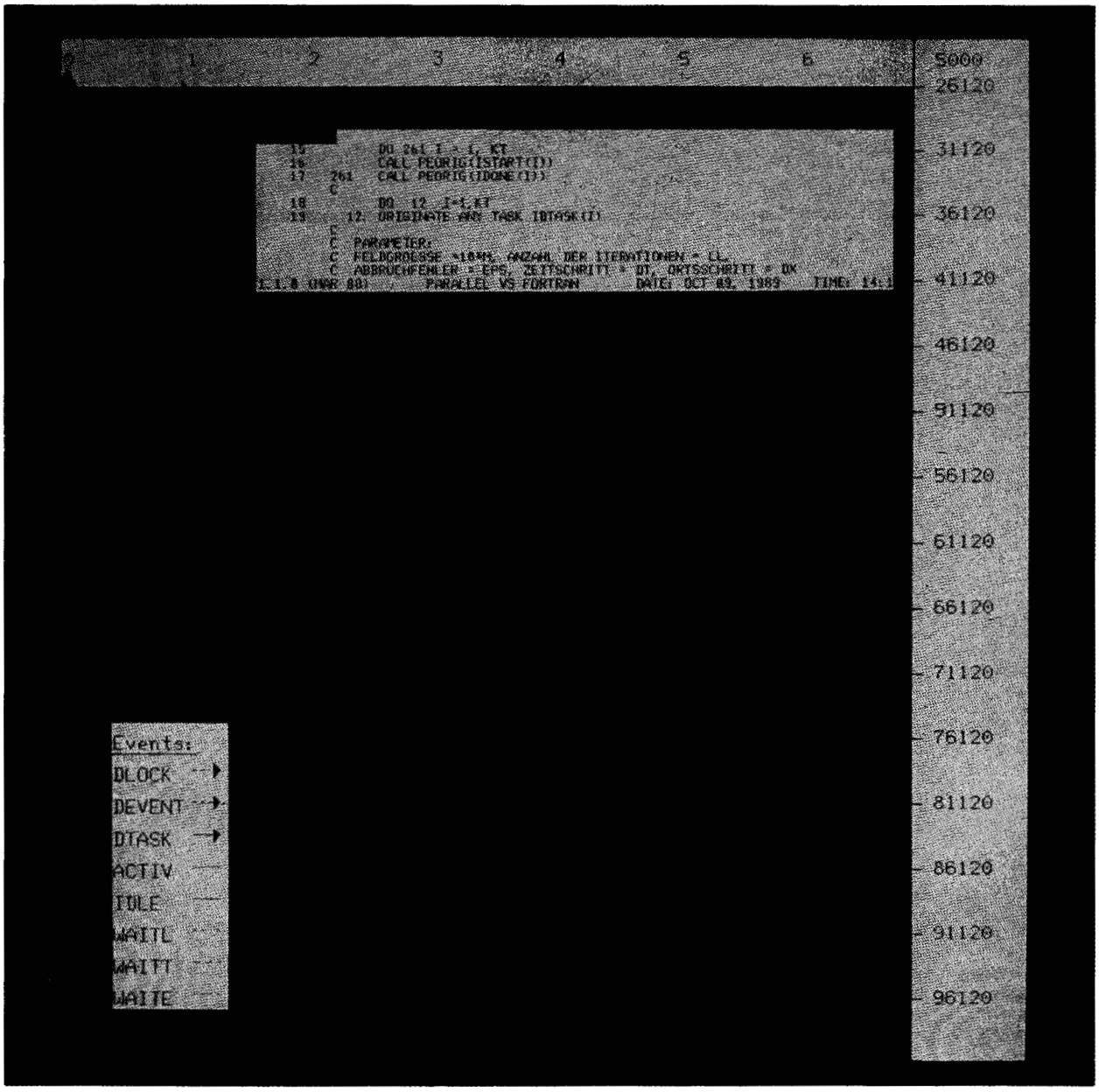

Figure 1

Profile graph for the MHD master/slave implementation, showing the source text window indicating in yellow the current statement spotted by the green mark in the time lines. The help window contains the color line explanations. The chart focuses on the high task dependencies caused by event and locking synchronization in the lower part.

parallel performance as a second step. Similar tools have been developed for the analysis of workstation displays and message-passing multiprocessors [11], as well as for shared-memory multiprocessors (i.e., processors with the SCHEDULE package [12]), all based on the execution analysis of a monitored parallel program. Static analysis, merged with performance estimations through the use of a graphical editor [13], helps in the parallelization and fine-tuning of programs. Another approach for the dynamic analysis on VLIW machines is a trace-scheduling compiler [14], which permits program flow study with interactive debugging.

Parallel FORTRAN (PF) provides an environment for the execution of parallel programs consisting of parallel constructs that allow a set of *FORTRAN processors* (FP) to be associated with a single FORTRAN program.

Parallel FORTRAN has recently been merged into the IBM VS FORTRAN Version 2.5 environment. Parallel execution is achieved by having multiple FPs associated with the program, and a root FP controls the parallel execution of all program-defined FPs. The programmer defines the work to be done by the FPs and declares at execution time the number of FORTRAN processors which are controlled by the PF library. Parallel FORTRAN provides a trace facility for recording all of the significant events of parallel program flow. This trace is used by the Parallel Execution Profiler as a monitoring tool. As with any trace facility, it is difficult to improve the parallel application without a deep and thorough knowledge of the PF trace output. The Parallel FORTRAN trace facility can be enabled or disabled at run time. The following events, which are significant for the Parallel Execution Profiler, are recorded:

- Start and end of program execution.
- Origination, assignment, completion, and termination of PF tasks.
- Task access to shared or copied memory.
- Locking and event-handling management.
- Parallelized DO-loops and parallel sections.
- Waiting for events, locks, tasks, loop chunks, and parallel sections.
- User-defined events.

A series of trace records are produced, each showing a time stamp, a FORTRAN processor number, a real processor number, the originating task, the trace event name, and additional information. The format of the time stamp can be selected by the user (for example, microseconds or seconds can be specified). Additional information (e.g., the chunk size of a parallel DO-loop or even user-written information) completes the PF trace output. The overhead for tracing parallel execution varies between 3% and 250% for a set of parallel test programs [15], depending on the number of parallel events: Many tasks all accessing the same shared variable in a critical section require an extensive trace recording. The higher overhead figure occurs for heavy tracing activity when intensive and prolonged input/output is performed on the trace file. These overheads have been measured for several programs using different PF parallelization strategies executing with six processors on a dedicated IBM 3090 System Model 600E under the MVS/XA[2] operating system.

Visualizing parallel execution

The Parallel Execution Profiler uses the Parallel FORTRAN trace as input to generate the database storing processor state diagrams: A processor can have five different states: *idle*, *active*, *wait for a task*, *wait for an event*, and *wait for a lock*. **Figure 2** shows how the state of a processor changes with the execution of PF primitives. (For example, a processor state changes from *idle* to *active* in computing a chunk of a DO-loop, while a processor enters the *idle* state when the work item has been terminated and no new work item has been assigned.) The inefficient states should be minimized to obtain good parallel performance. The second input for PEP consists of the Parallel FORTRAN compiler output listing for the connection between the parallel execution graphs and the source program.

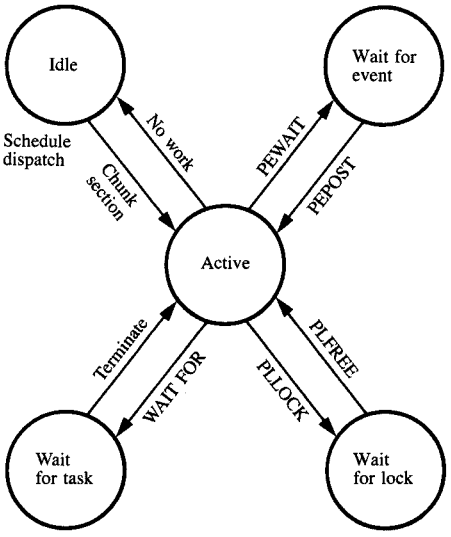

Figure 2

State diagram of a processor, showing the five different states for a processor using Parallel FORTRAN primitives. The arrows represent the PF statements for the transit from one state to another state.

Visualization of parallel execution starts in a two-dimensional plane, with the x-axis representing the processor scale and the y-axis representing the time scale. Time lines representing the actual state of a processor develop from top to bottom, showing dependencies between processors with horizontal lines which indicate the type of dependency. The time scale can be adjusted by defining the start time of the actual window, adjusting the time increment, paging up and down, zooming, and moving the time axis. The user can focus on important parts of the parallel execution with these PEP functions.

[2] MVS/XA is a trademark of International Business Machines Corporation.

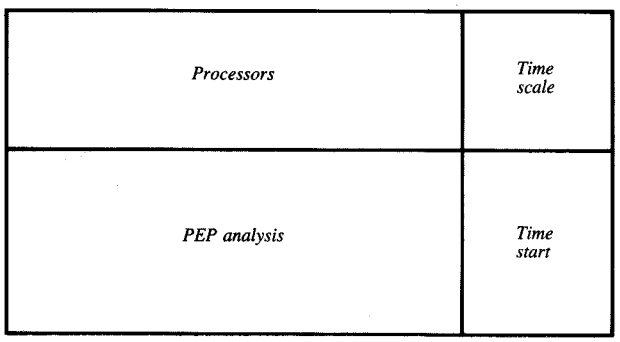

Figure 3

Schematic profiling: *Processors* specifies the FORTRAN processors from the root task to the number of processors defined, *time scale* the actual increment in the time axis, and *time start* the actual starting time (these two parameters can be modified interactively by the user); *PEP analysis* represents the graphical display of the parallel program execution.

The connection to the source program can be done by opening the source program window at the current time, or by pointing to an event specified by the processor number and time stamp. The program window can show parts of the source program in a manner similar to the VS FORTRAN interactive debugger, or it can show the source statement with the subroutine name. The window is updated with respect to the current time stamp. The user can jump to the next executed event at the displayed statement, for either the same processor or another processor. Furthermore, the values of predefined variables can be displayed for each processor and time. These variables are recorded in the PF trace through an interface program provided by PEP, which is described later.

The efficiency of a piece of code can be computed by pointing in the graph to the extremes of the selected piece. The actual real (physical) processor state can be determined by pointing within the graph, with PEP showing the number of FORTRAN processors associated with each of the real processors (eventually in an interleaved mode). Print files for page printers are saved by PEP for printing and documenting purposes. All PEP functions are assisted by help text windows, which explain the PEP messages and function keys. Error messages can be explained in more detail by pressing the help key that gives the user suggestions for solving the problem.

Parallel execution statistics can be shown in simple tables or in more complex color charts, which can be saved for printing purposes. The statistical analysis can be guided through the windows interface, and the following information can be obtained:

- A summary of parallel statements in absolute numbers and elapsed times.
- A parallel loop analysis: chunk size, number of iterations, and number of processors with minimum, maximum, and average values. Furthermore, the number of parallel loop executions is summarized, for hot-spot detection.
- Overhead caused by inefficient states for the FORTRAN processors.
- A summary of subroutine usage.
- Calculations of subroutine efficiency.

Examples of parallel execution statistics are shown in Figure 1 and in Figures 4–7, and are described later.

PEP performance data

An on-line analysis of parallel program execution is divided into two parts, performance profiling, as shown schematically in **Figure 3**, and execution statistics on charts. All Parallel Execution Profiler features are invoked by function keys. Figure 3 depicts a two-dimensional plane with a processor x-axis starting from the root processor and a time y-axis starting from the beginning of execution. The x-axis shows all processors defined in the parallel program, while the *time scale* is determined by the screen size. The unit of time corresponds to the setting in the Parallel FORTRAN trace time scale. The starting time and incrementing time can be set by the user at any time. The *PEP analysis* window shows the time lines with dependencies among the processes. This section gives an overview of the information provided and the use of the analysis tool; a complete description can be found in [16].

Performance profiling is assisted by interactive help functions to explain the meaning of symbols and lines shown in the profiling chart. Explicative messages for each function are shown in the beginner mode, which can be disabled for faster analysis. Error messages are displayed in special windows, and PEP-proposed corrective actions should allow a fast user response without losing previous analysis work. Parallel analysis of the evolution of the FORTRAN processor states is done in the time direction by paging or by direct setting of the actual screen starting time and time increment. The program flow with branching dependencies due to parallel actions (e.g., a processor has forked to a new processor because of a SCHEDULE statement) can be observed in the time direction. This profiling chart can be saved for plotting and documentation purposes by pressing a function key. Zooming and cutting functions allow the detailed study of critical program parts: Zooming focuses on these parts by readjusting the screen starting time and increment, while the cutting function excludes certain events, with notification to the user.

Branches from a processor are specified in detail by indicating the Parallel FORTRAN statements that caused each branch. This function can be enabled and disabled by the user at any time with any degree of detail on the Parallel FORTRAN statements.

Measurements of parallel efficiency in the user-specified time interval can be done by simply pointing to the area of interest on the screen, and these measurements allow the programmer to improve the efficiency of program fragments. Parallel efficiency is calculated as the sum of the times when the processors are in the active state, divided by the elapsed time interval of the fragment multiplied by the number of processors. The state of the real (physical) processors can be obtained for a selected time: PEP indicates with different colors the FORTRAN processors that are executed on a real processor. The parallel FORTRAN trace contains its own time-stamp units, which must be converted to elapsed time. The time pointed to in the window can be further detailed by altering the time-stamp unit setting (i.e., the unit of incrementing or the relative starting time).

A set of interface routines callable from a Parallel FORTRAN program has been developed to put the values of selected variables in the trace output. These variables can be displayed in the PEP analysis display. The user must add the interface call in the application before the compiling step, and the recorded values are stored in the trace generated by Parallel FORTRAN. For example, to display the double-precision variable "VARIAB" the following call must be included:

```
CALL PEP$R8 ('VARIAB', VARIAB)
```

These values can be displayed by pointing to the processor and time event in the profile chart. If a value close to this point has been recorded, the value and variable name with the issuing internal statement number are shown in an extra window. This is a first step toward parallel debugging and gives a fuller response to the program state query.

The connection between the graphical analysis display and the source program code is obtained either by displaying the current source statements in a window or by pointing the cursor to the time and processor point (similar to displaying a preselected variable value), thus displaying the actual program or subprogram name with the internal statement number and the statement text. Conversely, a specific parallel statement can be selected by designating its internal statement number and (sub)program name to show the first occurrence of this parallel statement. The next or previous occurrences of this statement on the same or another processor can be pointed out.

A more advanced source text connection is provided by opening a new, movable window containing the program text, the program name, and the internal statement numbers, and pointing to the actual statement as the VS FORTRAN interactive debugger would do. When the window is opened, the actual statements are displayed, but, as shown in the previous paragraph, the window can be visualized by pointing into the parallel program flow graph. The same functions are provided for moving to the next or previous occurrences in the time and processor axes.

After termination of the on-line parallel program analysis, statistics in the form of charts or tables help the programmer to estimate overall performance and find bottlenecks. The summary of the parallel statements counts the number of executed parallel statements (e.g., generating a task, loop chunk) and computes the elapsed time required for these parallel statements (for an example, see **Figure 4**). The overheads are calculated from the Parallel FORTRAN overhead measurements for all parallel primitives [17], giving the programmer an estimate of the overhead introduced with parallelism. Unavoidable statements (e.g., originating a task) can be excluded for a better overview. In **Figure 5**, parallel loop execution is summarized by giving the minimum, maximum, and average values for the number of processors used, the number of iterations, the chunk size per processor, and the number of times the parallel loop has been executed. In general, Parallel FORTRAN is able to balance the parallel execution of a loop so that, as in this case, the minimum and maximum chunk sizes and processor numbers are the same. These data can be displayed for a specific parallel loop or for a single specific performance characteristic (e.g., chunk size) with bar charts. In **Figure 6**, program inefficiencies such as waiting for the termination of tasks, for events, and for locks, are summarized for the individual processors with percentages and elapsed times. In **Figure 7**, the active times, the time spent in parallel loops, and the overhead due to waiting for events, locks, and tasks are summarized for each subroutine with elapsed times. These data are required for the calculation of the overall parallel efficiency e of a subroutine s with the following formula:

$$e_s = \frac{np(T_t - T_d)}{\sum_{i=1}^{np} T_i^s},$$

where np is the number of processors, T_d and T_t are the clock time when the task containing the subroutine s was dispatched and terminated, respectively, and T_i^s is the time required on processor i for the execution of subroutine s.

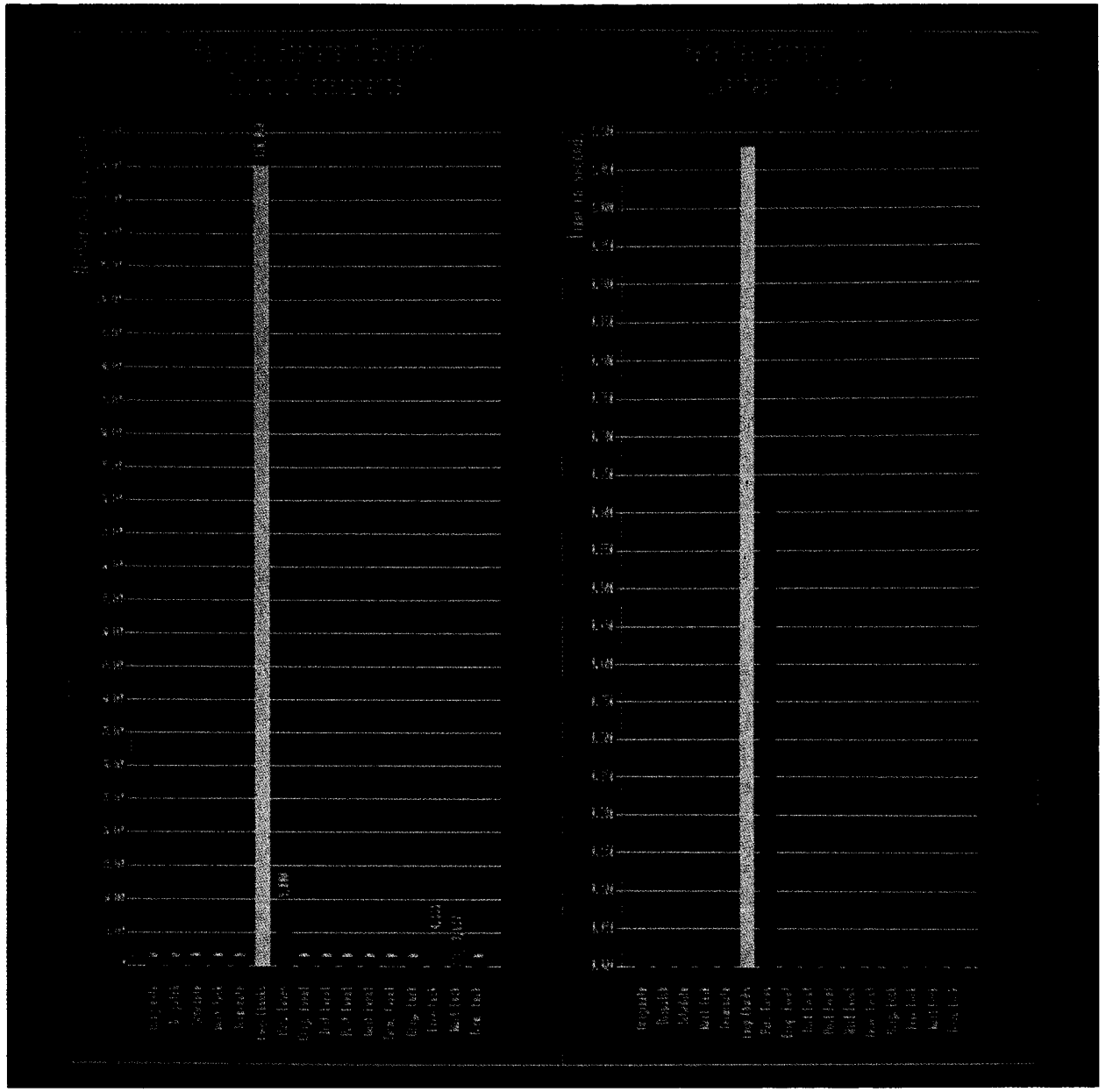

Figure 4

Overhead of the MHD implementation. The overhead summary of the Parallel FORTRAN language elements is shown by displaying the absolute number of executed Parallel FORTRAN statements and the corresponding elapsed times spent in performing these calls. The elapsed times are calculated as previously measured for the current Parallel FORTRAN implementation [17]. Statements whose overhead is already known or unavoidable (e.g., *originate a task*) can be excluded for a better overview.

A magnetohydrodynamic parallel code

The magnetohydrodynamic (MHD) system of partial differential equations [15] describes the motion of a plasma influenced by a magnetic field, as in a fusion reactor. The main physical variables in two dimensions are the pressure and velocity of the plasma and the components of the magnetic field. The MHD equations are approximated in the points of a grid by discrete difference equations, resulting in a large system of algebraic equations to be solved. The numerical method used in the MHD system leads to explicit finite-difference equations, in which only the values at time step $n - 1$ are

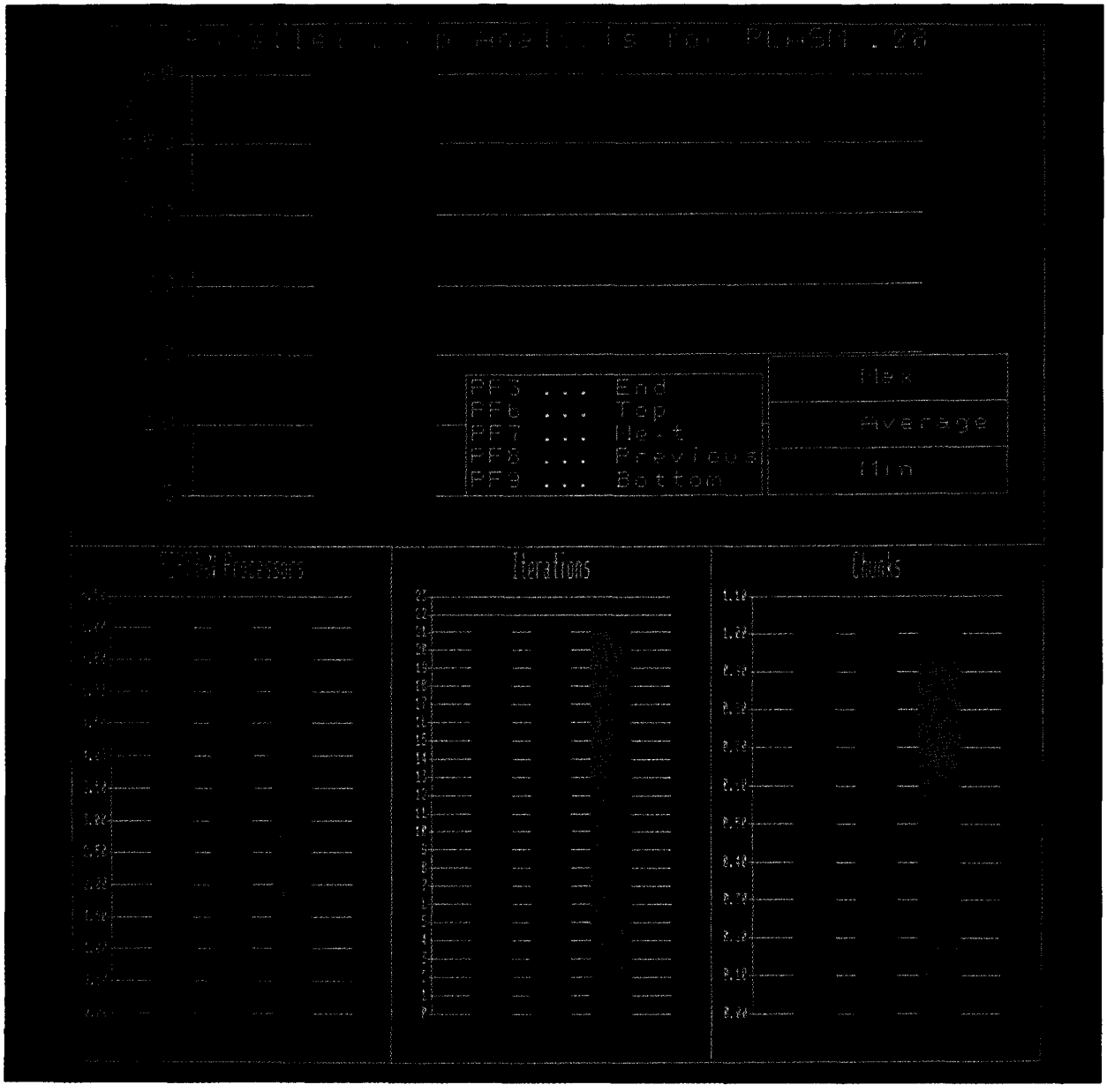

Figure 5

Statistics of a parallel loop. These charts show the number of executions, the minimum, maximum, and average value for the number of iterations, the number of FORTRAN processors used, and the number of chunks per processor for the parallel loop from the program PLASM at internal statement number 28.

required to update the physical variables at the new time step n. Therefore, all values of the previous state are known, and no data dependencies between variables of the current and the previous state arise. These numerical methods are easily parallelizable. The calculations for a given number of time steps or to reach the steady state comprise the following steps:

1. Definition of the variables and calculation of the initial state of the plasma and magnetic field.
2. Time-stepping from $n = 1$ to NT of the physical variables computed in all the inner grid points.
3. Terminating the program when the steady state is reached and the convergence condition for all grid points is fulfilled.

Figure 6

Statistics for the overheads of the root task. These charts show the overheads due to wait for tasks, events, and locks for the FORTRAN root task on FORTRAN processor 0. All FORTRAN processors are selectable, and the total overhead summary can be analyzed as well. The overheads are listed in percentage of the total time and in elapsed times.

The MHD system has been implemented in four different parallel versions: *Fine-grain parallelism* has been exploited by using 1) Parallel FORTRAN automatic parallelization and 2) Parallel FORTRAN language primitives for in-line parallelism. 3) Macrotasked and 4) event-synchronized versions exploit PF *coarse-grain* parallelism. The version with automatic parallelization has been prepared by compiling the vectorized sequential code without any changes. Using the Parallel FORTRAN language statements PARALLEL CASES and PARALLEL LOOP exploits in-line parallelism, and a few optimizations with compiler directives characterize the second fine-grain implementation. The slightly unsatisfactory performance results can be explained partly by the reduced vector length due to parallelization.

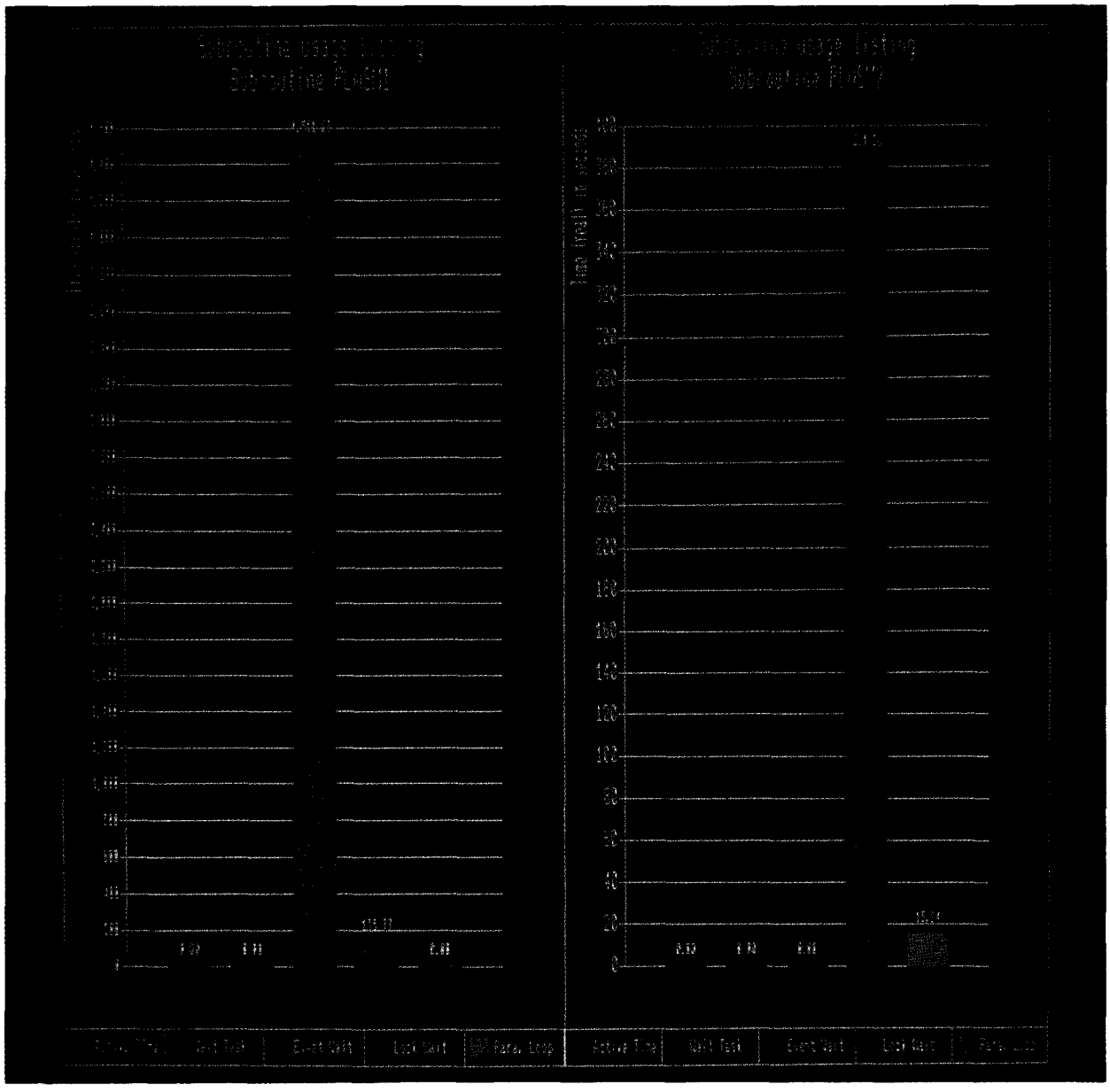

Figure 7

Subroutine usage of the master and the slave task, showing the times spent in waiting for the completion of tasks, of events, and acquisition of locks. The master task in subroutine PLASM1 waits for a remarkable time for completion of the slave tasks, whereas the bottleneck in the slave tasks within subroutine PLASM2 occurs because of the update of global variables within critical sections. The active time is excluded from the charts to provide a better overview.

Examination of the data structures within the most time-consuming parts of the program shows that within any single time step there are no data dependencies on neighbors of the physical values. Therefore, the computational domain can be divided for coarse-grain parallelism into equally spaced strips in such a way that the vector length in the innermost iteration loops remains unchanged. The pure macrotasked version assigns a strip to each task and then waits for the termination of all tasks. Successively, the convergence is tested in the main program. The remaining potential parallelism is exploited by using the automatic parallelizing capabilities of Parallel FORTRAN. Synchronization using events is more efficient: The work is assigned initially to one

Table 1 Parallel performance improvement for MHD: Times were measured on an IBM 3090 System Model 600E with Vector Facility under the MVS/XA operating system using Parallel FORTRAN.

Method of parallelization	*Speedup with six processors*
Automatic parallelization	1.85
PF language additions	1.85
Macrotasked	3.91
Event synchronization	4.08

master task and $np - 1$ "slave" tasks (np is the number of processors) in the main program, and the root task waits for the termination of all other tasks after dispatching. Updating of global variables is performed in a critical section by using Parallel FORTRAN locks, while synchronization and convergence are controlled by the master task through the use of Parallel FORTRAN events. The parallel performance improvements of all versions are summarized in **Table 1**. We want to point out that the fastest execution speed is obtained for the versions using event synchronization and macrotasking combined with automatic parallelization.

These four parallel implementations have been used as a test case for the Parallel Execution Profiler by running them with the Parallel FORTRAN trace facility.

Experimenting with PEP

The four parallel implementations of the magnetohydrodynamic parallel code (as described in the previous section) were executed on a dedicated IBM 3090 System Model 600E with Vector Facility using all six of its processors. The Parallel FORTRAN trace facilities were able to request all of the information available to monitor parallel execution. The execution time for the parallel runs with trace enabled was greater by a factor of 2.5 than that of the same run with trace disabled on all six processors. The trace data occupied between 15 and 70 MB of disk space. A large quantity of data for the parallel execution trace was generated for the macrotasked version with additional automatic parallelism because of the many synchronization points for nested parallelism that occur in the run. This trace is three times larger than the version using pure automatic parallelism. Another 70 MB of monitored data were generated for the event-synchronized version because in this case a huge number of parallel events occurred.

The data generated by the Parallel FORTRAN trace facility from the parallel execution of the MHD programs were stored in a database for use in the graphical analysis. Although the analysis tool was designed to hold all data in memory, significant amounts of data were stored on disks to provide portability. This could be avoided by using IBM System/370[3] Extended Architecture (XA), which can place up to 2 GB of data in virtual storage using the VS FORTRAN dynamic COMMONs feature.

The first goal of our study was to understand the unsatisfactory performance results of the fine-grain parallelized implementations. The profiling graph showed high intertask communication activity for the parallel loops within the automatic parallelized version. Measuring the efficiency of one series of parallel loops revealed that the loops were too fine-grained (i.e., the chunk size was too small). This fact was confirmed by inspecting the statistics of parallel loop performance and overhead estimations. For example, the parallel execution of the parallel loops (see Figure 5) shows very fine-grain execution, because the chunk size was always one and only 21 iterations were performed.

The second fine-grain implementation of MHD using Parallel FORTRAN language elements showed similar behavior in the profiling chart, but examination of the overhead required for parallel primitives showed that parallel sections implemented with the PARALLEL CASES statement were too expensive with respect to the computational work performed (see Figure 4). Furthermore, a significant amount of time was required to update the global variables within critical sections using the Parallel FORTRAN locks.

Further improvement of the MHD macrotasked version can be achieved by removing the additional overhead of the automatic parallelized loops. The execution statistics show that the loops increase parallel overhead by a factor of five. (Because of overhead for nested parallelism, the execution of parallel loops was not balanced, and the loop chunks were consequently assigned to three, four, or six processors.) Furthermore, the computational grain was too small for additional nested parallelism overhead.

Figure 1 shows the first part of the profiling graph for the event-synchronized version. The source program window is shown at the upper right. The current statement is identified by the yellow highlight corresponding to the green spot at the upper left. At the lower left, the help window contains explanations for the colors used in this profiling chart. The master/slave coarse-grain implementation can be followed: First, the tasks for the parallel processors are originated by the root task. Work is assigned to the master task residing on subroutine PLASM1 and to the slave tasks on subroutine PLASM2. Then the root task enters an idle state, while computations are performed by the FORTRAN processors numbered one to six. By zooming in to the profile graph, the dependencies among the tasks can be recognized as arising from event synchronization and locking.

[3] System/370 is a trademark of International Business Machines Corporation.

Toward better parallel programming

The Parallel Execution Profiler helps to improve the parallel performance by visualizing the dependencies among the parallel work items. These dependencies are related to scheduling of subroutines and waiting for their termination, waiting for the completion of all parallel loop chunk computations, synchronization of events, and critical sections implemented with locks. A good profile is achieved for a moderate number of processors and parallel events. Indication of the type of parallel event helps to improve parallel performance by allowing the programmer to reduce substantially the number of unnecessary fork/join primitives. A high number of synchronization points makes the profiling graphs difficult to analyze. In this case, zooming can help the programmer to focus on important parts of the program flow.

The connection of the graphic profile to the parallel source code permits better understanding of the parallel programs. Reviewing the parallel execution of time-consuming program sections can improve performance substantially. The best solution is to display the source code with the context in a special window. The displaying of preselected variable values is useful for removing errors introduced with parallelism, but the user must first identify the program section where the error occurs. It turns out that computing the efficiency of program fragments in the time scale helps in estimating the quality of the parallelization.

The parallel statement statistics show all important parallelism overheads associated with parallel loops (either automatically parallelized or explicitly coded). The statistics for parallel loop analysis were very useful during the fine tuning of the parallel loops. It was found that fine-grain parallelism also improved performance on nondedicated executions. The subroutine statistics immediately revealed bottlenecks for both coarse- and fine-grain parallel execution by indicating the overhead due to inefficient programming.

Future developments must be focused on a parallel debugger, where general breakpoint setting and task manipulation for parallel program flow will help in debugging and improving parallel algorithms. A direct interface to Parallel FORTRAN such as the trace facility can help to debug a parallel program by using different windows for program flow display, parallel execution manipulation, and statistics.

Conclusions

The Parallel Execution Profiler (PEP) helps in analyzing the execution of parallel programs by visualizing data and program dependencies among processors due to parallelism in profiling graphs and statistic charts. The IBM Parallel FORTRAN environment can be used for parallel programming and monitoring purposes, resulting in a compiler output indicating the parallelized program parts and a trace output from parallel program execution. Both outputs are the basis for the PEP visualization analysis. The Parallel FORTRAN trace does not require expensive monitoring of the source code, but monitoring the parallel execution results in the potential problem of distorting real execution due to the overhead of trace recording. An alternative would be a hardware monitor, but that would inhibit portability.

Based on color graphics implemented by the Graphical Data Display Manager (GDDM), PEP, our interactive performance analysis profiler, is an easy-to-use tool for visualizing parallel program execution. All functions are assisted by help text within windows. The analysis tool gives an overview of parallel execution by showing program flow structure with line charts evidencing the task dependencies. The connection to the source code allows an easy performance improvement by identifying critical program parts. Furthermore, display of preselected data values is a first step toward debugging the parallel program by examining the program state during the parallel execution.

The statistics generated by our analysis tool help to reduce the parallel overheads and point out the efficiency of single program parts. Statistics improve the code tuning of Parallel FORTRAN, especially for parallel loops, by identifying critical overhead due to parallelism.

References

1. F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *Research Report RC-13115*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1987.
2. H. Garcia-Molina, F. Germano, and W. Kohler, "Debugging a Distributed Computing System," *IEEE Trans. Software Eng.* **SE-10**, 210–219 (1984).
3. L. Toomey, E. Plachy, R. Scarborough, R. Sahulka, J. Shaw, and A. Shannon, "IBM Parallel FORTRAN," *IBM Syst. J.* **27**, 416–435 (1988).
4. *Graphical Data Display Manager Base: Programming Reference,* Order No. SC33-0101, available through IBM branch offices.
5. K. Schwan and J. Matthews, "Graphical Views of Parallel Programs," *ACM SIGSOFT Software Engineering Notes* **11**, 51–64 (1986).
6. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Sci. Computer Program.* **8**, 231–274 (1987).
7. H. Kobayashi, *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology*, Addison-Wesley Publishing Co., Reading, MA, 1978.
8. *VS FORTRAN Version 2 Interactive Debug Guide and Reference,* Order No. SC26-4223, available through IBM branch offices.
9. G. S. Goldszmith, S. Katz, and S. Yemini, "High Level Language Debugging for Concurrent Programs," *Research Report RC-14341*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1989.
10. K. So, A. Bolmareich, F. Darema-Rogers, and V. Nortran, "A Speedup Analyzer for Parallel Programs," *Research Report RC-12546*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1987.

11. A. D. Malony and D. A. Reed, "Visualizing Parallel Computer Performance," *CSRD Report 812*, Center for Supercomputing Research and Development, University of Illinois, Urbana, 1988.
12. M. Seager, S. Campbell, S. Sikora, R. Strout, and M. Zosel, "Graphical Multiprocessing Analysis Tool (GMAT)" *Report No. ISCR-87-2*, Lawrence Livermore National Laboratory, Livermore, CA, 1987.
13. D. Gannon, D. Atapattu, M. Lee, B. Shei, A. Saini, and J. Lee, "The Sigma System: A Tool for Parallel Program Design," *Proceedings of the Third International Conference on Supercomputing (ICS'88)*, May 15–20, 1988, International Supercomputing Institute, St. Petersburg, FL, Vol. 3, pp. 157–163.
14. R. Gupta, "Debugging Code Reorganized by a Trace Scheduling Compiler," *ICS'88 Proc.* **3**, 422–430 (1988).
15. W. Gentzsch, F. Szelényi, and V. Zecca, "Use of Parallel FORTRAN for Some Engineering Problems on the IBM 3090 VF Multiprocessor," *Parallel Comput.* **9**, 107–115 (1988).
16. F. Szelényi, "Analyzing Parallel FORTRAN Programs with an Execution Profiler," *ECSEC Report No. ICE-VS06*, IBM European Center for Scientific and Engineering Computing, Rome, Italy, 1990.
17. W. Nagel and F. Szelényi, "A Comparison of Parallel Processing on CRAY X-MP and IBM 3090 VF Multiprocessors," *Proc. ICS'89* **4**, 271–282 (1989).

Received November 18, 1989; accepted for publication November 27, 1990

Ferenc Szelényi *IBM Germany, AN Supercomputing, P.O. Box 800880, 7000 Stuttgart 80, Germany.* Mr. Szelényi obtained his degree in computer science from Linz University, Austria, in 1987, with an emphasis on parallelizing compilers. From 1987 to 1989 he worked on his Ph.D. thesis on parallel programming at the IBM European Center for Scientific and Engineering Computing (ECSEC) in Rome, Italy. His current interest is in tools for technical computing, particularly parallel processing. In 1990 Mr. Szelényi joined IBM Germany in the Scientific and Technical Computing Department.

Vittorio Zecca *IBM Italy, European Center for Scientific and Engineering Computing (ECSEC), Via Giorgione 159, 00147 Rome, Italy.* Dr. Zecca obtained his degree in electronic engineering from Rome University in 1980. From 1982 to 1985 he worked in the aerospace field, with responsibility for data management for the San Marco project. In 1985 he joined the IBM Rome Scientific Center and was one of the initiators of the European Center for Scientific and Engineering Computing (ECSEC). For his contributions to the area of parallel processing, he has received an IBM Outstanding Technical Achievement Award. His current interest is in exploiting the features of supercomputers for scientific and industrial applications, particularly vectorization, parallel processing, and data in memory.