A Performance Measurement Infrastructure for Co-Array Fortran

Bernd Mohr[1], Luiz DeRose[2], and Jeffrey Vetter[3]

[1] Forschungszentrum Jülich, ZAM,
Jülich, Germany
b.mohr@fz-juelich.de
[2] Cray Inc.
Mendota Heights, MN, USA
ldr@cray.com
[3] Oak Ridge National Laboratory
Oak Ridge, TN, USA
vetterjs@ornl.gov

Abstract. Co-Array Fortran is a parallel programming language for scientific applications that provides a very intuitive mechanism for communication, and especially, one-sided communication. Despite the benefits of this integration of communication primitives with the language, analyzing the performance of CAF applications is not straightforward, which is due, in part, to a lack of tools for analysis of the communication behavior of Co-Array Fortran applications. In this paper, we present an extension to the KOJAK toolkit based on a source-to-source translator that supports performance instrumentation, data collection, trace generation, and performance visualization of Co-Array Fortran applications. We illustrate this approach with a performance visualization of a Co-Array Fortran version of the Halo kernel benchmark using the VAMPIR event trace visualization tool.

1 Introduction

Co-Array Fortran (CAF) [12] extends Fortran 95 providing a simple, explicit notation for data decomposition, communication, and synchronization, expressed in a natural Fortran-like syntax. These extensions provide a straightforward and powerful paradigm for parallel programming of scientific applications based on one-sided communication. One of the problems that CAF users face is the lack of tools for analysis of the communication and synchronization behavior of the application. One of the reasons for the lack of tools is because communication operations in CAF programs are not expressed through function calls, as in MPI, or via directives that are executed by a run-time library, as in OpenMP. In contrast, CAF communication operations are integrated into the language, and, on certain platforms like the Cray X1, they are implemented via remote memory access instructions provided by the hardware.

For MPI applications, performance data collection is, in general, facilitated by the existence of the MPI profiling interface (PMPI), which is used by most MPI tools [14,7,2]. Similarly, performance measurement of OpenMP applications can be done by instrumenting the calls to the runtime library [4,1,5]. However, with the challenge of CAF

communication primitives being integrated into the language, and potentially implemented with special hardware instructions, the instrumentation of these communication primitives requires a different approach that is not straightforward.

In order to address this problem, we first defined PCAF, an interface specification of a set of routines intended to monitor all important aspects of CAF applications. Then, we extended the OPARI source-to-source instrumentation tool [10] to search for CAF constructs and to generate instrumented source code with the appropriate PCAF calls. Finally, we implemented the PCAF interface for the the KOJAK measurement system [13] enabling it to trace CAF communication and synchronization instructions. With this extension, the KOJAK measurement system is able to support performance instrumentation and performance data collection of CAF applications, generating trace files that can be analyzed with the VAMPIR event trace visualization tool [11]. In this paper, we describe our approach for performance measurement and analysis of CAF applications.

The remainder of this paper is organized as follows. In Section 2, we present an overview of Co-Array Fortran. In Section 3, we briefly describe the KOJAK performance measurement and analysis environment. In Section 4, we describe our approach for performance instrumentation and measurement of Co-Array Fortran applications. In Section 5, we discuss performance visualization with an example using the Halo kernel benchmark code. Finally, we present our conclusions in Section 6.

2 An Overview of Co-Array Fortran

Co-array Fortran [12] is a parallel programming language extension to Fortran 95. At the highest level, CAF uses a Single Program Multiple Data (SPMD) model to allow multiple copies (*images*) of a program to execute asynchronously. Each image contains its own private set of data objects. When data objects are distributed across multiple images, the array syntax of CAF uses an additional trailing subscript in square brackets to allow explicit access to remote data (as shown in Figures 2 and 4), and it is referred to as the *co-dimension*. Data references that do not use these square brackets are strictly local accesses.The CAF compiler translates these remote data accesses into underlying communication mechanisms for each target system. CAF also includes intrinsic routines to synchronize images, to return the number of images, and to return the index of the current image. Besides functions for delimiting a critical region, CAF provides four different forms of a barrier synchronization:

SYNC_ALL(): a global barrier where every image waits for every other image.
SYNC_ALL(<wait list>): a global barrier where every image waits only for the listed images.
SYNC_TEAM(<team>): a barrier where a team of images wait for every other team member.
SYNC_TEAM(<team>, <wait list>): a barrier where a team of images wait for a subgroup of the team members.

CAF was originally developed on the Cray-T3D, and, as such, it is very efficient on platforms that support one-sided messaging and fast barrier operations. On systems with globally addressable memory, such as the Cray X1 or the SGI Altix 3700, these

mechanisms may be as simple as load and store memory references. By contrast, on distributed memory systems that do not support efficient Remote Direct Memory Access (RDMA), these mechanisms can be implemented in MPI.

3 The KOJAK Measurement System

The KOJAK performance-analysis tool environment provides a complete tracing-based solution for automatic performance analysis of MPI, OpenMP, or hybrid applications running on parallel computers. KOJAK describes performance problems using a high level of abstraction in terms of execution patterns that result from an inefficient use of the underlying programming model(s). KOJAK's overall architecture is depicted in Figure 1. Tasks and components are represented as rectangles and their inputs and outputs are represented as boxes with rounded corners. The arrows illustrate the whole performance-analysis process from instrumentation to result presentation.

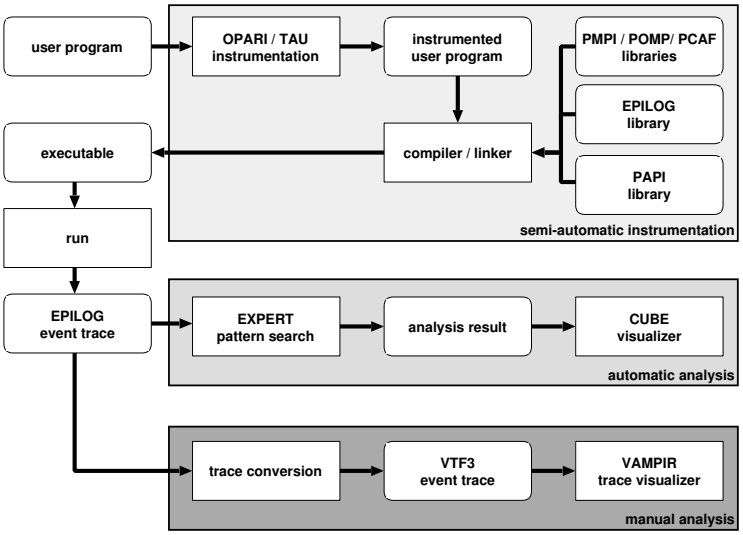


Fig. 1. KOJAK OVERALL ARCHITECTURE.

The KOJAK analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data. The first part is considered semi-automatic because it requires the user to slightly modify the makefile.

To begin the process, the user supplies the application's source code, written in either C, C++, or Fortran, to OPARI, which is a source-to-source translation tool. OPARI performs automatic instrumentation of OpenMP constructs and redirection of OpenMP-library calls to instrumented wrapper functions on the source-code level based on the POMP OpenMP monitoring API [9]. In Section 4.2, we describe how we extended OPARI for instrumentation of CAF programs with the appropriate PCAF calls.

Instrumentation of user functions is done either during compilation by a compiler-supplied instrumentation interface or on the source-code level using TAU [2]. TAU is able to automatically instrument the source code of C, C++, and Fortran programs using a preprocessor based on the PDT toolkit [8].

Instrumentation for MPI events is accomplished with a wrapper library based on the PMPI profiling interface. All MPI, OpenMP, CAF and user-function instrumentation calls the EPILOG run-time library, which provides mechanisms for buffering and trace-file creation. The application can also be linked to the PAPI library [3] for collection of hardware counter metrics as part of the trace file. At the end of the instrumentation process, the user has a fully instrumented executable.

Running this executable generates a trace file in the EPILOG format. After program termination, the trace file is fed into the EXPERT analyzer. (See [13] for details of the automatic analysis, which is outside of the scope of this paper.) In addition, the automatic analysis can be combined with a manual analysis using VAMPIR [11], which allows the user to investigate the patterns identified by EXPERT in a time-line display via a utility that converts the EPILOG trace file into the VAMPIR VTF3 format.

4 Performance Instrumentation and Measurement Approach

In this section, we describe the event model that we use to describe the behavior of CAF applications, and the approach we take to instrument CAF programs and to collect the necessary measurement data.

4.1 An Event Model of CAF

KOJAK uses an event-based approach to analyze parallel programs. A stream or trace of events allow to describe the dynamic behavior of an application over time. If necessary, execution statistics can be calculated from that trace. The events represent all the important points in the execution of the program. Our CAF event model is based on KOJAK's basic model for one-sided communication [6]. We extended KOJAK's existing set of events, which cover describing the begin and end of user functions and MPI and OpenMP related activities, with the following events for representing the execution of CAF programs:

- Begin and end of CAF synchronization primitives
- Begin and end of remote read and write operations

For each of these events, we collect a time stamp and location. For CAF synchronization functions, we also record which function was entered or exited. For the barrier routines we also collect the group of images which participate in the barrier and the group of images waited for, if applicable. Finally, for reads and writes, we collect the amount of data which is transferred (i.e., the number of array elements) as well as the source or destination of the transfer.

The event model is also the basis for the instrumentation and measurement. The events and their attributes specify which elements of CAF programs need to be instrumented and which data has to be collected.

4.2 Performance Instrumentation

Instrumentation of CAF programs can be done on either of two levels depending on how CAF is implemented on a specific computing platform. On systems where CAF constructs and API calls are translated into calls to a run-time library, these calls could easily be instrumented by traditional techniques (e.g., linking a pre-instrumented run-time library or instrumenting the calls with a binary instrumentation tool). However, for systems like the Cray X1, where the CAF communication is executed via hardware instructions, this approach is not possible. Therefore, we extended OPARI, KOJAK's source-to-source translation tool, to also locate and instrument all CAF constructs of a program.

As Fortran is line-oriented, it is possible for OPARI to read a program line by line. Of course, it is also necessary to take continuation lines into account. Then, each line is scanned for occurrences of CAF constructs and synchronization calls (but ignoring comments and contents of strings). CAF constructs can be located by looking for pairs of brackets ([...]). The first word of the statement determines whether it is a declaration line or a statement containing a remote read or write operation. For CAF declarations, OPARI collects attributes like array dimensions, and lower and upper bounds for later use.

The handling of statements containing remote memory operations is more complex. First, all operations are located in the line. If it is an assignment statement and the operation appears before the assignment operator, it is a write operation. In all other cases it is a read. OPARI determines which CAF array is referenced by the operation, the number of elements transferred (by parsing the index specification), and the source or destination of the transfer (determined by the expression inside the brackets). Simple assignment statements containing a single remote memory operation are instrumented by inserting calls to the corresponding PCAF monitoring functions before and after the statement, which get passed in the attributes determined by OPARI. In case of more complex statements where a remote memory operation cannot be easily separated out and wrapped by the measurement calls, or when it is necessary to keep instrumentation overhead low, OPARI uses the single call version of the PCAF remote memory access monitoring functions (instead of separate begin and end calls) and inserts them either before (for reads) or after (for writes) the statement for each identified remote memory access operation.

Finally, OPARI scans the line for calls to CAF synchronization routines, and replaces them by calls to PCAF wrapper functions that will execute the original call in addition to collecting all important attributes.

Figure 2(b) shows the instrumented source code generated for the example in Figure 2(a). In this example, there is a two-dimensional array `A`, which is distributed on all processors. In the CAF statement `A(me,::2)[left] = me`, each processor updates the odd entries of the row corresponding to its image in the left neighbor array with its index, and then waits on a barrier. OPARI identifies the CAF statement, and adds a begin and end instrumentation event. The call to indicate the beginning of the event contains the destination of the write (normalized to the range 0 to `num_images()-1`) and the number of array elements being transferred; the end call only gets passed the

```
integer :: me, num, left
integer :: A(1024,1024)[*]
. . .
me = this_image()
num = num_images()
left = me - 1
if ( left < 1 ) left = num



A(me,::2)[left] = me

call sync_all()

(a)
```

```
integer :: me, num, left
integer :: A(1024,1024)[*]
. . .
me = this_image()
num = num_images()
left = me - 1
if ( left < 1 ) left = num
call PCAF_rma_write_begin(-1+left, &
    1 * max((ubound(A,2)-         &
             lbound(A,2)+2)/2,0))
A(me,::2)[left] = me
call PCAF_rma_write_end(-1+left)
call PCAF_sync_all()

(b)
```

Fig. 2. (a) Example of a CAF source code and (b) OPARI instrumented version

destination. The barrier call (`sync_all`) is translated into a call of the corresponding *wrapper* function.

4.3 Performance Measurement

Finally, the KOJAK measurement system was extended by implementing the necessary PCAF monitoring functions and wrapper routines and adding support for the handling of the new remote memory access event types. We chose to implement our approach within the KOJAK framework, as KOJAK is very portable and supports all major HPC computing platforms. Also, this way, we could re-use many of KOJAK's features like event trace buffer management, generation, and conversion. Finally, it allows us not only to analyze plain CAF applications but also hybrid programs using any combination of MPI, OpenMP, and CAF. A separate, new instrumentor just for CAF would probably be problematic in this respect, as the modifications done by two independent source-to-source preprocessors could conflict.

The PCAF interface is shown in Figure 3. Since this monitoring API is open, and OPARI is a stand-alone tool, other performance analysis projects could use this infrastructure to also support CAF. For example, it would be very easy to implement a version of the PCAF monitoring library which (instead of tracing) just collects basic statistics (number of RMA transfers, amount of data transferred) for each participating image. Ideally, in the future, CAF compilers could support this interface directly.

5 Performance Visualization

For illustration of our performance analysis approach, we ran the Halo kernel benchmark on the Cray X1 system at the Oak Ridge National Laboratory, using 16 and 64 processors. The Halo benchmark simulates a halo border exchange with the four different synchronization methods CAF provides (see Section 2). The exchange procedure is

Remote Memory Access Monitoring Routines

```
  SUBROUTINE PCAF_rma_write_begin(dest, nelem)
  SUBROUTINE PCAF_rma_write_end(dest)
  SUBROUTINE PCAF_rma_write(dest, nelem)
  SUBROUTINE PCAF_rma_read_begin(src, nelem)
  SUBROUTINE PCAF_rma_read_end(src)
  SUBROUTINE PCAF_rma_read(src, nelem)
where  INTEGER, INTENT(IN) :: dest, src, nelem
```

CAF **Synchronization Wrapper Routines**

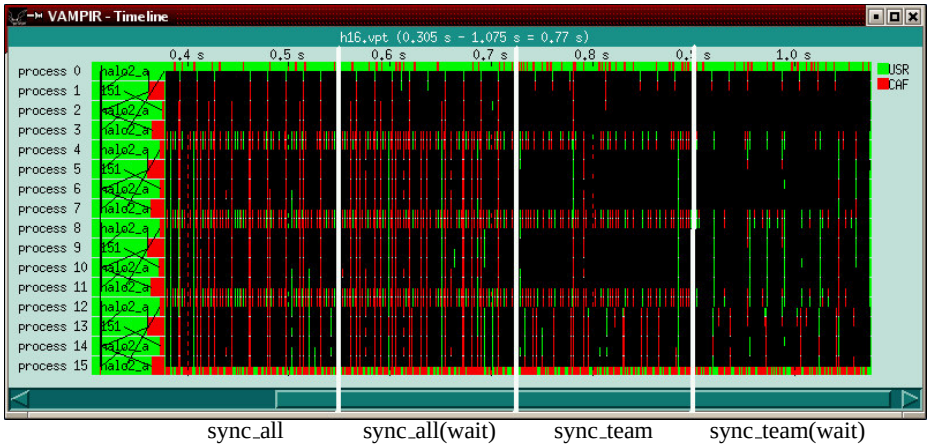
```
  SUBROUTINE PCAF_sync_all()
  SUBROUTINE PCAF_sync_all(wait)
  SUBROUTINE PCAF_sync_team(team)
  SUBROUTINE PCAF_sync_team(team, wait)
  SUBROUTINE PCAF_sync_file(unit)
  SUBROUTINE PCAF_sync_memory()
  SUBROUTINE PCAF_start_critical()
  SUBROUTINE PCAF_end_critical()
where  INTEGER, INTENT(IN) :: unit
       INTEGER, INTENT(IN) :: wait(:), team(:)
```

Fig. 3. PCAF Measurement Function Interface Specification

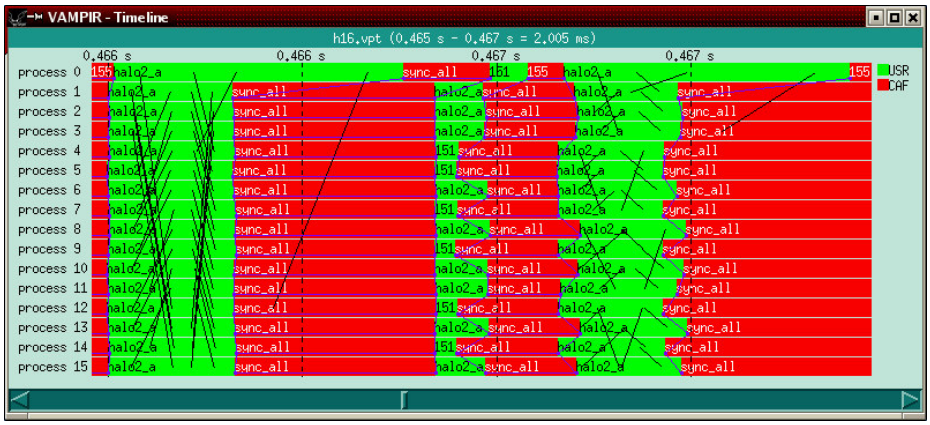
outlined in Figure 4. During each iteration, the following events from our event model occur: S2 a synchronization call; S3 a remote read of n elements from the north neighbor; S4 a remote read of $2n$ elements from the south neighbor; S5 a synchronization call; S7 another synchronization call; S8 a remote read of n elements from the west neighbor; S9 a remote read of $2n$ elements from the east neighbor; and finally S10 a synchronization call. For each synchronization method, this procedure is repeated 5 times per iteration, with 10 iterations being executed with n varying from 2 to 1024 in powers of 2.

```
S1          HINS(1:3*n)   = HOEW(1:3*n)
S2          CALL synchronization method
S3          HONS(1:n)     = HINS(1:n)[MYPEN]
S4          HONS(n+1:3*n) = HINS(n+1:3*n)[MYPES]
S5          CALL synchronization method
S6          HIEW(1:3*n)   = HONS(1:3*n)
S7          CALL synchronization method
S8          HOEW(1:n)     = HIEW(1:n)[MYPEW]
S9          HOEW(n+1:3*n) = HIEW(n+1:3*n)[MYPEE]
S10         CALL synchronization method
```

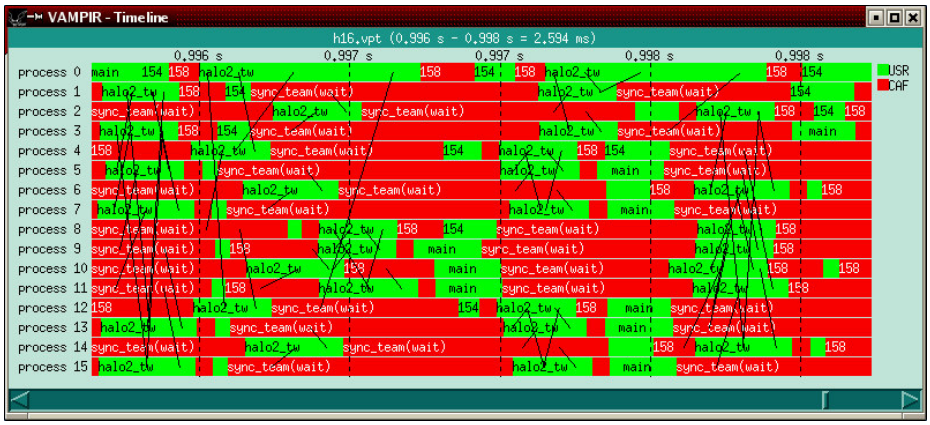
Fig. 4. Pseudo-code for the halo exchange procedure



(a) Complete program



(b) One exchange using `sync_all`

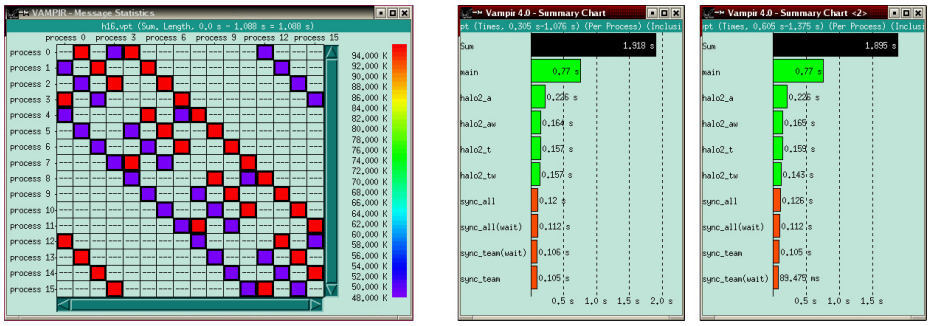


(c) One exchange using `sync_team(wait)`

Fig. 5. Timeline views of the Halo benchmark using 16 processors

Figure 5 (a) shows the timeline view of the Halo benchmark running with 16 processors. The four phases of the code (marked with white lines in the figure) can easily be identified due to the different communication behavior of each of the synchronization methods. The communication pattern between processors, as well as the amount of data exchanged, can be observed with the pair-wise communication statistics view, shown in Figure 6 (left).

Figure 5 (b) and Figure 5 (c) show a section of the timeline corresponding to a full exchange (one call to the subroutine outlined in Figure 4) for `sync_all` and `sync_team(wait)` synchronization methods respectively. We observe that the region corresponding to the `sync_team(wait)` synchronization method is much more irregular (unsynchronized) than the one for the `sync_all`, where the waiting times are longer, due to the global synchronization.



(16 processor run) (64 processor run)

Fig. 6. Message statistics view of the Halo benchmark using 16 processors (left) and Summary Chart View of Function times running on 16 and 64 processors (right)

Finally, on Figure 6 (right), we observe the time spent on each synchronization method for the 16 and 64 processors runs respectively. We notice that with the increase of number of processors, the `sync_team(wait)` method performs significantly better than the `sync_all` method, going from about 10% faster with 16 processors to about 30% faster with 64 processors.

6 Conclusion

The CAF parallel programming language extends Fortran 95 providing a simple technique for accessing and managing distributed data objects. This language-level abstraction hides much of the complexity of managing communication, but, unfortunately, this also makes diagnosing performance problems much more difficult. In this paper, we have proposed one approach to solve this problem. Our solution uses a source-to-source translator to allow performance instrumentation, data collection, trace generation, and performance visualization of Co-Array Fortran applications implemented as an extension of the KOJAK performance analysis toolset. We illustrated this approach with performance visualization of a Co-Array Fortran version of the Halo kernel benchmark

using the VAMPIR event trace visualization tool. Our initial results are promising; we can obtain statistical quantification and graphical presentation of CAF communication and synchronization characteristics. We will extend KOJAK's automated analysis to also cover CAF constructs and determine the benefits of this approach for real applications.

References

1. E. Ayguadé, M. Brorsson, H. Brunst, H.-C. Hoppe, S. Karlsson, X. Martorell, W. E. Nagel, F. Schlimbach, G. Utrera, and M. Winkler. OpenMP Performance Analysis Approach in the INTONE Project. In *Proceedings of the Third European Workshop on OpenMP - EWOMP'01*, September 2001.
2. R. Bell, A. D. Malony, and S. Shende. A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proceedings of Euro-Par 2003*, pages 17–26, 2003.
3. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.
4. J. Caubet, J. Gimenez, J. Labarta, L. DeRose, and J. Vetter. A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications. In *Proceedings of the Workshop on OpenMP Applications and Tools - WOMPAT 2001*, pages 53 – 67, July 2001.
5. L. DeRose, B. Mohr, and S. Seelam. Profiling and Tracing OpenMP Applications with POMP Based Monitoring Libraries. In *Proceedings of Euro-Par 2004*, pages 39–46, September 2004.
6. Marc-André Hermanns, Bernd Mohr, and Felix Wolf. Event-based Measurement and Analysis of One-sided Communication. In *Proceedings of Euro-Par 2005*, September 2005.
7. S. Kim, B. Kuhn, M. Voss, H.-C. Hoppe, and W. Nagel. VGV: Supporting Performance Analysis of Object-Oriented Mixed MPI/OpenMP Parallel Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.
8. K. A. Lindlan, Janice Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, and C. Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In *Proceedings of Supercomputing 2000*, November 2000.
9. B. Mohr, A. Mallony, H.-C. Hoppe, F. Schlimbach, G. Haab, and S. Shah. A Performance Monitoring Interface for OpenMP. In *Proceedings of the fourth European Workshop on OpenMP - EWOMP'02*, September 2002.
10. Bernd Mohr, Allen Malony, Sameer Shende, and Felix Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, 2002.
11. W. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. Vampir: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, January 1996.
12. R. W. Numrich and J. K. Reid. Co-Array Fortran for Parallel Programming. *ACM Fortran Forum*, 17(2), 1998.
13. Felix Wolf and Bernd Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing'*, 49(10–11):421–439, November 2003.
14. C. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proceedings of Supercomputing 2000*, November 2000.