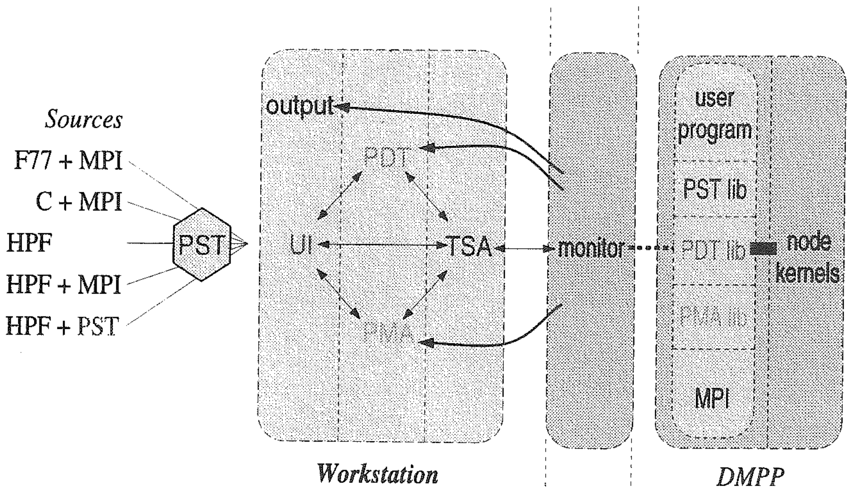
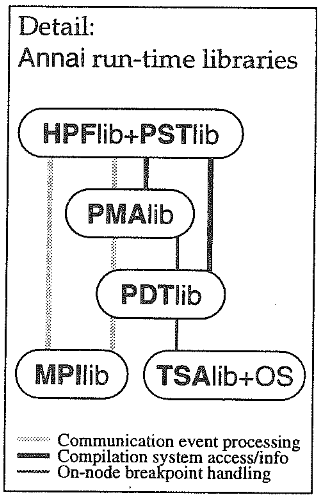
CSCS
SCSC
Centro Svizzero di Calcolo Scientifico
Swiss Center for Scientific Computing
ETH

Annai Scalable Run-time Support for Interactive Debugging and Performance Analysis of Large-scale Parallel Programs

Christian Clémençon, Akiyoshi Endo, Josef Fritscher, Andreas Müller & Brian J. N. Wylie




Detail:
Annai run-time libraries
HPFlib+PSTlib
PMAlib
PDTlib
MPIlib
TSAlib+OS
Communication event processing
Compilation system access/info
On-node breakpoint handling

TECHNICAL REPORT

TR-96-04

April 1996

Annai Scalable Run-time Support for Interactive Debugging and Performance Analysis of Large-scale Parallel Programs

Christian Clémençon[1], Akiyoshi Endo[2], Josef Fritscher[1],
Andreas Müller[1] & Brian J. N. Wylie[1]

TR-96-04, April 1996

Abstract. The Annai tool environment helps exploit distributed-memory parallel computers with High Performance Fortran and/or explicit communication, using MPI as a portable machine interface. Integration within a unified environment allows the component parallelization and compilation support, debugging and performance tools to synergetically use common facilities. Additionally, massive quantities of partitioned data and execution information, from large-scale applications on multiple processors, needs to be effectively managed and presented during program engineering. This has been achieved by scalable design and cooperative integration of tool component run-time libraries, and interactive debugging and performance analysis is demonstrated with a representative user session.

Keywords. parallel program development environment; integrated tools and run-time support libraries; distributed data management and visualization; multi-level execution performance data browsing; HPF and MPI.

To appear in series *Lecture Notes in Computer Science, Proceedings of EuroPar'96 (Lyon, France)*, Springer-Verlag, August 1996.

For further information see URL `http://www.cscs.ch/Official/Project_CSCS-NEC.html`

[1] Centro Svizzero di Calcolo Scientifico (CSCS/SCSC)
[2] NEC European Supercomputer Systems, Swiss Branch
Via Cantonale, CH-6928 Manno, Switzerland
`clemencon | endo | fritscher | mueller | wylie @cscs.ch`

Contents

List of Figures

1 Introduction

The main justification for investing development effort in producing a parallel program is usually the desire to handle large, or even huge, problems. Or to put it another way, to deal with very large amounts of data. Managing the sheer quantity of data itself is a major challenge for the program developer — and often also for the computer system as well — but when the data has to be partitioned and distributed in a manner which the computer can handle efficiently, the real challenge has only just begun. Users expect appropriate tools which will help them to take care of their data and program development needs.

Compiler technology, however, remains a long way from being able to automatically and effectively parallelize run-of-the-mill sequential programs, even those written in modern languages and ignoring the ubiquitous legacy codes. Current technology requires programmers to explicitly write parallel programs, or at least provide appropriate directives, or 'hints,' informing the compiler of potentially exploitable parallelism or the best-suited data layouts. Recently standardized examples of each of these approaches are the High-Performance Fortran (HPF) language based on directives, and Message-Passing Interface (MPI) communication libraries which are used for explicit message passing between node programs (or within libraries) expressed in sequential languages.

Whether a program has been written from scratch with parallelism in mind, or based on the parallelization of a sequential program, the challenge of managing very large amounts of distributed data remains daunting. This is especially the case during program development, when cycles of debugging and performance analysis punctuate the algorithm development process. Tools are required which efficiently present data in forms which are easily and quickly interpreted, and preferably presented in a manner which closely relates to the programmer's conception of their program and data objects. The fact that large amounts of data, and possibly large numbers of processors, are involved shouldn't interfere with this process, at least not until it becomes necessary for the programmer to consider these 'details.'

Graphical views of distributed data sets and performance data collected from assemblies of processors are natural mechanisms to reduce large quantities of information to more readily managed and easily interpreted forms. Many different views of the same or slightly different data are possible, and indeed, the effectiveness of a graphical presentation depends strongly on the correct combination of the right view with the relevant data. Interactive browsing of such views, however, quickly and conveniently locates points of interest, which stand out by being different.

Dumping large amounts of raw data from many processors, often through a sequential bottleneck such as some file-systems, and to then process this data and reduce it to a presentable form, is not only extremely slow but terribly inefficient. For graphical data reduction to be done effectively, it has to be incorporated within the run-time support system of a parallel program, and much of the reduction has to be done in parallel on the same processors where the parallel program resides. Of course, this additional processing has a cost, in terms of unavoidably intrusive effects on the user program, but since in the first instance it is essential to debug and perform basic tuning of the parallel program, the intrusion can be appropriately dealt with later.

The Annai integrated tool environment [CEF+95], developed as part of the *Joint CSCS/NEC Collaboration in Parallel Processing* [CDD+96], was designed to provide a comprehensive program development environment for distributed-memory computer systems. After briefly introducing Annai and its tool components, for which detailed accounts are separately available, attention is focussed on the run-time libraries which do much of the 'behind-the-scenes' processing, and which together make interactive high-level debugging and performance analysis possible. An example session follows which demonstrates the effective implementation of these ideas, as seen by users engineering large-scale parallel applications.

2 Annai integrated tool environment

The Annai environment offers integrated tools for parallelization, debugging, and performance monitoring and analysis, with common user and machine interfaces. Building on agreed international standards, 'high-level' extended High Performance Fortran (HPF) and 'low-level' explicit message-passing programs (based on MPI) are handled by all of the tools, supporting application flexibility and portability. Through user-driven, application-oriented development in a series of prototypes, ease-of-use for non-expert parallel programmers and scalable functionality to handle actual user (and program) requirements are refined.

PST, the ***Parallelization Support Tool***, extends the current HPF definition by providing language constructs and extensive run-time support for the parallelization of irregular computations [MR95]. These PST extensions support dynamic data distributions and run-time preprocessing of critical code sections, and are currently being investigated in conjunction with the High Performance Fortran Forum. Along with comprehensive compilation support for mixed-language program sources — such as any combination of HPF, HPF with PST extensions, Fortran/MPI, and C/MPI — applications are also (optionally) instrumented to generate information for use by the other tools.

PDT, the ***Parallel Debugging Tool***, is a conventional source-level symbolic debugger, enhanced to support different levels of abstraction [CFR95]. At the data-parallel level, PDT provides coherent graphical representations of large, distributed data-sets (both views of the data values and the data distribution itself), and control- and data-breakpoints with global break conditions. At the message-passing level, PDT assists programmers with deadlock and race detection, and deterministic execution replay.

PMA, the ***Performance Monitor and Analyzer***, exploits profile summary and trace information from interactively specified source code regions where instrumentation is inserted and data collected during the execution of a parallel program [WE96]. PMA then assists with the performance tuning and interpretation of program execution through visualization and analysis of this information. Different levels of abstraction are supported, from execution summary profiles and global views of time-varying behavior down to individual processes and analysis of communication events and memory utilization.

Figure 1 is a diagram of the interaction between different Annai tool components. PDT and PMA have a common interface to the parallel computing platform via the *Tool Services Agent* (TSA), which provides basic, low-level functions for controlling parallel program execution. An OSF/Motif *User Interface* (UI) to the run-time environment and Annai tool components primarily consists of source code listing and program structure browsers which can also be directed and annotated by the other tools. These interactive components run on Unix workstations.

Within the parallel machine user programs execute utilizing the MPI, HPF, and Annai tool libraries, where bulk information from the executing program may also be processed. Optimized implementations have been developed for the NEC Cenju-3 distributed-memory parallel computer, along with versions for Unix workstations and workstation clusters. Annai cross-development utilities allow the specialized parallel

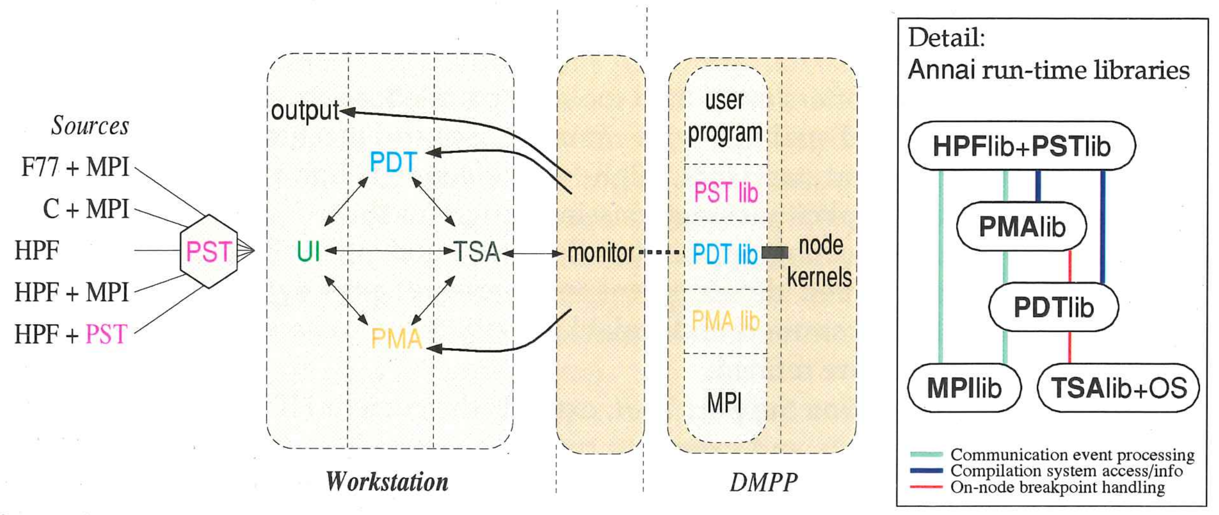


Figure 1: Annai *tool environment overview (with detail of run-time libraries)*

computer systems to be dedicated to final development, performance tuning and production use, whereas Annai provides a complete code development and debugging environment for any of these systems. Applications developed with Annai (and its run-time libraries), based on both PST/HPF and MPI, have also run readily on Intel Paragon and Cray T3D systems, verifying their portability [CEF+95].

3 Annai run-time libraries

3.1 Organization

The modular construction of the Annai interactive user environment from a number of component tools also applies to the Annai run-time system of libraries linked with the user program and running on the parallel computer system. The Annai libraries are organized as shown in the detail box of Figure 1.

The most basic configuration, corresponding to that for fully-optimized explicitly-parallelized programs without detailed debugging and performance analysis support, only uses the standard run-time and system libraries. These include the MPI communication library, optimized for the target system, which handles all of the inter-processor communication requests, buffering and routing messages as required.

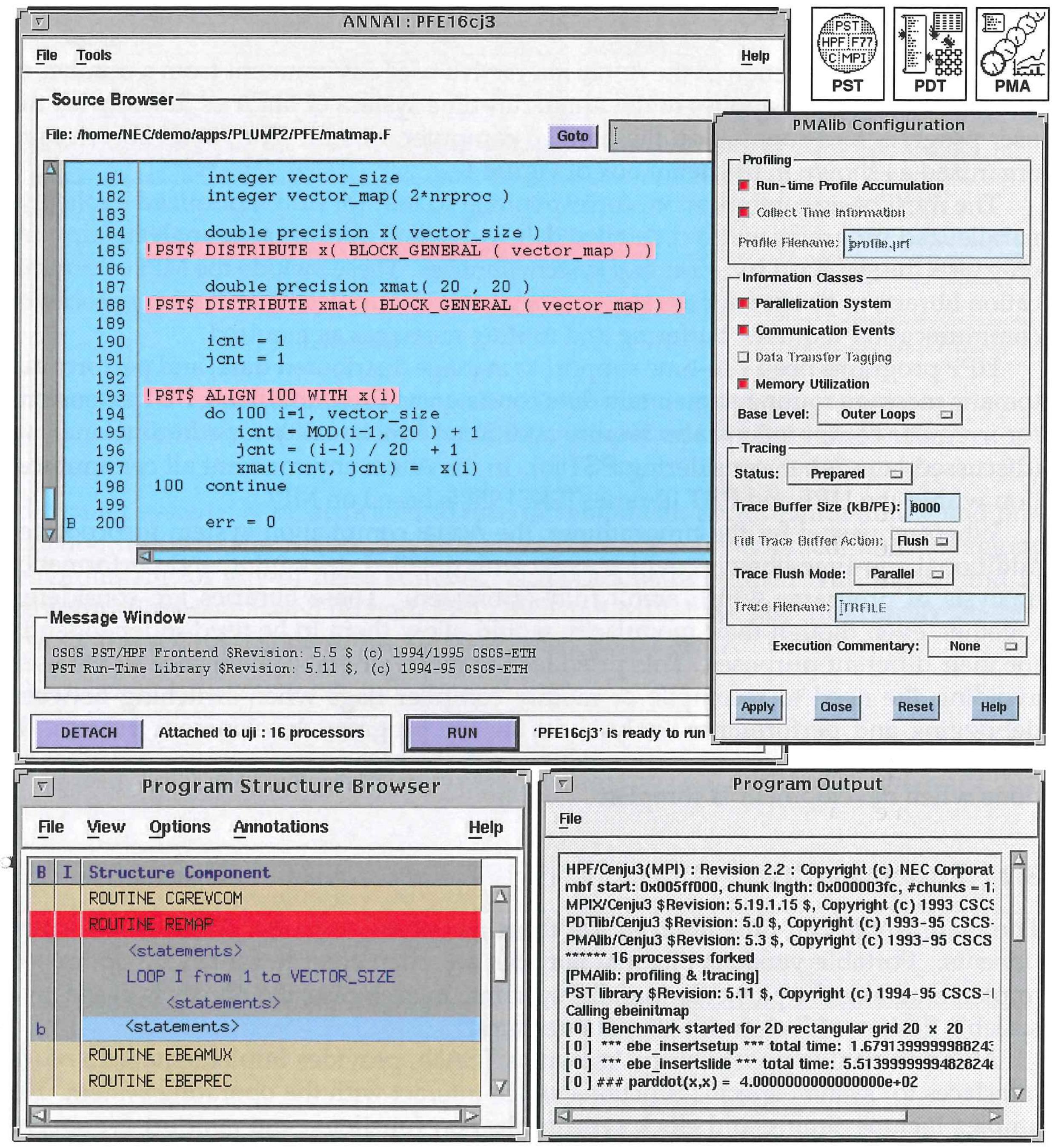
HPF programs need run-time support to manage distributed data, and perform automatic message routing to maintain data consistency. Programs using PST extensions for irregular computations also require additional functionality for run-time analysis (discussed later when considering PSTlib). In the Annai environment all communication within the HPF and PST libraries [SKS+95] is based on MPI.

When requested by the programmer, the Annai compilation system incorporates additional run-time libraries, which assist with detailed debugging and performance analysis of programs which aren't fully-optimized. These libraries are considered together, even though their modularity would allow them to be used independently for their different purposes. This provides the most general convenience to the user, avoiding the need to recompile or modify compiler flags when switching between debugging and performance analysis during the program development process. If desired for 'production' use, recompilation with the highest optimization level can be done when development is complete.

3.2 Functionality

The main Annai run-time libraries are those associated with the different tool components. Portable versions of these libraries are often complemented by optimized implementations for particular parallel systems. In each case, the library is based on a scalable design with clean functional interfaces.

An additional support library, known as TSAlib, provides implementations of (or interfaces to) system-specific functions which interact with the operating system. Examples include timing and clock synchronization functions, and support for special 'lightweight' breakpoints which are notified directly to the run-time library and processed locally on the respective processor. These breakpoints are set like normal breakpoints, but don't halt all of the processors nor interact with Annai via TSA. This avoids unnecessary synchronization and interference with other processors, providing a convenient mechanism for modifying instrumentation and performing other reconfiguration actions during execution.



File: /home/NEC/demo/apps/PLUMP2/PFE/matmap.F

```
      integer vector_size
      integer vector_map( 2*nrproc )

      double precision x( vector_size )
!PST$ DISTRIBUTE x( BLOCK_GENERAL ( vector_map ) )

      double precision xmat( 20 , 20 )
!PST$ DISTRIBUTE xmat( BLOCK_GENERAL ( vector_map ) )

      icnt = 1
      jcnt = 1

!PST$ ALIGN 100 WITH x(i)
      do 100 i=1, vector_size
         icnt = MOD(i-1, 20 ) + 1
         jcnt = (i-1) / 20  + 1
         xmat(icnt, jcnt) =  x(i)
  100 continue

B     err = 0
```

```
CSCS PST/HPF Frontend $Revision: 5.5 $ (c) 1994/1995 CSCS-ETH
PST Run-Time Library $Revision: 5.11 $, (c) 1994-95 CSCS-ETH
```

Attached to uji : 16 processors

'PFE16cj3' is ready to run

Profiling
- 🟥 Run-time Profile Accumulation
- 🟥 Collect Time Information
- Profile Filename: profile.prf

Information Classes
- 🟥 Parallelization System
- 🟥 Communication Events
- ☐ Data Transfer Tagging
- 🟥 Memory Utilization
- Base Level: Outer Loops

Tracing
- Status: Prepared
- Trace Buffer Size (kB/PE): 8000
- Full Trace Buffer Action: Flush
- Trace Flush Mode: Parallel
- Trace Filename: TRFILE

Execution Commentary: None

B	I	Structure Component
		ROUTINE CGREVCOM
		ROUTINE REMAP
		<statements>
		LOOP I from 1 to VECTOR_SIZE
		<statements>
b		<statements>
		ROUTINE EBEAMUX
		ROUTINE EBEPREC

```
HPF/Cenju3(MPI) : Revision 2.2 : Copyright (c) NEC Corporat
mbf start: 0x005ff000, chunk lngth: 0x000003fc, #chunks = 1;
MPIX/Cenju3 $Revision: 5.19.1.15 $, Copyright (c) 1993 CSCS
PDTlib/Cenju3 $Revision: 5.0 $, Copyright (c) 1993-95 CSCS-
PMAlib/Cenju3 $Revision: 5.3 $, Copyright (c) 1993-95 CSCS
****** 16 processes forked
[PMAlib: profiling & !tracing]
PST library $Revision: 5.11 $, Copyright (c) 1994-95 CSCS-I
Calling ebeinitmap
[ 0 ]  Benchmark started for 2D rectangular grid 20  x  20
[ 0 ]  *** ebe_insertsetup *** total time:  1.6791399999988243
[ 0 ]  *** ebe_insertslide *** total time:  5.51399999994828246
[ 0 ] ### parddot(x,x) =  4.0000000000000000e+02
```

Figure 2: Annai *finite-element application engineering session. The main* Annai/UI *window (top left) includes a source listing browser, which is complemented with a customizable program structure browser (lower left). Output from the previous execution of the loaded program, 'PFE16cj3,' appears in a separate window (lower right).* PDT *has already been used to set breakpoints and check distributed arrays, and* PMA *run-time library configuration is in progress.*

3.2.1 PSTlib

The core of the PST library provides run-time analysis support for PST/HPF programs, automatically managing shared and distributed variables by constructing and controlling essential message transfers (via communication patterns), and handling the execution of parallel loops. (For a thorough exposition and implementation details refer to [MR95] and references therein.)

In addition to run-time support for PST/HPF programs, the PST run-time library also provides basic support for debugging and performance analysis of such programs.

In general, PST only knows at run-time how (or even whether) an array has been partitioned, and where the different fragments are located. Support for dynamic array allocation, dynamic array re-alignment and re-distribution, assumed-size array arguments, user-defined and inherited data distributions all mean that representations of distributed arrays and symbol tables are complex structures which have to be set up at run-time. Functions are therefore provided by the PST run-time library which allow access to distributed arrays and their structure and distribution information. In this way, distributed array fragments can be reconstituted for presentation or treated as united objects by PDT.

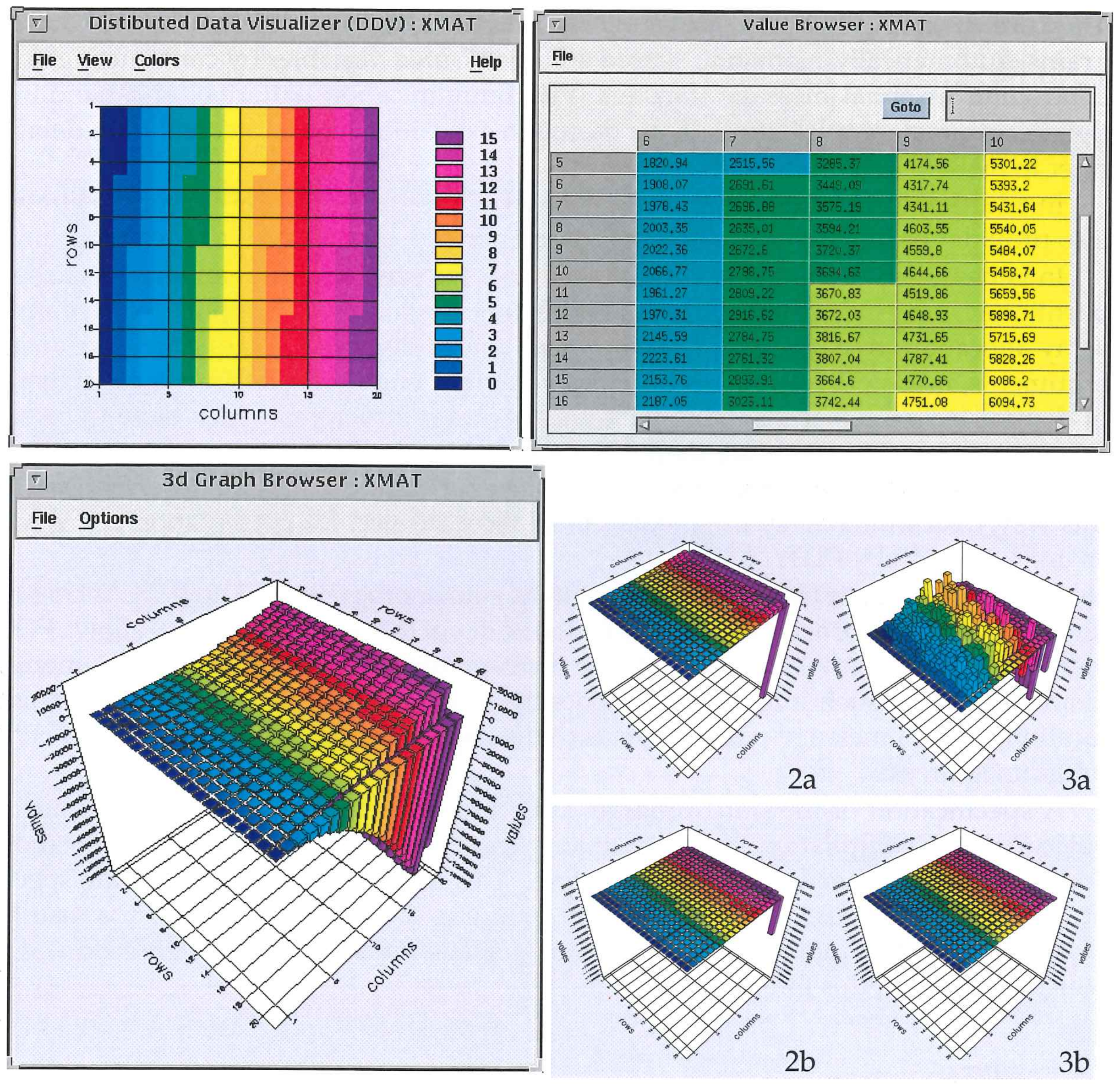
Information about the static structure of the source program, and how this relates to the transformed code actually running on the nodes of the parallel machine, allows PDT and PMA to relate low-level events to the corresponding high-level source statements. Examples where such information is essential are matching addresses of breakpoint locations or MPI message-passing events to the appropriate line (or lines) of PST/HPF program source.

A specification of the static program structure is also incorporated within source objects during compilation, and after linking constituted into a complete specification added as an extra 'section' of the executable. This mechanism ensures that the specification is always consistent with the corresponding source files, and that it is readily accessible to Annai interactive components (such as the *Program Structure Browser*, which is based on this information).

3.2.2 PDTlib

PDT is able to reconstruct a distributed array from the fragments on each processor using the array distribution information available from the PST run-time library. This 'raw' data is most conveniently processed within the parallel machine, to extract relevant parts for subsequent analysis or to reduce the quantity of data to an appropriate amount for presentation.

Data breakpoints, or 'watchpoints,' specify a break condition in terms of the program's memory state, e.g., when an array is modified or an element becomes zero, and provide an indispensable debugging service when tracking down run-time violations when they first occur. The PDT run-time library includes a scalable mechanism where all memory updates performed by the program are checked locally on each processor. Store operations are instrumented to determine whether the address is within the range of an array section to which a predicate has been specified, and then, if appropriate,



	6	7	8	9	10
5	1820.94	2515.56	3285.37	4174.56	5301.22
6	1908.07	2691.61	3449.09	4317.74	5393.2
7	1978.43	2696.88	3575.19	4341.11	5431.64
8	2003.35	2635.01	3594.21	4603.55	5540.05
9	2022.36	2672.6	3720.37	4559.8	5484.07
10	2066.77	2798.75	3694.63	4644.66	5458.74
11	1961.27	2809.22	3670.83	4519.86	5659.56
12	1970.31	2916.62	3672.03	4648.93	5898.71
13	2145.59	2784.75	3816.67	4731.65	5715.69
14	2223.61	2761.32	3807.04	4787.41	5828.26
15	2153.76	2893.91	3664.6	4770.66	6086.2
16	2187.05	3023.11	3742.44	4751.08	6094.73

Figure 3: PDT *Distributed Data Visualizer views of an array section from a PLUMP application with* PST *extended* BLOCK_GENERAL *distribution. The data map visualizer (top left) shows clearly how array elements are allocated to processors, and the same color-coding to distinguish processors is used in the other data views. To examine array elements closely, a tabular value browser is provided (top right), and this is complemented with a graphical value browser for quick overviews (bottom). In the three-dimensional graphs, the heights of the bars correspond to the data values, and these may be interactively re-oriented and rescaled for better views. The progressive reduction of the residual validates the solver, seen by comparing the main view of the initial state with those of the next two iterations: the small graphs are in pairs using the same scale as the first view and rescaled for close inspection.*

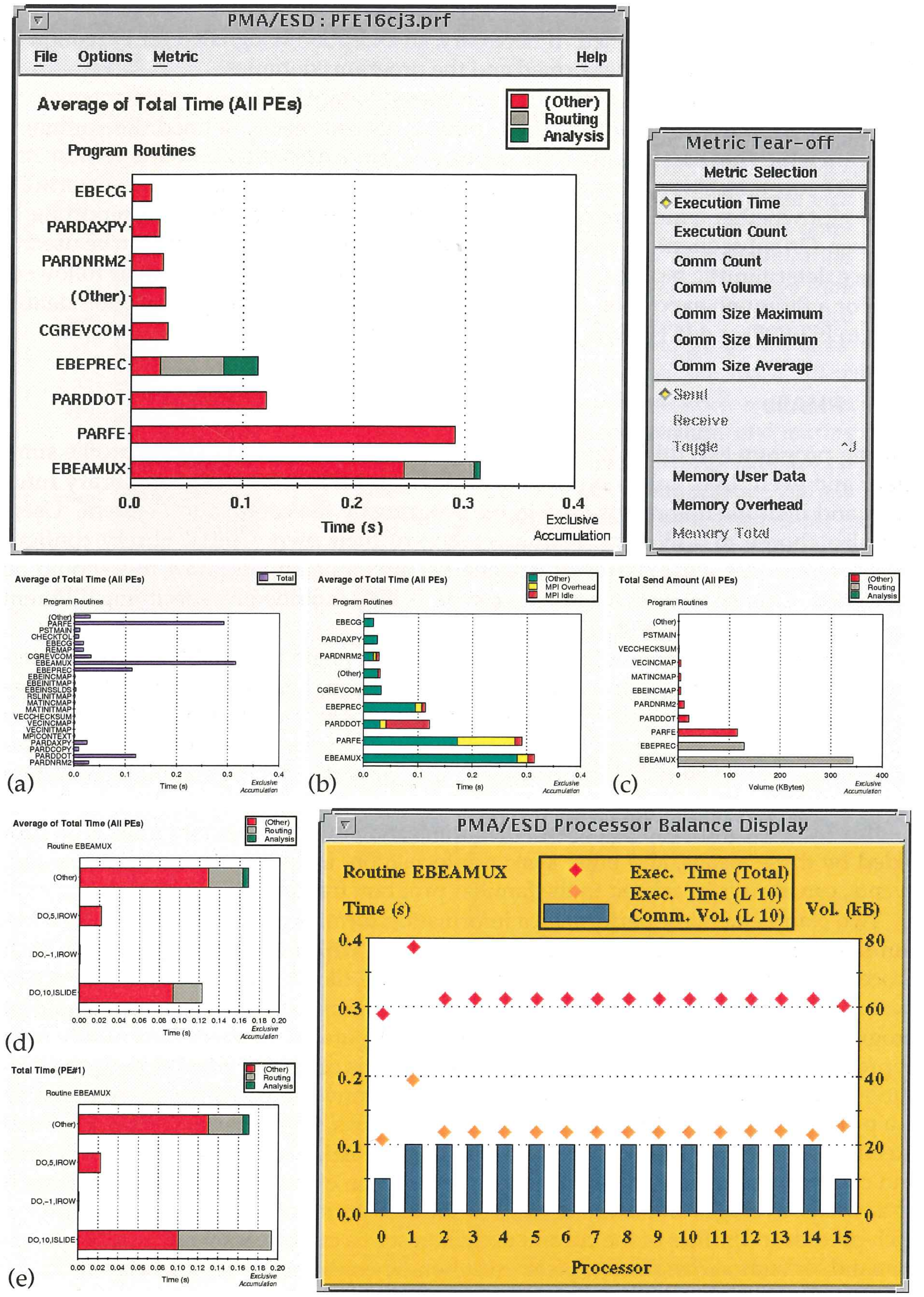


Figure 4: PMA *Execution Statistics Display (top), associated Processor Balance Display (bottom right), and assorted additional graphs produced during profile browsing.*

the value itself is also examined. If necessary, an exception is signaled for execution to halt, otherwise the store is performed and the program continues.

Messages in distributed systems are considered to race if they are simultaneously in transit and the order of receipt at a single point is not completely defined. Re-running a program which has races complicates debugging, since execution is non-deterministic. The PDT run-time library uses vector timestamps sent with each message to determine on receipt whether a race has occured, and such cases are noted for later reporting to the user. On subsequent re-execution, the trace of messages which raced can be used to enforce deterministic replay, or a choice of alternative execution paths can be followed.

(For a thorough exposition of these topics and PDT library implementation details refer to [CFR95] and [CFMR95] and references therein.)

3.2.3 PMAlib

During program execution, the PMA library is responsible for managing profile summary and event trace collection from instrumented MPI communication library functions and instrumentation inserted in the program by the compilation system. Users can also choose to add extra instrumentation of their own. Initial library configuration determines what classes of information are gathered, and how this should be processed. This 'base' instrumentation can be flexibly modified by specifying different instrumentation for selected program regions.

The state and utilization of the message-passing system is straightforwardly determined from the instrumented communication functions. Similarly, the instrumentation inserted by the compilation system makes it possible to determine the program state in terms directly related to the user's view of the program structure, i.e., routines, loops, etc. Additional information is also provided by the compilation system about parallelization overheads, such as run-time analysis, (explicit and implied) data re-distributions, and extra storage for communication buffers. Memory utilization provided by these means, and other system information, including the message-passing events, can always be related to the familiar program framework.

Two complementary performance information formats are supported by the PMA run-time library. The simplest is an event trace log which is kept in a buffer on each processor, to be processed and transfered to a trace file or directly to the interactive part of PMA within Annai on demand when the buffers become full, or at the end of program execution. Complete tracing of large-scale parallel programs is typically only appropriate (and even necessary) for detailed analysis of limited parts of program execution, such as particular routines or loops. A more convenient and scalable alternative is a profile summary, accumulated as the program runs, which keeps track of essential execution statistics. These summaries build up statistics such as the execution time and count, the number and volume of communication events, and memory utilization — as totals, averages and extreme cases — for each entity of the program structure on each processor. Profiles of the current execution status can be presented graphically on demand, or summarize the complete execution when the program finishes.

Knowledge of the processing overheads associated with an implementation of the instrumentation functions allows PMA to estimate (or, in important cases, measure)

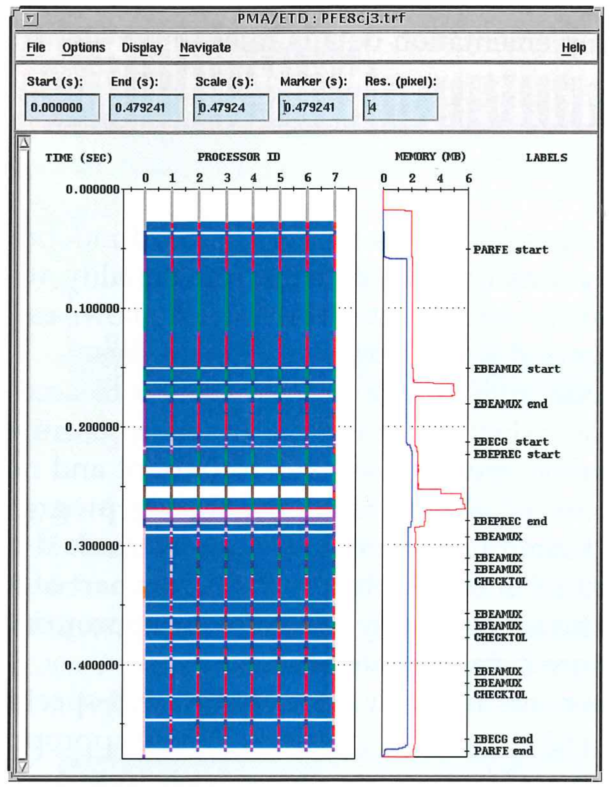
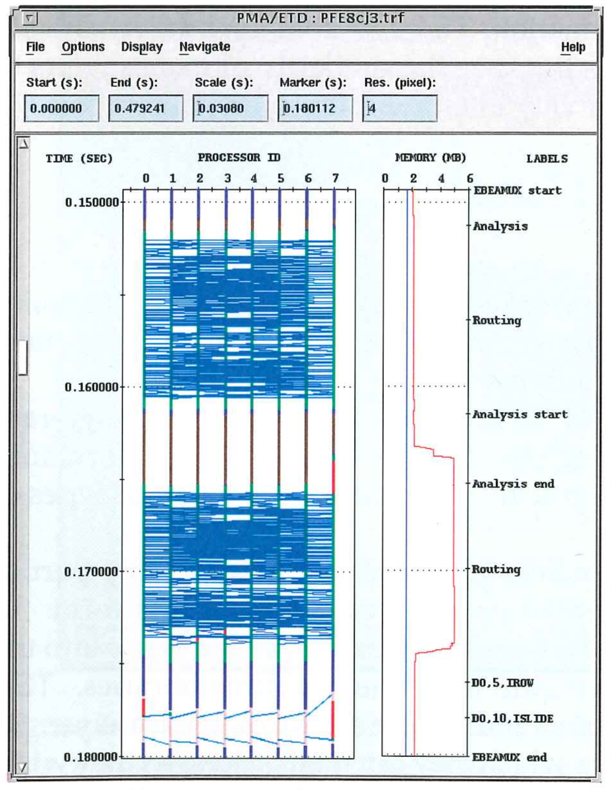
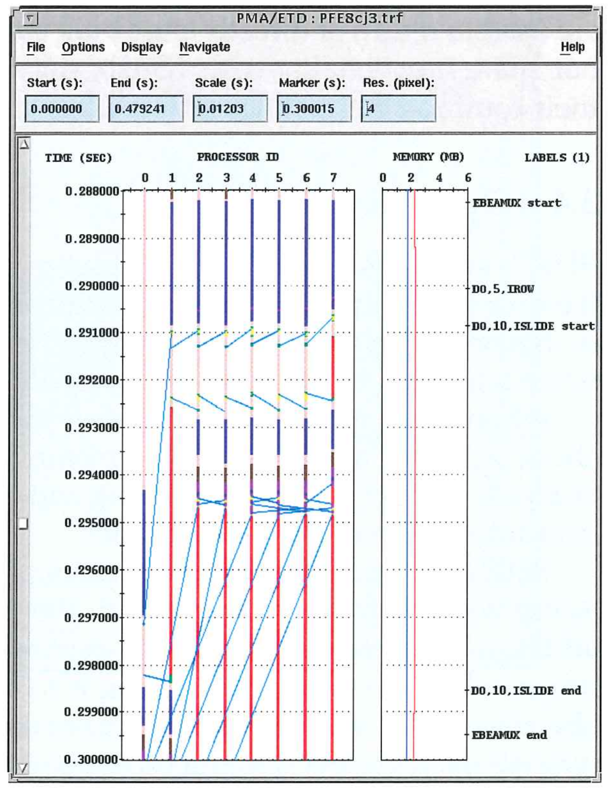


Figure 5: PMA *Evolution Time-line Display execution charts of PLUMP application processor interaction summaries and memory utilization graphs. The top-level view of the complete trace (left) highlights times where different amounts of communication occur, resolved into individual message transfers in the expanded time-scales of the two lower views. The first and second instances of routine* EBEAMUX *are shown scaled differently to completely show the whole routine execution. Costly run-time analysis and routing information exchange — requiring two phases to handle indirect array accesses, and additional working memory — dominates the first instance (bottom left), but is efficiently reused in later instances. Skew associated with PE#0 (bottom right) shows synchronization problems have arisen, however, limiting the overall effectiveness.*





program execution intrusion for presentation as part of subsequent analyses. Execution interruptions, such as breakpoints, can be presented in a similar way, or used to invalidate measurements which are directly affected by them. (For a thorough exposition of these topics and PMA library implementation details refer to [EW96] and references therein.)

3.3 Interaction

The modular design of the Annai run-time libraries allows them to be used independently of each other, but more importantly was done so that the functionality was clearly separated. Interaction is through cleanly defined interfaces, which allows each library to be separately developed and optimized for different target platforms.

During a debugging session the PDT run-time library can use PSTlib to access the relevant parts of distributed arrays, for reduction and presentation to users as single consistent objects. The array distribution itself is also often important and can be presented separately or together with array values. Similarly, during program execution message events related to the MPI communication library can be checked by PDTlib to determine whether a race has occured and/or noted by PMAlib as part of its execution record. Low-level events such as this are efficiently related to the appropriate high-level source via PSTlib program structure mapping information.

Interaction between the interactive components and their associated system-specific run-time libraries is handled by the TSA portable machine interface. Where appropriate, files written directly by the run-time libraries (such as execution trace fragments) are used to return or directly store bulk information. TSA also does not interact directly nor share functionality with TSAlib, since in this case the similarity of names refers to their common but distinct service and portability interface roles for the other tools.

3.4 Operation

PDTlib and PMAlib would have significantly intrusive effects, on each other as well as the user program, if all of their functionality were always enabled. Their functionality is, however, only *latent*, requiring appropriate configuration by Annai at run-time: when latent the intrusiveness is generally minimal.

When debugging, PDT can activate the PDTlib functionality to enable message-race checking, or to configure watched memory regions. Alternatively, during performance analysis, PMA will activate profiling and/or tracing and configure the various types of instrumentation managed by PMAlib.

Both PDT and PMA use the common facilities provided by TSA, the only part of Annai which interacts directly with the parallel program on the target platform. In addition to controlling program execution, TSA is able to read from and write into the program's address space, enabling it to configure the Annai run-time libraries. TSA also allows PDT and PMA to specify breakpoints and (where appropriate) subsequently informs the high-level tools of breakpoint hits which they can then process as they wish. At any time program execution has been halted, the run-time library configurations can be further modified by TSA: tables of memory regions corresponding to arrays

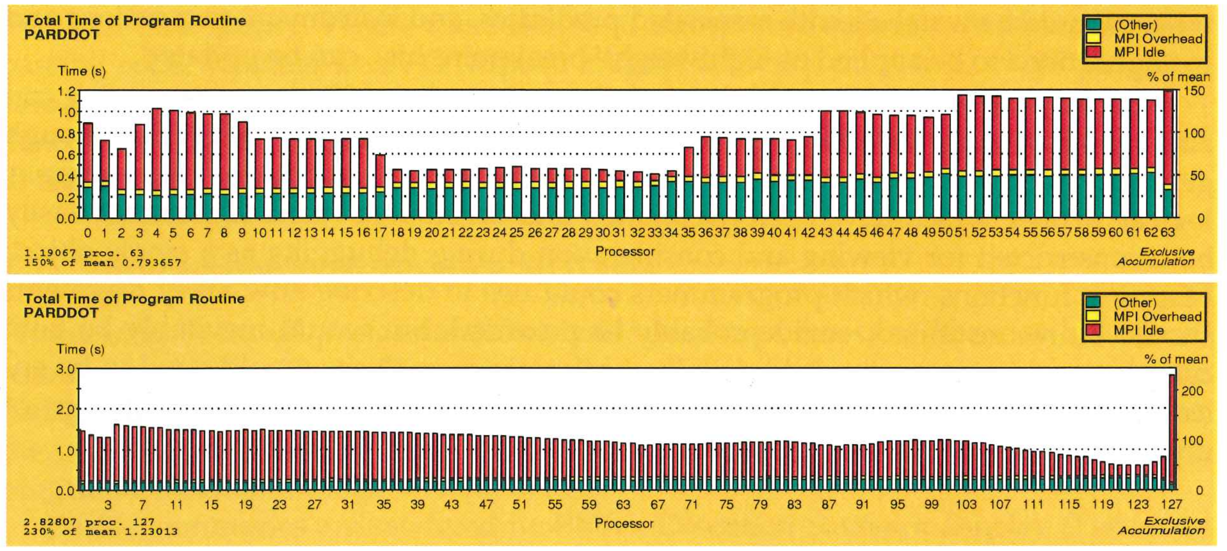
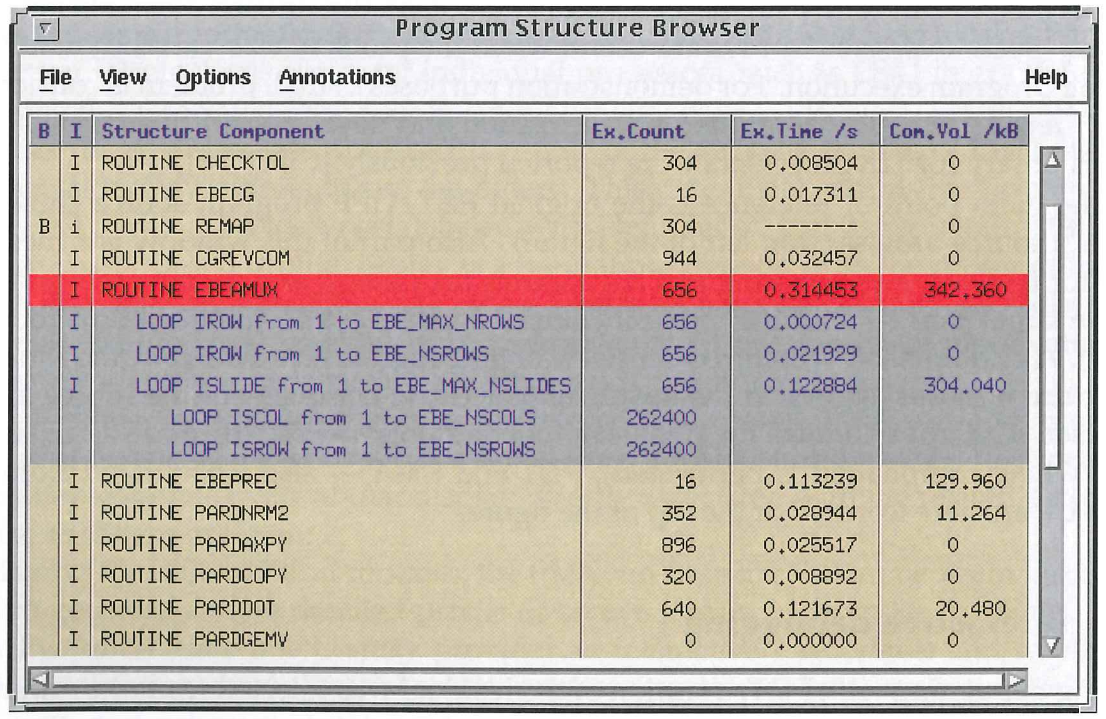


Figure 6: Annai/PMA *Processor Balance Display output graphs, showing profiles of the total execution time of routine* PARDDOT *on 64 and 128 processors, with breakdown of MPI communication overhead and idle time when blocked (red). Although computation time (green, at the bottom of each bar) is reasonably well balanced in both cases, synchronization overheads become increasingly dominant and unbalanced on larger numbers of processors leading to poor overall efficiency.*



B	I	Structure Component	Ex.Count	Ex.Time /s	Com.Vol /kB
	I	ROUTINE CHECKTOL	304	0.008504	0
	I	ROUTINE EBECG	16	0.017311	0
B	i	ROUTINE REMAP	304	--------	0
	I	ROUTINE CGREVCOM	944	0.032457	0
	I	ROUTINE EBEAMUX	656	0.314453	342.360
	I	LOOP IROW from 1 to EBE_MAX_NROWS	656	0.000724	0
	I	LOOP IROW from 1 to EBE_NSROWS	656	0.021929	0
	I	LOOP ISLIDE from 1 to EBE_MAX_NSLIDES	656	0.122884	304.040
		LOOP ISCOL from 1 to EBE_NSCOLS	262400		
		LOOP ISROW from 1 to EBE_NSROWS	262400		
	I	ROUTINE EBEPREC	16	0.113239	129.960
	I	ROUTINE PARDNRM2	352	0.028944	11.264
	I	ROUTINE PARDAXPY	896	0.025517	0
	I	ROUTINE PARDCOPY	320	0.008892	0
	I	ROUTINE PARDDOT	640	0.121673	20.480
	I	ROUTINE PARDGEMV	0	0.000000	0

Figure 7: Annai/UI *Program Structure Browser with* PMA *execution statistics annotations. This concise representation of the loaded program is synchronized with the main source browser and annotated by* PDT *and* PMA. *It also provides convenient customization facilities to fold/unfold routines or loop blocks, and to mask/filter or sort chosen entries.*

which should be watched with associated predicates, and instrumentation actions and configurations to be applied at 'lightweight' breakpoint hits, can be updated.

Programs developed partially, or entirely, with explicit MPI message-passing can also benefit from the support provided by the PDT and PMA run-time libraries, though generally not to the same extent that PST/HPF programs are supported. For example, logical 'arrays' which have been partitioned and distributed manually, cannot easily be reconstructed for viewing and consideration during debugging as a single object. Mapping functions, which programmers could use to describe how array fragments should be reconstituted, could probably be provided, but would inevitably be both complex and error prone. Additionally, performance analyses would need to be enhanced to provide selection and filtering facilities based on individual or groups of processors which have executed different code.

4 Use of Annai in parallel program engineering

Figure 2 shows a typical session with Annai, interactively debugging and analyzing the performance of an application using a Parallel Library for Unstructured Mesh Problems (PLUMP) [BLM+95] on 16 processors of a NEC Cenju-3 distributed-memory parallel computer (host-name 'uji'). The library is based on the PST `BLOCK_GENERAL` data distribution, which extends the HPF `BLOCK` distribution by allowing blocks to be specified at run-time with variable sizes and potentially unused elements between blocks. Such 'oversized' arrays are useful when the problem size varies dynamically during program execution. For demonstration purposes a small problem is considered here: details of the Annai-assisted parallelization and development of an eigensolver in this library for 128 processors were reported previously [CDD+96].

The main point of reference is the original PST/HPF program source shown in Annai's source browser (top left of the figure). Also part of this window is a message area for tool status information and the controls for loading and running the program on the target parallel machine: program output appears in a separate window (bottom right). A customizable structured overview of the program is provided by the *Program Structure Browser* (shown in the lower, left corner) to complement the source listing browser. Control windows for Annai component tools — PST (or more accurately a shell where compilation is performed), PDT and PMA — also instantiated from the main UI, are here iconized at the top of the figure.

4.1 Interactive debugging

During debugging with PDT, a section of the residual was selected from the Annai/UI source browser and analyzed with the *Distributed Data Visualizer* shown in Figure 3. From these graphical views of the residual, where color distinguishes the allocation of array elements to processors, algorithm correctness and convergence are verified.

Since the Annai/PST compilation system guarantees that HPF/PST programs are deadlock- and race-free, in this case there was no need to use that functionality of the

PDT run-time library. Where users choose to incorporate their own explicit message-passing code, whether as an `EXTRINSIC` routine, a library, or even a whole program, such debugging support can be easily activated through a PDT library configuration dialog (not shown). Similarly, PDT's memory watching functionality can be configured to check array (write) accesses for undesirable conditions.

4.2 Interactive performance analysis

Progressing to initial performance analysis, the Annai run-time libraries have been reconfigured to enable profiling and set desired classes of instrumentation (using the dialog on the right in Figure 2). This reconfiguration avoided recompiling/rebuilding the program and even the need to reload it, accumulating a profile summary during program execution for later writing to file.

Profile summaries are rendered by PMA's *Execution Statistics Display* (Figure 4), where different execution metrics can be browsed: from an initial overview of the execution time of all routines, graph (a), ranking and filtering isolates the key routines, graph (b), or analysis can quickly switch to show the communication volume for each routine, graph (c). The main view shown presents average execution time, for each program routine and with parallelization overheads distinguished: routines where distributed arrays have been analyzed at run-time and the associated message routing are found to be the most critical, particularly routine `EBEAMUX` which has then been examined in more detail. By focusing on (or 'zooming' into) this routine, graph (d) shows the same information with respect to each of the constituent loop blocks. Further restriction selects the statistics for individual processors, such as PE#1 in graph (e).

Using the associated *Processor Balance Display*, which collectively graphs selected statistics for all processors, the execution time of the `EBEAMUX` routine of PE#1 is seen to be considerably different from the others. This potentially serious inefficiency can be located to the main loop nest, labeled with identifier `10` and using iterator `ISLIDE`. This processor-based profile display is a natural progression from the program-based profile, the one complementing the other. Figure 6 demonstrates how such displays also scale well to large numbers of processors, when it becomes even more valuable for quickly identifying and isolating inefficiencies due to load imbalance.

In addition to producing graphs for reference purposes, PMA can also transfer data from its analysis displays to the Annai *Program Structure Browser*, to appear as annotation columns in its tabular display directly associated with the related program source, as shown in Figure 7.

Having identified critical routines, the PMA run-time library can be again reconfigured to generate a more detailed profile or an event trace of those key routines. The PMA *Evolution Time-line Display* provides an interactively rescalable and scrollable trace visualization chart. Vertical scales (progressing downwards) are annotated both with time and program structure information — additional 'landmark' annotations appear at higher resolution scales. Various charts of time-varying behavior can be combined, such as a processor interaction summary and a graph of memory utilization, as shown in Figure 5. Compared with the scalable statistics summary profiles, for detailed analysis a reduced number of processors and a shorter execution are favored.

5 Comparison to related work

Integrated environments for parallel systems have recently become a matter of considerable user and developer interest. Thinking Machines Corporation's commercial Prism environment [SAB+92] successfully demonstrated the principle, and motivated research groups to investigate similar support environments for Fortran D [HKTW94] and Vienna Fortran [PZ95], as well as other programming languages. Numerous more restricted efforts have also partially-integrated various editors, compilers, debuggers and analysis tools in research and commercial contexts, some now coordinated under the auspices of the Parallel Tools Consortium [Ptools] (who maintain an up-to-date list of current and former activities).

Annai is the practical realization of the further development of these ideas in a consistent, easy-to-use environment. The latest parallelization, debugging, performance monitoring and analysis techniques have been incorporated in integrated tools, providing comprehensive scalable support for flexible data-parallel and explicit message-passing application development with standardized languages and portable machine interfaces. This provision of key functionality and adherence to accepted standards are two of the criteria which came through most strongly in recent recommendations from the high-performance computing user community to those who develop and provide system software and tools for parallel and distributed computer systems [PBF95].

6 Conclusion

Annai has developed into a comprehensive environment supporting the parallelization, debugging and performance analysis of large-scale programs. Graphical displays, coupled with compilation system support (and high-level language features when necessary), provide appropriate metaphors for user interaction, but also place considerable demands on the run-time system. The design of the Annai run-time libraries, such that they can handle distributed events and process huge amounts of data — from both the user program and the system itself — in a scalable fashion, ensures support for interactive parallel program analysis.

Based on feedback from pilot users, the key usability feature is essential source reference for low-level events and distributed program objects. Also appreciated is the convenience of being able to switch between debugging and performance analysis without even needing to reload the executable, supported via dynamic configuration of the run-time libraries. The most significant benefits of the integration of individually powerful tools, however, are the complementary treatments of subtle, yet critical, parallelization aspects, and the consistent user presentations and interaction which result.

Acknowledgements Roland Rühl played an integral part in the design and implementation of Annai and its run-time libraries. NEC (Tokyo) provided the PST compiler front-end, the code generator and run-time library for standard HPF subroutines, as well as specific support for the optimized Cenju-3 run-time libraries. Additional tool development work was also performed by Jim Blandy, Maung Ting Nyeu, Emanuele La Cognata, Umesh Krishnaswamy, Mike Meehan, Andreas Gutzwiller, Paulo Lorenzo and Ramprasad Sampath during the 1993–1995 CSCS Summer Student Internship Programs. Karsten M. Decker, and the other project members, Vaibhav Deshpande, Gabrielle Jost, Norio Masuda, Will Sawyer, Eric de Sturler and Frank Zimmermann have patiently used and evaluated our prototypes.

References

[BLM+95] Ivan Beg, Wu Ling, Andreas Müller, Piotr Przybyszewski, Roland Rühl, and William Sawyer. PLUMP: Parallel Library for Unstructured Mesh Problems. In Alfonso Ferreira and Jose D. P. Rolim, editors, *Parallel Algorithms for Irregular Problems: State of the Art*, pages 45–67. Kluwer Academic Publishers, Dordrecht, Netherlands, August 1995. [ISBN: 0-7923-3623-2].

[CDD+96] Christian Clémençon, Karsten M. Decker, Vaibhav R. Deshpande, Akiyoshi Endo, Josef Fritscher, Paulo A. R. Lorenzo, Norio Masuda, Andreas Müller, Roland Rühl, William Sawyer, Brian J. N. Wylie, and Frank Zimmermann. Tool-supported development of parallel application kernels. In *Proc. 15th Int. Phoenix Conf. on Computers and Communications (Phoenix, AZ, USA)*, pages 294–302. IEEE Comp. Soc. Press, March 1996. [ISBN: 0-7803-3255-5]. Further details in CSCS-TR-95-03.

[CEF+95] Christian Clémençon, Akiyoshi Endo, Josef Fritscher, Andreas Müller, Roland Rühl, and Brian J. N. Wylie. The 'Annai' environment for portable distributed parallel programming. In *Proc. 28th Hawai'i Int. Conf. on System Sciences (HICSS-28, Vol. II)*, pages 242–251. IEEE Comp. Soc. Press, January 1995. [ISBN: 0-8186-6935-7].

[CFMR95] Christian Clémençon, Josef Fritscher, Michael J. Meehan, and Roland Rühl. An implementation of race detection and deterministic replay with MPI. In *Proc. EURO-PAR'95 (Stockholm, Sweden)*, Lecture Notes in Computer Science 966, pages 155–166. Springer-Verlag, August 1995. [ISBN: 3-540-60247-X]. Further details in CSCS-TR-95-01.

[CFR95] Christian Clémençon, Josef Fritscher, and Roland Rühl. Visualization, execution control and replay of massively parallel programs within Annai's debugging tool. In *Proc. High Performance Computing Symposium, (HPCS'95, Montréal, Canada)*, pages 393–404. Centre de recherche informatique de Montréal (CRIM), July 1995. [ISBN: 2-921316-12-9]. Further details in CSCS-TR-94-09.

[EW96] Akiyoshi Endo and Brian J. N. Wylie. Annai/PMA instrumentation intrusion presentation. In *Proc. 4th Int. Work. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'96, San Jose, CA, USA)*, pages 51–57. IEEE Comp. Soc. Press, February 1996. [ISBN: 0-8186-7235-8]. Further details in CSCS-TR-95-05.

[HKTW94] Seema Hiranandani, Ken Kennedy, Chau-Wen Tseng, and Scott Warren. The *D Editor*: A new interactive parallel programming tool. In *Proc. Supercomputing'94 (Washington DC)*. IEEE Comp. Soc. Press, November 1994.

[MR95] Andreas Müller and Roland Rühl. Extending High Performance Fortran for the support of unstructured computations. In *Proc. 9th Int. Conf. on Supercomputing (ICS'95, Barcelona, Spain)*, pages 127–136. ACM Press, July 1995. [ISBN: 0-89791-728-6]. Further details in CSCS-TR-94-08.

[PBF95] Cherri M. Pancake, Bruce Blaylock, and Robert Ferraro. Guidelines for writing system software and tools requirements for parallel and clustered computers. Technical Report 95-80-11, Oregon State University Department of Computer Science, Corvallis, OR 97331, USA, November 1995. `http://www.nero.net/~pancake/SSTguidelines/`.

[Ptools] The Parallel Tools Consortium (Ptools). `http://www-ptools.llnl.gov/`.

[PZ95] Mario Pantano and Hans P. Zima. An integrated environment for the support of automatic compilation. In Jack J. Dongarra et al., editors, *High Performance Computing: Technology, Methods and Applications*, pages 159–176. Elsevier Science B.V., 1995.

[SAB+92] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title. Data visualization and performance analysis in the *Prism* programming environment. In Nigel Topham et al., editors, *Programming Environments for Parallel Computing*, pages 37–52. North-Holland, 1992.

[SKS+95] Yoshiki Seo, Tsunehiko Kamachi, Kenji Suehiro, Masanori Tamura, Andreas Müller, and Roland Rühl. *Kemari*: a portable HPF system for distributed memory parallel machines. Tech. Rep. CSCS-TR-95-04, May 1995. Revised version to appear in *Scientific Programming*.

[WE96] Brian J. N. Wylie and Akiyoshi Endo. Annai/PMA multi-level hierarchical parallel program performance engineering. In *Proc. 1st Intl. Work. on High-Level Programming Models and Supportive Environments (HIPS'96, Honolulu, USA)*, pages 58–67. IEEE Comp. Soc. Press, April 1996. [ISBN: 0-8186-7567-5]. Further details in CSCS-TR-94-07.

1994

TR-94-06	R. GRUBER: PE2AR: Program Environments for Engineering Applications and Research. (August 1994)
TR-94-07	B. J. N. WYLIE AND A. ENDO: Design and Realization of the Annai Integrated Parallel Programming Environment Performance Monitor and Analyzer. (August 1994)
TR-94-08	A. MÜLLER AND R. RÜHL: Extending High Performance Fortran for the Support of Unstructured Computations. (November 1994)
TR-94-09	C. CLÉMENÇON, J. FRITSCHER AND R. RÜHL: Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool. (November 1994)
TR-94-10	E. GERTEISEN: Implementation of Finite Volume Fluid Solvers into the PE2AR Database Environment. (December 1994)
TR-94-11	E. GERTEISEN: A Generic Data Structure for the Communication of Arbitrary Domain Splitted Mesh Topologies. (December 1994)

1995

TR-95-01	C. CLÉMENÇON, J. FRITSCHER, M. MEEHAN, AND R. RÜHL: An Implementation of Race Detection and Deterministic Replay with MPI. (January 1995)
TR-95-02	K. DECKER AND S. FOCARDI: Technology Overview: A Report on Data Mining. (February 1994)
TR-95-03	C. CLÉMENÇON, K. DECKER, V. DESHPANDE, A. ENDO, J. FRITSCHER, N. MASUDA, A. MÜLLER, R. RÜHL, W. SAWYER, B. J. N. WYLIE, AND F. ZIMMERMANN: Tool-Supported Development of Parallel Application Kernels. (April 1995)
TR-95-04	Y. SEO, T. KAMACHI, K. SUEHIRO, M. TAMURA, A. MÜLLER, AND R. RÜHL: Kemari: a Portable HPF System for Distributed Memory Parallel Machines. (June 1995)
TR-95-05	A. ENDO AND B. J. N. WYLIE: Annai/PMA Instrumentation Intrusion Management of Parallel Program Profiling. (November 1995)
TR-95-06	P. ACKERMANN AND U. MEYER: Prototypes for Audio and Video Processing in a Scientific Visualization Environment based on the MET++ Multimedia Application Framework. (June 1995)
TR-95-07	M. GUGGISBERG, I. PONTIGGIA AND U. MEYER: Parallel Fractal Image Compression Using Iterated Function Systems. (May 1995)

1996

TR-96-01	W. P. PETERSEN: A General Implicit Splitting for Stabilizing Numerical Simulations of Langevin Equations. (February 1996)
TR-96-02	C. CLÉMENÇON, K. M. DECKER, V. R. DESHPANDE, A. ENDO, J. FRITSCHER, P. A. R. LORENZO, N. MASUDA, A. MÜLLER, R. RÜHL, W. SAWYER, B. J. N. WYLIE, AND F. ZIMMERMANN: Tools-supported HPF and MPI Parallelization of the NAS Parallel Benchmarks. (March 1996)
TR-96-03	B. J. N. WYLIE AND A. ENDO: Annai/PMA Multi-level Hierarchical Parallel Program Performance Engineering. (April 1996)

CSCS/SCSC — Via Cantonale — CH-6928 Manno — Switzerland
Tel: +41 (91) 610 8211 — Fax: +41 (91) 610 8282

CSCS/SCSC — ETH Zentrum, RZ — CH-8092 Zürich — Switzerland
Tel: +41 (1) 632 5574 — Fax: +41 (1) 632 1104

CSCS/SCSC WWW Server: http://www.cscs.ch/