STRUCTRED CONTROL OF PARALLEL TRACING

Technical Report CSE-91-18

Cherri M. Pancake

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849-5347

December 16, 1991

Structured Control of Parallel Tracing

December 16, 1991

Cherri M. Pancake
(Visiting Scientist, Cornell Theory Center)
Department of Computer Science and Engineering
Auburn University, Alabama 36849
email: pancake@ducvax.auburn.edu

Abstract

Structured programming techniques have provided the scientific programmer with a way to make source code structure reflect underlying design logic. Tracing tools, however, typically provide only crude mechanisms for establishing which portions of a parallel program should be monitored and what kind of information should be reported. This report describes mechanisms which improve the situation by allowing programmers to control the number and type of run-time events in a structured fashion that reflects their conceptual models of program behavior. Although the mechanisms were designed to improve tracing facilities, they also can be applied effectively to interactive debuggers.

The advantages of the approach derive from its clear relationship to program structure. The model for specifying trace output matches that used for program code, so it is easier for the programmer to come up with a useful trace. Since events are grouped into levels according to common needs during the program life cycle, the number of unnecessary event records is reduced as well, making trace interpretation faster and less frustrating.

Index Terms: parallel programming, program traces, parallel debugging, software tools

Introduction[1]

Despite the increasing attention being given to the design and implementation of parallel debuggers (see [19, 12]), users continue to be dissatisfied [15, 13, 3]. Some of the criticisms reflect the technological difficulties of monitoring parallel execution in nonn-intrusive ways, or of reproducing behavior in an inherently unstable environment. Other complaints, however, address a more fundamental problem: providing execution information that relates meaningfully to program development activities.

Techniques for portraying parallel behavior graphically have been the focus of a number of recent research efforts [18, 20]. To date, however, little attention has been given to the problem of how debugging tools should support interaction with the user. Existing breakpoint-style debuggers (e.g., Intel's IPD [10], CONVEX's CXdb [1], or Sequent's Pdbx [23]) rely on extensions of serial debugger technology. The user manually specifies where execution should be halted or monitored, typically through breakpoints (positions in the instruction stream where processing should halt), watchpoints (data elements whose values should be monitored, with execution halting when the value is touched or if a specific condition is met), and/or tracepoints (instruction or data locations whose access should trigger generation of a message). Trace-based tools, as the term indicates, rely on just the tracepoint mechanism (e.g., the trace analysis facilities of SCHEDULE [2], GMAT [22], IBM's Parallel Fortran [8], CONVEX's CXpa [6], or Paragraph [7]); during execution, messages are logged to a trace file for real-time or post-mortem analysis. The disadvantage of this approach is that the user cannot interact with or alter program execution. On the other hand, the software hooks required to implement tracing are relatively straightforward, and

[1]A portion of this research was conducted using the Cornell National Supercomputer Facility, a resource of the Center for Theory and Simulation in Science and Engineering, which receives major funding from the National Science Foundation and IBM Corporation. The work was carried out as part of a joint development study between Auburn University and IBM Palo Alto Scientific Center.

can be inserted automatically by the compiler (e.g., CXpa) or in a preprocessing step (e.g., SCHEDULE).

Regardless of the mechanism used, the user is confronted with an all-or-nothing support situation. If monitoring is controlled by automatic instrumentation or features in the run-time library, copious amounts of data are generated, much of which may be irrelevant to the programming task at hand. In contrast, manual control over monitoring requires that the user specify where information should be gathered; this entails predicting what data will be useful and at what locations, and then either adding new statements to the program (which will need to be eliminated later) or issuing commands at run-time (which may be difficult to duplicate in a subsequent session).

This paper suggests a compromise approach, whereby the user and tool collaborate to establish an optimal level of instrumentation for a given program and tracing task. The user indicates very generally the type of information desired and the areas of the program for which trace records should be generated, by annotating the block-structured organization already present in the source code. Since the specifications are tied unambiguously to program structure, the appropriate software hooks can then be inserted by a compiler, preprocessing tool, or the debugger itself.[1] Although the techniques are described in terms of tracing tools, they could also be employed in breakpoint-style debuggers if rudimentary source-code analysis facilities were available.

The proposed strategy explots three concepts which have been largely neglected in the past, but could go a long way in making debugging tools more palatable to the user community:

- Flexible ways to limit the potentially huge amounts of data generated during execution of a scientific application.

[1]This could also be accomplished through postprocessing (e.g., application of a filter to the trace file). As the effects would be transparent, no specific implementation mechanism is described here.

- Clear correlation of dynamic/multi-stream behavior with the static/single-stream program manipulated by the user.
- Adaptation to changing requirements during the program development cycle.

The scheme is based on user-defined *event regions*, used to establish the portions of execution during which events are reported, and *event levels*, which determine the types of events to be monitored within a region. The two orthogonal controls interact to provide flexible control over monitoring. The advantages of this approach derive from its clear relationship to program structure. The model for specifying trace output matches that used for program code, so it is easier for the programmer to arrive at a useful trace. The reduction in the number of event records also makes trace interpretation faster.

The discussion begins with an analysis of the requirements for program behavior information at different points in the development cycle. This establishes the need for independent levels of trace support, outlined in the next section. The section which follows describes how the scope of tracing can be varied to fit cyclic patterns of debugging and program analysis. By way of example, the region and level mechanisms are applied to a program written in PCF Fortran [14].

Requirements for Program Behavior Information

As shown in Figure 1, the development cycle for parallel applications typically begins with a correct serial version [16]. The programmer has a general idea of which portions of the program might be performed in parallel, but it is not always clear if parallelization will be cost-effective. With the help of a profiling tool or hand-coded instrumentation, timing statistics are gathered to determine which of those areas are sufficiently compute-intensive to warrant the effort of restructuring. Parallelization then begins. As new structures are added to the program and old ones modified, the code is tested to determine if the results match those obtained from the serial

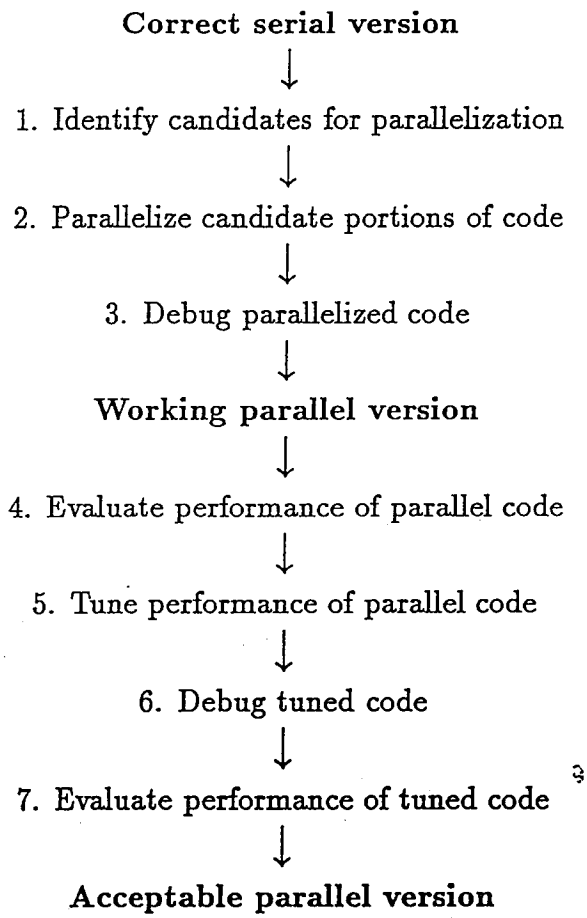

Figure 1. Development Cycle for Parallel Scientific Applications

baseline. When they diverge, a period of cyclic debugging intervenes. This alternation of testing and debugging is necessary even when software tools have been used to guide parallelization activities. Once a functional parallel version has been achieved, its performance can be tuned to maximize speedup. The tuning process often results in the discovery of additional bugs, precipitating new bouts of debugging activities. Eventually, the programmer is satisfied that further improvements are impossible or unprofitable.

Execution tracing, as a source of dynamic information on program behavior, is potentially useful at all stages in the development cycle. Although certain steps are repeated more than once (as shown in Figure 1), they may be grouped into

four categories of activities: performance profiling, debugging, benchmarking, and performance tuning.

Prior to initiating parallelization, the programmer needs a high-level *profile* of computational activities in order to determine where to focus efforts. The principal requirement here is timing information, which can be used to confirm or contradict intuitive notions of program hot-spots. As a minimum, entry to and exit from all user-supplied program units should be reported so that timing statistics can be calculated and compared.

As parallelism is introduced, run-time errors will surface. In *debugging*, the primary concern is to determine where program behavior does not match that expected by the programmer. Because the program is thought of as a sequence of manipulations on data structures, such as multi-dimensional arrays, the programmer assures correctness by tracking changes to those structures. In parallel sections of code, this activity takes on an added dimension: tracking the order in which parallel processes access the data. Not only must value changes be noted, but also the source of each change (i.e., which process made it and at what point in its activities). Determining access order often entails the analysis of synchronization events, such as which process entered a critical section last.

Once all obvious bugs have been eliminated, tracing can be used to determine the effectiveness of parallelization efforts in terms of performance. *Benchmarking* requires a finer granularity than subprogram profiling. Activities within the parallelized section of code are timed, to verify that parallelism has achieved some degree of speedup and to ascertain the possibilities for further improvement. Programmers are concerned with quantifying the execution cost or benefit of each parallelizing transformation. Moreover, they draw a distinction between the system overhead involved in starting up and terminating processs (referred to here as *system costs*) versus that incurred when processs are idle because of barrier waits, failure to obtain locks, etc. (*waiting costs*). The former represents the fixed costs associated with parallelism,

while the latter can be manipulated — at least indirectly — by the programmer.

During *tuning*, the primary concern is to identify situations which can be improved by code manipulation. The programmer needs detailed information on load balancing: the order in which work is distributed, time required to distribute shared data, time spent by each process at a barrier, etc. Since the programmer must rely on this data to fine-tune the degree of parallelism, the specifics of which work (i.e., which loop iterations or other subtasks) was assigned to each process is also important. Finally, as tuning modifications are made to the source code, additional benchmarking is needed to verify that the timings improved or to compare the effects of different tuning strategies.

Matching Trace Information to Programming Activities

A recent survey of the trace facilities available with IBM's Parallel and Clustered Fortran compilers [8, 9] revealed that users are remarkably unaware of the potential of parallel program traces [21]. Many programmers, for example, who employed traces for benchmarking or performance tuning activities had never considered using them to isolate program errors. Others underestimated their reporting capabilities, resorting to hand-coded instrumentation to acquire data already available (albeit obscured) in the trace files. This situation results in a great deal of unnecessary programmer effort and may introduce new sources of error which are extremely difficult to isolate.

The extremely large quantities of data generated for a full program trace are daunting to most programmers. In some cases, there are mechanisms available to reduce trace volume; CXpa, for example, allows selective profiling at the routine, loop, or parallel region level [6], while IBM's trace facility offers nine levels in a number of permutations [8, 9]. Users claim, however, that the mechanisms are unusable, either because they are inappropriate for the need at hand or because their use is incomprehensible or inconsistent. Moreover, the type of information reported in most

traces reflects the requirements of systems programmers, not scientific users. Much of the data reflects system factors that are irrelevant to program development, while common programming needs are left unsatisfied. Consequently, existing tools are under-utilized and under-valued by the user community.

How can the situation be improved? The first step is to organize the type of data reported in order to correspond with typical programming activities. In our block-structured approach, the type of trace records generated is controlled through *trace levels.* A level defines which execution-time events are of interest and should be reported; it therefore functions as a masking mechanism to reduce the amount of trace output. We propose five levels, reflecting the most common uses for traces:

- to establish timings for entry to and exit from subprogram units (PROFILE)
- to isolate the portion of the program where an error has occurred (DEBUG1)
- to identify the error and determine the efficacy of repairs (DEBUG2)
- to tune program performance for maximum efficiency (TUNE)
- to benchmark and compare program performance (BENCHMARK)

Normally, one level will apply to the entire program, reflecting the activity in which the programmer is engaged, be it debugging, tuning, or performance analysis. In some cases, however, it may be desirable to combine multiple levels during a single execution. The effects of each level are described in relation to typical parallel language constructs, amplified by the concept of user-defined trace messages (arbitrary text emitted in the trace file at the specification of the user).

The results of applying levels are illustrated by a brief program for the computation of π with the rectangle rule (Figure 2), written in PCF Fortran [14] and adapted from the example in [11]. The trace output shown is generalized and does not reflect any particular trace format. The columns present timestamp, process ID, source code location, and minimal messages, respectively; such information is compatible with most existing formats, as well as the suggestions for a standardized trace format summarized in [17].

```
1          PROGRAM PI
2          DO I=1,3
3              READ(*,*) NRECS
4              CALL INTEG(NRECS,RN)
5              WRITE(*,*)'Number of rectangles:',NRECS
6              WRITE(*,*)'Number of processes available:',MPRTOT
7              WRITE(*,*)'Approximation:',RN
8          END DO
9          END

10         SUBROUTINE INTEG(N,SUM)
11         GATE ADDUP GUARDS(SUM)
12         SUM = 0.0
13         UNLOCK(ADDUP)
   C parallel region and scoping declarations
14         PARALLEL
15             PRIVATE(PSUM,H,X)
   C parallel initializations (redundantly executed, once per process)
16             PSUM = 0.0
17             H = 1.0/N
   C parallel work (groups of iterations executed by each process)
18             PDO INDEX=1,N
19                 X = (INDEX-0.5)*H
20                 PSUM = PSUM + 4.0/(1.0+X*X)
21             END PDO
   C reduction executed once per process and one process at a time
22             CRITICAL SECTION (ADDUP)
23                 SUM = SUM + H*PSUM
24             END CRITICAL SECTION (ADDUP)
25         END PARALLEL
26         RETURN
27         END
```

Figure 2. Example PCF-Fortran Program

PROFILE: This level results in a minimal number of trace records (Figure 3). It is intended primarily for summarizing the amount of time spent in each program unit (main program/subroutine/function, or finer-grained blocks of code), as an indication of where parallelization or improvement efforts should be directed. The flow of program control into and out of each unit is reported in the trace file. User-defined

trace messages may identify the organization of logical activities within a unit, so these are recorded as well.

```
00000000 1   1  BEGIN PROGRAM
00000041 1  10  ENTER INTEG
00000249 1  26  EXIT INTEG
00000321 1  10  ENTER INTEG
00000549 1  26  EXIT INTEG
00000630 1  10  ENTER INTEG
00000812 1  26  EXIT INTEG
00000896 1   9  END PROGRAM
```

Figure 3. Trace Output for PROFILE Level

DEBUG1: This also results in a restricted number of trace records (Figure 4), and is particularly useful during initial attempts to localize a program error. Only events marking the very general progress — or lack of progress — of parallelism are reported. Thus, the user is able to obtain an overview of which portions of the program executed and in what general order they occurred.

```
00000000 1   1  BEGIN PROGRAM
00000041 1  10  ENTER INTEG
00000042 1  14  SHARED (SUM,ADDUP)
00000042 2  14  SHARED (SUM,ADDUP)
00000043 3  14  SHARED (SUM,ADDUP)
00000047 2  18  BEGIN PDO
00000048 1  18  BEGIN PDO
00000048 3  18  BEGIN PDO
00000202 1  21  END PDO
00000203 3  21  END PDO
00000204 3  22  WAIT CRIT SECT
00000225 2  21  END PDO
00000249 1  26  EXIT INTEG
00000321 1  10  ENTER INTEG
         ...
00000896 1   9  END PROGRAM
```

Figure 4. Trace Output for DEBUG1 Level

For parallel loops, the trace records include each process's arrival at the start and end of the construct, plus any waits caused by unsuccessful attempts to enter critical sections. Similar information is reported for parallel sections, except that waits occur due to the explicit ordering of sibling sections. User manipulation of synchronizers (such as lock and event variables) is also reported in terms of unsuccessful attempts which resulted in waits. This information gives the programmer an extremely rough idea of the extent to which contention may be affecting program behavior. Subroutine-level parallelism is also traced in terms of coarse-grained activities: the start and end of each process's work, and the satisfaction of barrier synchronization. Access to shared variables is reported only in the most general way, via lists identifying which ones were accessed by each process. Entry to and exit from subprogram (whether the invocations were serial or in parallel) continue to be traced in order to indicate the general flow of program control. User-defined trace messages are recorded as well.

DEBUG2: Like DEBUG1, this level is intended to facilitate the isolation and correction of program errors. It provides the level of detail most likely to reveal the sources of behavioral anomalies (Figure 5), but does not include performance-related information. Since DEBUG2 has the potential for generating considerable volume, it will be most useful when restricted to small portions of the program, such as those suspected (through analysis of previous DEBUG1-level output) of containing anomalies or those where code modifications have been made.

Tracing for a parallel construct reflects its progression through execution: construct entry, privatization of variables, start of each process's work, assignment of iteration groups or sections, end of each process's work, and construct exit when the barrier is satisfied. When critical section occurs, detailed information on this is reported as well, including successful and unsuccessful attempts to obtain access, as well as exit from the section. The level of detail is similar for parallel sections, except that process suspension and resumption, due to ordered execution, is reflected.

```
00000000 1   1  BEGIN PROGRAM
00000041 1  10  ENTER INTEG
00000042 1  11  GATE (ADDUP)
00000044 1  12  SHARED (SUM = 0.0)
00000044 1  13  UNLOCK (ADDUP)
00000045 1  15  PRIVATE (PSUM,H,X,INDEX)
00000047 2  18  BEGIN PDO (INDEX = 1,10)
00000048 1  18  BEGIN PDO (INDEX = 11,20)
00000048 3  18  BEGIN PDO (INDEX = 21,30)
00000176 2  18  BEGIN PDO (INDEX = 31,33)
00000202 1  21  END PDO
00000203 1  22  ENTER CRIT SECT
00000203 3  21  END PDO
00000206 1  23  SHARED (SUM = 0.7880)
00000207 3  22  WAIT CRIT SECT
00000209 1  24  EXIT CRIT SECT
00000210 3  22  ENTER CRIT SECT
00000211 1  25  WAIT BARRIER
00000214 3  23  SHARED (SUM = 2.0135)
00000217 3  24  EXIT CRIT SECT
00000218 3  25  WAIT BARRIER
00000225 2  21  END PDO
00000227 2  22  ENTER CRIT SECT
00000231 2  23  SHARED (SUM = 3.1416)
00000237 2  24  EXIT CRIT SECT
00000238 2  25  WAIT BARRIER
00000242 1  25  PASS BARRIER
00000249 1  26  EXIT INTEG
00000321 1  10  ENTER INTEG
         ...
00000896 1   9  END PROGRAM
```

Figure 5. Trace Output for DEBUG2

All user-defined synchronizer operations are now reported in the trace, whether or not a delay was involved. Thus, the creation, termination, and freeing of a lock are reported as well as attempts to gain control of it. This fine level of granularity allows the programmer to observe every transaction on synchronizers. Subroutine-level parallelism is also traced at the lowest level manipulatable by the programmer: process creation and termination, start and end of work, arrival at barriers, and

barrier satisfaction. Updates and accesses to shared data are reported in terms of the value assigned or read. Finally, subprogram entry/exit and user-defined trace messages are still recorded.

TUNE: Unlike the DEBUG levels, TUNE is intended for programs which function correctly (or appear to function correctly). This level reports on program performance (Figure 6), specifically those aspects of performance which can be tuned by the programmer to achieve maximum efficiency. Its focus, therefore, is the "variable" overhead due to poor load balancing, lock contention, etc. Information on the "fixed" costs incurred by the system during process initiation and cleanup will be reported at the BENCHMARK level.

```
00000000 1   1  BEGIN PROGRAM
00000042 1  12  SHARED (SUM,ADDUP)
00000045 1  15  PRIVATE (PSUM,H,X,INDEX)
00000047 2  18  BEGIN PDO (INDEX = 1,10)
00000048 1  18  BEGIN PDO (INDEX = 11,20)
00000048 3  18  BEGIN PDO (INDEX = 21,30)
00000176 2  18  BEGIN PDO (INDEX = 31,33)
00000202 1  21  END PDO
00000203 1  22  OBTAIN (ADDUP)
00000203 3  21  END PDO
00000207 3  22  TRY (ADDUP)
00000209 1  24  RELEASE (ADDUP)
00000210 3  22  OBTAIN (ADDUP)
00000211 1  25  WAIT BARRIER
00000217 3  24  RELEASE (ADDUP)
00000218 3  25  WAIT BARRIER
00000225 2  21  END PDO
00000227 2  22  OBTAIN (ADDUP)
00000237 2  24  RELEASE (ADDUP)
00000238 2  25  WAIT BARRIER
00000242 1  25  PASS BARRIER
00000322 1  12  SHARED (SUM,ADDUP)
         ...
00000896 1   9  END PROGRAM
```

Figure 6. Trace Output for TUNE

The events of interest for parallel loop and cases constructs include the start of the construct, start of each process's work, assignment of iteration groups or cases, termination of each process's work, and end of the construct. From this information, the programmer (or a trace analysis tool) can determine to what extent "slow" or improperly balanced processes are provoking long barrier waits. He or she can also observe the effects of attempts to tune loop/sections performance by controlling iteration groups, etc. When the construct includes synchronization constructs (critical section or ordered case execution), this is traced too, as described below for synchronizers. The record produced for subroutine-level parallelism include the start and end of each process's work, arrival at barriers, and barrier satisfaction. In addition, the distribution of shared data is reported so that the programmer can observe the delays associated with data distribution.

For user-defined synchronizers, tracing at this level reports all accesses, but not creation/termination (which cannot be tuned for efficiency). Successful and unsuccessful attempts to obtain locks, lock releases, event posting, and event waits are included. The programmer thus can observe first-hand the causes and costs of synchronizer contention. Entry to and exit from functions and subroutines are not reported at this level, but user-defined trace messages are included for the convenience of programmers who use this technique to mark or measure general program activities.

BENCHMARK: The benchmarking level is intended to provide information that will be useful in the analysis of system (as opposed to program) performance. Its events report on systems-related overhead such as process start-up time. The data will also be of interest to programmers who wish to compare the performance of alternative program versions in detail — for example, to determine where the cost breakoff point is between loop-level and subroutine-level parallelism for a particular section of code.

Tracing for parallel loop or cases constructs now reflects the system startup time incurred between entry to the construct and the initiation of process work, as well

any lag time between the arrival of the last process at the barrier and final barrier satisfaction. The full set of trace records therefore includes construct start, process creation, start of process's work, end of process's work, and each process's arrival at construct end.

```
00000000 1   1  BEGIN PROGRAM
00000042 1  12  SHARED (SUM,ADDUP)
00000044 3  14  BEGIN PARALLEL
00000045 3  15  PRIVATE (PSUM,H,X,INDEX)
00000046 1  18  BEGIN PDO
00000047 2  18  DISPATCH PDO
00000048 1  18  DISPATCH PDO
00000048 3  18  DISPATCH PDO
00000170 2  21  COMPLETE PDO
00000176 2  18  DISPATCH PDO
00000196 1  21  COMPLETE PDO
00000197 3  21  COMPLETE PDO
00000202 1  21  DONE PDO
00000202 1  22  OBTAIN (ADDUP)
00000203 1  22  ENTER CRIT SECT
00000203 3  21  DONE PDO
00000206 3  22  TRY (ADDUP)
00000207 3  22  WAIT CRIT SECT
00000208 1  24  RELEASE (ADDUP)
00000209 1  24  EXIT CRIT SECT
00000209 3  22  OBTAIN (ADDUP)
00000210 3  22  ENTER CRIT SECT
00000210 1  25  TEST BARRIER
00000211 1  25  WAIT BARRIER
         ...
00000896 1   9  END PROGRAM
```

Figure 7. Trace Output for BENCHMARK

For user-defined processes and subroutine-level parallelism, tracing records the system overhead for process management activities. These include the amount of time spent originating and terminating processes, as well as the time elapsed between arrival of the last process at a barrier and barrier satisfaction. The tracing of lock and event synchronizations is identical to that performed under TUNE, since it allows the

determination of how much system overhead time elapses between, say, the release of a lock and the re-activation of a waiting process. Again subprogram entry/exit are ignored, but any user-defined trace messages are reported.

Restricting the Scope of Analysis Information

Tracing levels alone will not reduce to manageable proportions the amount of trace data generated by scientific applications. Researchers at CONVEX, for example, found that a 10-minute program run generated 1.3 gigabytes of profiling statistics [6]. Organizing levels in terms of program development activities decreases the number of records that are extraneous to the task at hand, but it should be clear that large traces will still result.

One aspect of program development that merits closer attention in this respect is the hierarchical approach employed by most users. Empirical studies suggest that programmers "funnel in" on the code, starting with a high-level view of overall program behavior and progressively moving to more specific levels of detail [4, 5]. This procedure, which allows the programmer to put off complex issues as long as possible, mimics the top-down approach to program development. Take, for example, the way hand-coded instrumentation is added to a program to detect the source of an error (Figure 8). The programmer first investigates general behavior at the level of subprogram units. The focus is then narrowed to a particular block of code. Finally, code modification is performed at the level of individual statements. A similar procedure is followed for benchmarking and performance improvement activities. In pre-improvement benchmarking, for example, the first order of business is determining which subprogram units account for the greatest proportion of execution time. Within those units, analysis is then refined to pinpoint the areas which have the greatest potential for yielding improvements.

To support this approach, a second mechanism interacts orthogonally with the

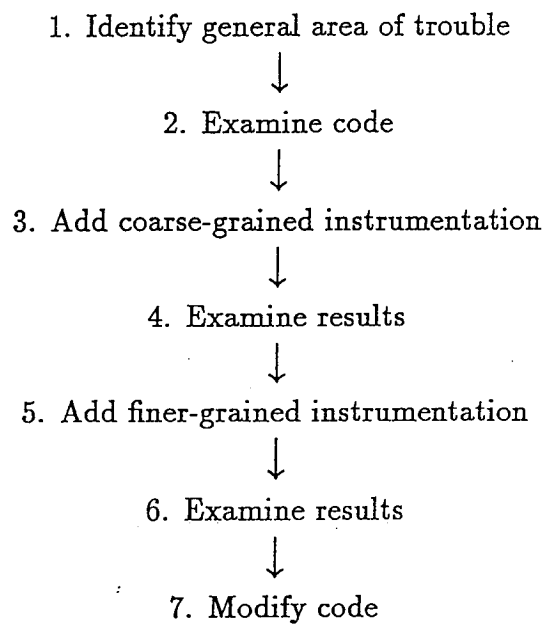

Figure 8. Hierarchical Approach in Hand-coded Debugging

trace level controls. *Trace regions* limit the scope of tracing, or the period of time during which event records are generated. Because the program already represents a block-structured expression of problem logic, it makes sense that tracing scope relate directly to source code organization. A region, therefore, corresponds to a subprogram unit (SUBPROGRAM and IGNORE controls), a block construct (CONSTRUCT), or an arbitrary area (BEGIN and END). The first three control static (lexical) scope, while the other two delimit dynamic regions. The number and nature of the regions were established through extensive interviews with scientific users [21].

Each type of region is described below. For convenience, the controls are shown in the form of compiler or preprocessor directives. It is intended, however, that regions be specified graphically through the use of a program editor or other interactive tool. Facilities for highlighting regions with shading or color will allow the user to pinpoint the areas of interest quickly and accurately. They will also emphasize the distinction between "step-over" (static) and "step-down" (dynamic) tracing of subordinate program modules.

```
      SUBROUTINE INIT(SUM,N)
      ...
      SUM = 0.0
C$ T$SUBPROGRAM
      PARALLEL DO IPLANE=1,N
         PRIVATE(PSUM)
         PSUM = INITGLX(IPLANE,PSUM)
      END PARALLEL DO
      UNLOCK(ADDUP)
      ...
      RETURN
      END
```

Figure 9. Example of SUBPROGRAM Region

SUBPROGRAM: The programmer uses this region to indicate interest in a particular subprogram or portions thereof. Tracing will be active during the execution of all statements within the region (in this case, after the occurrence of the T$SUBPROGRAM directive). Its effect is limited to the immediate static (lexical) scope; that is, tracing is deactivated at calls to subordinate functions or subroutines. For example, the region defined in Figure 9 begins in the middle of the subroutine and encompasses all subsequent statements, but does not "step down" to include the code executed by the invocation of INITGLX.

CONSTRUCT: This region provides finer granularity than SUBPROGRAM, corresponding to the execution of a program block. Block constructs include all block-structured elements in the language, but typically only parallel blocks (e.g., parallel loops and sections) are of interest for tracing. Since the programmer uses CONSTRUCT to indicate interest in a particular construct or group of constructs, its effect is limited to the immediate static scope. In the example of Figure 10, tracing begins just prior to execution of the PARALLEL DO and continues until the loop has terminated; it is deactivated during the invocation of INITGLX.

```
      SUBROUTINE INIT(SUM,N)
      ...
      SUM = 0.0
C$ T$CONSTRUCT
      PARALLEL DO IPLANE=1,N
         PRIVATE(PSUM)
         PSUM = INITGLX(IPLANE,PSUM)
      END PARALLEL DO
      UNLOCK(ADDUP)
      ...
      RETURN
      END
```

Figure 10. Example of CONSTRUCT Region

```
      SUBROUTINE INIT(SUM,N)
      ...
      SUM = 0.0
C$ T$BEGIN
      PARALLEL DO IPLANE=1,N
         PRIVATE(PSUM)
         PSUM = INITGLX(IPLANE,PSUM)
      END PARALLEL DO
      UNLOCK(ADDUP)
C$ T$END
      ...
      RETURN
      END
```

Figure 11. Example of BEGIN/END Region

BEGIN and END: The user can also define arbitrary regions that are not restricted to construct or subprogram boundaries, and that reflect the dynamic flow of program control through subprograms. A BEGIN/END region effectively toggles tracing on and off, as shown in Figure 11. Note that in this case, trace records are generated from the start of the parallel loop until after the UNLOCK operation, including during all subprograms invoked within the scope of the region (INITGLX and

```
      SUBROUTINE INIT(SUM,N)
      ...
         PSUM = INITGLX(IPLANE,PSUM)
      ...
      END
```

```
      SUBROUTINE INITGLX(I,SUM)
C$ T$IGNORE
      PARALLEL DO J=1,I
         ...
      END PARALLEL DO
      RETURN
      END
```

Figure 12. Example of IGNORE Region

any subordinates it might have). Due to the nesting of subprograms during execution, a previous BEGIN/END region may be active when a new region is encountered, although it will be more common that regions are closed for the duration of subordinate routines, as described below.

IGNORE: Because user-delimited regions transcend invocation boundaries, they have the potential for generating considerable amounts of trace data. An IGNORE region therefore offers a convenient mechanism for temporarily closing a region for the duration of a subprogram. By specifying that a subprogram should be ignored, the programmer disables all tracing at that level of invocation; tracing is resumed after return to the caller. In Figure 12, any region which was open at the calling site to INITGLX will be temporarily closed during execution of that subroutine. The scope of the IGNORE region is static, so tracing will again become active within any of its subordinate routines. The effects of this region are antithetical to those of SUBPROGRAM; where SUBPROGRAM initiates statically-scoped tracing at the indicated point and continues until the end of the subprogram unit, IGNORE disables tracing for the same area.

It is also possible to combine regions of different types. Their interaction provides a tight control over exactly which portions of code are traced. Returning to the CONSTRUCT region in Figure 10, for example, the specification of a SUBPROGRAM region containing INITGLX would have the effect of suppressing all records except those in the subroutine or in the parallel loop.

Conclusions

Structured programming techniques offer the scientific programmer ways to make source code structure reflect the underlying design logic. As a result, it has become commonplace for users to apply cyclic and hierarchical approaches in code development. Block-structured tracing capitalizes on this observation. It allows the programmer to control the number and type of run-time events in a structured fashion that reflects both source code organization and changing requirements during the program development cycle.

The cyclic and hierarchical approaches interact throughout the parallel program cycle. Most programmers develop or parallelize their applications one section at a time. A full cycle — converting code to parallel form, testing and debugging it, benchmarking the results, then fine-tuning it it to achieve the best possible performance — is applied to a subportion of the program. Once it is complete, the programmer moves on to another area, typically returning only if a latent bug emerges or if later work generates a new idea for performance improvement. This suggests that tracing tools should provide separate controls for (a) indicating the program area of current interest and (b) identifying what type of information should be reported for that area. The controls should be easy to specify and easy to change.

The orthogonal trace and region mechanisms provide direct support for this approach. Hierarchical patterns indicate that at any stage during program development, a single trace region or collection of trace regions is likely to be of interest for a length

of time. For that reason, the region mechanism is potentially fine-grained, while level provides a simple, coarser control. Cyclic patterns, on the other hand, indicate that varying collections of trace data will be desired for the region as the user progresses through different programming tasks. By organizing trace events according to typical activities, the level mechanism eliminates the tedium of discarding irrelevant records and clarifies the contribution of each record type. Together, the two controls interact to make parallel debugging tools easier and more effective for user applications.

References

[1] CONVEX Computer Corporation. 1991. *Convex CXdb User's Guide.* Convex Press, publication DSW-473.

[2] Dongarra, J. J. and D. C. Sorensen. 1987. SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs. In *The Characteristics of Parallel Algorithms*, Jamieson, *et al.*, eds. MIT Press.

[3] Fox, G. C. 1989. Parallel Computing Comes of Age: Supercomputer Level Parallel Computations at Caltech. *Concurrency: Practice & Experience*, 1 (1): 94.

[4] Gould, J. D. 1975. Some Psychological Evidence on How People Debug Computer Programs. *International Journal of Man-Machine Studies*, 7: 151–182.

[5] Gould, J. D. and P. Drongowski. 1974. An Exploratory Study of Computer Program Debugging. *Human Factors*, 16 (3): 258–277.

[6] Hansen, G. J., C. A. Linthicum and G. Brooks. 1990. Experience with a Performance Analyzer for Multithreaded Applications. *Proceedings of Supercomputing '90*, pp. 124–131.

[7] Heath, M. T. and J A. Etheridge. 1991. *Visualizing Performance of Parallel Programs.* Technical Report ORNL/TM-11813, Oak Ridge National Laboratory.

[8] IBM Corporation. 1990. *Clustered Fortran Language and Library Reference.* IBM Corporation, publication SC23-0523-0.

[9] IBM Corporation. 1988. *Parallel Fortran Language and Library Reference.* IBM Corporation, publication SC23-0431-0.

[10] Intel Supercomputer System. 1991. *iPSC/2 and iPSC/860 Interactive Parallel Debugger Manual.* Intel Corporation, publication 312043-001.

[11] Jakob, R. 1990. Parallel Programming Models for Shared Memory Multiprocessors. M.S. Thesis, Department of Electrical and Computer Engineering, University of Colorado.

[12] McDowell. C. E. and D. P. Helmbold. 1989. "Debugging Concurrent Programs." *ACM Computing Surveys*, 21 (4): 593–622.

[13] Moriarty, K. J. M. 1989. Parallel Processing of Large-Scale Applications on Powerful Multiple Processors. *International Journal of Supercomputer Applications*, 3 (1): 82–87.

[14] Parallel Computing Forum. 1990. *PCF Fortran, Version 3.1.* (Prepared for ANSI X3H5)

[15] Pancake, C. M. 1991. Software Support for Parallel Computing: Where Are We Headed? *Communications of the ACM*, 34 (11): 52–64.

[16] Pancake, C. M. and D. Bergmark. 1990. Do Parallel Languages Respond to the Needs of Scientific Programmers? *IEEE Computer*, 23 (1): 13–23.

[17] Pancake, C. M., D. Gannon, S. Utter and D. Bergmark. 1991. Supercomputing '90 BOF Session on Standardizing Parallel Trace Formats. Technical Report CTC91-TR53, Cornell Theory Center.

[18] Pancake, C. M. and S. Utter. 1991. Debugger Visualizations for Shared-Memory Multiprocessors. In *High-Performance Computing II*, ed. M. Durand and F. El Dabaghi, North-Holland, pp. 145–158.

[19] Pancake, C. M. and S. Utter. 1991. "A Bibliography of Parallel Debuggers – 1990 Edition," *ACM SIGPLAN Notices*, 26 (1): 21–37. [The bibliographic database is available in electronic form through the Cornell National Supercomputer Facility. For information, contact the author.]

[20] Pancake, C. M. and S. Utter. 1989. Models for Visualization in Parallel Debuggers. *Proceedings of Supercomputing '89*, pp. 627–636.

[21] Pancake, C. M. and S. Utter-Honig. 1991. Improving the Effectiveness of the Parallel Fortran Trace Facility. Internal report prepared for IBM Palo Alto Scientific Center.

[22] Seager, M. K. *et al.*. 1989. *Graphical Multiprocessing Analysis Tool (GMAT)*. Technical Report UCID-21345, Lawrence Livermore National Laboratory.

[23] Sequent Computer Corporation. 1986. *Pdbx Parallel Debugger for Sequent Systems*. Sequent Technical Publications.