FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Initial Design of a Test Suite for Automatic Performance Analysis Tools - APART Technical Report -

*Bernd Mohr, Jesper Larsson Träff**

FZJ-ZAM-IB-2002-13

Oktober 2002

(letzte Änderung: 23.10.2002)

(*) C&C Research Laboratories, NEC Europe Ltd.

Initial Design of a Test Suite for Automatic Performance Analysis Tools

APART[1] Technical Report

http://www.fz-juelich.de/apart

Work Package 2

Common Interfaces and Integration Techniques

Bernd Mohr
Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich
b.mohr@fz-juelich.de

Jesper Larsson Träff
C&C Research Laboratories
NEC Europe Ltd.
traff@ccrl-nece.de

[1] The IST Working Group on *Automatic Performance Analysis: Real Tools* is funded under Contract No. IST-2000-28077

Abstract

Automatic performance tools must of course be tested as to whether they perform their task correctly. Because performance tools are meta-programs, tool testing is more complex than ordinary program testing, and comprises at least three aspects. First, it must be ensured that the tools do neither alter the semantics nor distort the runtime behavior of the application under investigation. Next, it must be verified that the tools collect the correct performance data as required by their specification. Finally, it must be checked that the tools indeed perform their intended tasks: For badly performing applications, relevant performance problems must be detected and reported, and, on the other hand, tools should not diagnose performance problems for well-tuned programs without such problems. In short, performance tools should be *semantics-preserving*, *complete* and *correct*. Focusing on the correctness aspect, testing can be done by using *synthetic test functions* with controllable performance properties, and/or real world applications with known performance behavior. A systematic *test suite* can be built from such functions and other components, possibly with the help of tools to assist the user in putting the pieces together into executable test programs.

Clearly, such a test suite can be highly useful to builders of performance analysis tools, such as the members of the APART working group, who currently pursue a variety of tool development projects. It is surprising that till now, no systematic effort has been undertaken to provide such a suite. The APART members have designed, implemented, and tested performance analysis tools for many years, and have solid experience in collaborative work in the group. This makes APART a natural forum for the design and implementation of such a test suite.

In this report we discuss the initial design of a test suite for checking the correctness (in the above sense) of automatic performance analysis tools. In particular, we describe a framework which allows to easily construct both simple and more complex synthetic test programs which contain desired performance properties. It is planned to implement this framework over the next year. The outcome will be made available to tool developers both within and outside of the APART group.

Contents

Chapter 1

Introduction

Automatic performance analysis tools, like any other tool, must be tested to ensure as far as possible that they perform their intended tasks. It is of critical importance that they work correctly, because their users rely on them to evaluate other programs. In addition, in contrast to regular, non-automatic performance analysis and visualization tools, which are mainly used by experts, the target audience for automatic tools are novices and non-expert users, who may not have the experience to recognize if a tool does not work correctly.

It is surprising that till now, no systematic effort has been undertaken to provide a test suite to check the correctness of performance analysis tools. Such a suite would immediately benefit all developers of automatic performance analysis tools. Currently, members of the APART working group are developing prototypes of automatic performance analysis tools [1, 2, 3, 4, 5, 6]. They also have solid experience in collaborative work in the group. This makes APART a natural forum for the design of such a test suite.

In this report we discuss the initial design of a test suite, which we from now on will simply term ATS (for APART Test Suite), for checking the correctness of automatic performance analysis tools. It summarizes the outcome of several APART Work Package 2 meetings organized over the last year. In particular, we describe a framework which allows to easily construct both simple and more complex synthetic test programs which contain desired performance properties. It is planned to implement this framework over the next year. The outcome will be made available to tool developers both within and outside of the APART group.

What does testing of performance tools mean? The following list summarizes the aspects which need to be considered in such a test suite and our proposal on how to solve them:

Semantics-preserving: Tests to determine whether the semantics of the original program were not altered
⇒ Can be partially based on existing validation suites

Completeness: Tests to see whether the recorded performance data are correct and complete (as required by the specification of the tool)
⇒ This is tool specific and cannot be provided by a generic test suite

Positive correctness: Positive synthetic test cases for each known and defined performance property and combinations of them
⇒ Design and implementation of an ATS framework

Negative correctness: Negative synthetic test cases which have no known performance problem
⇒ Can also be covered by the ATS framework

Scalability / Applicability: Real-world-size parallel applications and benchmarks
⇒ Collect available benchmarks and applications and document their performance behavior

What does that mean in detail? First, performance tools must be *semantics-preserving* in the sense that they do not change the semantics of the program being analyzed. To a large part, this can be checked by using existing validation suites for parallel programming paradigms like MPI or OpenMP. These suites can be instrumented by the performance analysis tool, and should still report no errors. Perhaps more importantly, performance tools must also be *non-intrusive* in the sense that they do not distort the relative timing behavior of the application being analyzed. The ATS framework will not address this item per se, other than by providing a WWW collection of resources and links to parallel programming API validation suites. In Chapter 2, we present a first collection of resources.

Each tool collects a specific set of performance data which it uses to analyze a user's program. It needs to be tested whether the right data are collected, e.g., for an MPI performance analysis tool it must be ensured that the correct sender and receiver ranks, message tags, and communicator IDs for a message are recorded. Performance tool testing should address this aspect, but this task is largely specific to each individual tool, and cannot be expected to benefit much from a general purpose test suite. So, this aspect will not be a covered by the ATS.

For the application programmer the correctness aspect is of course the most important issue. Here, the tool must find relevant performance problems in ill-behaving applications, but should not detect spurious problems in well-tuned programs. Here, a general-purpose test suite can be developed.

During the first phase of the APART working group, ASL, a specification language for describing performance properties was developed [7]. A *performance property* characterizes a specific type of performance behavior which may be present in a program. Performance properties have a *severity* associated with them, the magnitude of which specifies the importance of the property in terms of its contribution to limiting the performance of the program. The ASL report also includes a catalog of typical performance properties for MPI, OpenMP and HPF programs. For a hybrid MPI/OpenMP programming style, especially with the Hitachi SR-8000 in mind, this catalog was extended in [8]. Similar performance properties can be found formalized in [1, 2] and [4, 5]. These typical properties can form the basis for the ATS framework which allows to construct synthetic positive and negative test programs. The initial design for this framework is presented in Chapter 3.

Finally, is has to be ensured that a performance analysis tool not only works for carefully constructed "simple" test cases but also for large, complex applications in use on today's computer systems. Here we propose to collect pointers to publicly available application programs together with a standardized description including installation instructions, descriptions of the application's performance behavior and already available performance data. This is described in Chapter 4. Finally, future work is listed in Chapter 5.

Chapter 2

Benchmark and Validation Suites

As already mentioned, existing benchmark and validation suites of parallel programming APIs like MPI and OpenMP can be used to check whether a performance tool does not change the semantics of the program being analyzed (at least for scientific programs which typically are based on these parallel programming paradigms).

The procedure is simple: First, the test suite is executed on the target system. Second, if everything works as expected, the validation suite is executed again, but this time with instrumentation added by the performance analysis tool. The result of both runs must be the same. Of course, the guarantee that such a test can give is only as good as the validation suite used, and as always, even the most extensive testing can never ensure that a tool is indeed correct for all possible uses.

Benchmark suites sometimes also perform correctness checks, and can therefore be useful for the correctness testing of an automatic performance analysis tool as just described. They also can be used to give an idea of how much the instrumentation added by a tool affects performance, i.e., of the overhead introduced by the tool. Again the procedure is to run the benchmark suite without and with the tool instrumentation and compare the outcome. Note that this is not a thorough approach to the problem of evaluating the *intrusiveness* of a tool in the sense of its effect on (relative) timing behavior of the program being analyzed.

In the following, we list an initial set of pointers to benchmark and validation suites which was collected by members of the APART group and which is the starting point for a more extensive collection in the next years.

2.1 MPI Validation Suites

- MPICH test suite, Argonne National Laboratory
  `ftp://ftp.mcs.anl.gov/pub/mpi/mpi-test/mpich-test.tar.gz`

- MPI test suite, IBM
  `http://www-unix.mcs.anl.gov/mpi/mpi-test/ibmsuite.html`

- MPICH version of the IBM test suite, Argonne National Laboratory and IBM
  `ftp://ftp.mcs.anl.gov/pub/mpi/mpi-test/mpichibm.tar`

- Comprehensive test suite for MPI 1.1, Intel
  `ftp://ftp.mcs.anl.gov/pub/mpi/mpi-test/intel-mpitest.tgz`

- MPICH version of the Intel test suite, Argonne National Laboratory and Intel
  `ftp://ftp.mcs.anl.gov/pub/mpi/mpi-test/intel-mpitest-patched.tgz`

- List of changes of Sun they needed to make to the Intel test suite
  `ftp://ftp.mcs.anl.gov/pub/mpi/mpi-test/sun-intel-patch.txt`

2.2 MPI Benchmark Suites

- PARKBENCH (PARallel Kernels and BENCHmarks)
  `http://www.netlib.org/parkbench/`

- PMB (Pallas MPI Benchmarks)
  `http://www.pallas.com/e/products/pmb/`
- SKaMPI (Special Karlsruher MPI–Benchmark)
  `http://liinwww.ira.uka.de/~skampi/`

2.3 PVM Validation Suites

- PVM test suite
  `http://www.epm.ornl.gov/pvm/tester.html`
- Grindstone: A Test Suite for Parallel Performance Tools (9 PVM programs)
  `http://www.cs.umd.edu/~hollings/papers/grindstone.html`
  `ftp://ftp.cs.umd.edu/pub/faculty/hollings/software/grindstone.tar.gz`

2.4 OpenMP Validation Suites

To the best of our knowledge there are no OpenMP validation suites yet.

2.5 OpenMP Benchmarks Suites

- EPCC OpenMP Microbenchmarks
  `http://www.epcc.ed.ac.uk/research/openmpbench/openmp_index.html`

2.6 Hybrid (MPI / OpenMP) Benchmarks Suites

- LAMB (Los Alamos MicroBenchmarks suite)
  Supports MPI and multi-threading (Pthreads and OpenMP) programming models
  based on SKaMPI and EPCC suites
  `http://www.c3.lanl.gov/par_arch/CODES/LAMB/lamb.html`

Chapter 3

The APART Test Suite Framework

The main purpose of a performance tool test suite is to check that the tools indeed perform their intended tasks: For badly performing applications, relevant performance problems must be detected and reported, and, on the other hand, tools should not diagnose performance problems for well-tuned programs without such problems. Based on the specification and collection of performance properties done by the APART working group, we propose to implement the ATS (*APART Test Suite*) framework which allows the easy construction of synthetic positive and negative test programs.

3.1 Basic Structure of the ATS Framework

The main goal of our design is a collection of modules that can be easily combined to produce a program exhibiting some desired performance property. Thus, modules should have as little context as possible. Furthermore, automatic performance tools have different thresholds/sensitivities. Therefore it is important that the test suite is parametrized so that the relative severity of the properties can be controlled by the user.

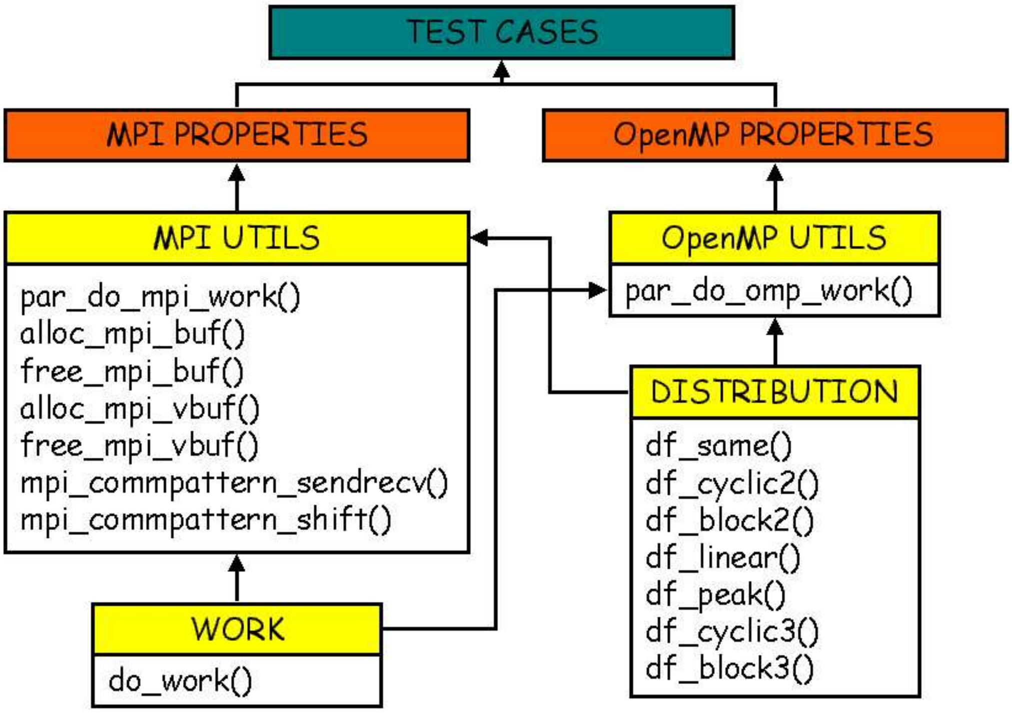


Figure 3.1: Basic Structure of the ATS Framework.

Figure 3.1 shows the basic structure of the ATS framework. The boxes with shaded titles represent the basic modules. Arrows indicate the *used-by* relationship. For the lower levels, a list of functions provided by the module is also shown.

The lowest two modules provide basic functionality, namely functions to specify the amount of generic work to be executed by the individual threads or processes of a parallel program. The next level provides generic support for the two main parallel programming paradigms MPI and OpenMP. The third level implements the so-called *property functions*, i.e., functions which when executed show one specific performance property. Finally, there are several ways of calling the property functions so that different aspects of performance correctness testing can be addressed. Our initial design only covers the “standard” parallel programming paradigms MPI and OpenMP. However, the modular structure of the design easily allows to add modules for other programming APIs (e.g., HPF, PVM, or POSIX threads). In the following, we describe the modules in more detail.

3.1.1 Specification of Work

On the highest level, applications consist of two parts: the actual desired user computation (*work*) and the necessary extra communication, synchronization, and organization overhead for executing the program in parallel. Therefore, for a generic test program we first need a way to specify a generic piece of work.

There are several possibilities to do this: The work could be specified by the time it takes to execute the desired computation. Another way would be to specify the amount and type of instructions to execute (e.g., how many floating-point, integer, load, and store operations), or the amount of other resources, e.g., memory, cache misses etc. The main problem is to implement this in a portable way, so the same results can be achieved on different computer systems. Further research is necessary in this area.

Specification of (sequential) work

```
void do_work(double secs);
```

Our current ATS prototype just provides a very simple function `do_work` which allows to specify the desired execution time. The real time is only approximated up to a certain degree (approx. milliseconds) and is not guaranteed to be stable especially under heavy work load. The goal was to represent real time without actually calling time measuring functions because these are typically implemented as system calls and therefore are not really reliable and potentially expensive. This means that the provided function cannot be used to validate time measurements. Of course, future versions of ATS need more elaborate versions of work functions.

Our current implementation uses a loop of random read and write accesses to elements of two arrays. Through the use of random access and the relatively large size of the arrays, the execution time should not be influenced by the cache behavior of the underlying processor. In a configuration phase during installation of the ATS framework, the number of iterations of this loop which represent one second is calculated through the use of calibration programs. This value is then used to generate the desired execution time during run time.

Specification of parallel work

```
void par_do_mpi_work(distr_func_t df, distr_t* dd, double sf, MPI_Comm c);

void par_do_omp_work(distr_func_t df, distr_t* dd, double sf);
```

The generic specification of parallel work is based on two ideas:

1. The participants are specified explicitly through an MPI communicator or implicitly by the active OpenMP thread team.

2. The distribution of work among the participants is described through generic distribution parameters (distribution function pointer `df`, distribution descriptor `dd`, and scale factor `sf`) which are described in the next section.

The implementation of these functions is very simple. The functions are supposed to be called by all participants of a parallel construct, very much like a collective communication call in MPI. First, the number of participants in the parallel construct and the rank/ID of the participant within this group is determined. Then, the amount of work for the calling participant is calculated using the distribution parameters. Finally, the sequential version of `do_work()` is called with the calculated amount of work. As an example, the implementation of `par_do_mpi_work()` is shown below:

Example: par_do_mpi_work()

```
void par_do_mpi_work(distr_func_t df, distr_t* dd, double sf, MPI_Comm c)
{
  int me, sz;
  MPI_Comm_rank(c, &me);
  MPI_Comm_size(c, &sz);
  do_work(df(me, sz, sf, dd));
}
```

Finally, an interesting implementation detail: our initial version of `do_work()` used the UNIX random generator `rand()`. However, this presented a problem when we implemented the parallel work function for OpenMP. As the version of `rand()` provided by the operating system was made thread-safe by using locks for accessing the random seed, the parallel version was implicitly serialized. Therefore, our current implementation uses our own simple (but efficient, while lock-free) parallel random generator.

3.1.2 Specification of Distribution

One major factor for the performance of parallel programs is a well balanced distribution of work and of communication to the participating threads and processes. Therefore, a second useful functionality for a generic test suite is to provide a way to specify distributions. As already described in the last section, we use the combination of a *distribution function pointer* (specifying the type of the distribution) and a *distribution descriptor* (providing the specific parameters to the distribution function) to specify a generic distribution.

Distribution function type

```
typedef double (*distr_func_t)(int me, int sz, double sf, distr_t* dd);
```

An ATS generic distribution function is passed by pointer and has the following four parameters: The first two parameters (`me`, `sz`) specify the rank/ID of the participant and the size of the parallel group for which the distribution should be calculated. A scaling factor `sf` allows the result to be scaled by a proportional factor. Finally, the last parameter `dd` describes the parameters of the desired distribution. To allow a generic implementation, this is done by passing a pointer to a C struct containing the necessary fields.

Predefined ATS distribution descriptors

```
typedef struct {
  double val;
} val1_distr_t;
```

```
typedef struct {
  double low;
  double high;
} val2_distr_t;
```

```
typedef struct {
  double low;
  double high;
  int n;
} val2_n_distr_t;
```

```
typedef struct {
  double low;
  double high;
  double med;
} val3_distr_t;
```

As shown above, ATS provides four predefined data descriptor types which allow to describe distributions with one to three double and integer parameters. Below, the current set of predefined distribution functions are listed:

Predefined ATS distribution functions

```
/* -- SAME Distribution: everyone gets the same value -- */
double df_same(int me, int sz, double scale, distr_t* dd);

/* -- CYCLIC2 Distribution: alternate between low and high -- */
double df_cyclic2(int me, int sz, double scale, distr_t* dd);

/* -- BLOCK2 Distribution: two blocks of low and high respectively -- */
double df_block2(int me, int sz, double scale, distr_t* dd);

/* -- LINEAR Distribution: linear extrapolation between low and high -- */
double df_linear(int me, int sz, double scale, distr_t* dd);

/* -- PEAK Distribution: task n -> high, all other low -- */
double df_peak(int me, int sz, double scale, distr_t* dd);

/* -- CYCLIC3 Distribution: alternate between low, med, and high -- */
double df_cyclic3(int me, int sz, double scale, distr_t* dd);

/* -- BLOCK3 Distribution: three blocks of low, med, and high -- */
double df_block3(int me, int sz, double scale, distr_t* dd);
```

Examples demonstrating the usage of distribution functions can be found in Section 3.1.3 and 3.1.5. If necessary, users can provide their own distribution functions and distribution descriptors, as long as their implementation follows the same basic principles and the signature of the distribution function is equivalent to `distr_func_t`.

3.1.3 MPI Buffer Management

In MPI, all communication is via exchange of data stored in buffers in memory controlled by the application process. To simplify this process for the test suite, components for maintaining buffers were defined.

An MPI buffer is described by a structure containing the address of the actual buffer, the number of data items that can be stored in the buffer (`count`), and their MPI data type. For most purposes, simple MPI types like integers (`MPI_INT`) and doubles (`MPI_DOUBLE`) will be sufficient, but MPI provides the possibility to work with arbitrarily complex, structured and possibly non-contiguous data, so the data type argument is needed to represent an MPI buffer. The C type `mpi_buf_t` represents an MPI buffer.

MPI simple buffer management

```
typedef struct {
  void* buf;
  MPI_Datatype type;
  int cnt;
} mpi_buf_t;

mpi_buf_t* alloc_mpi_buf(MPI_Datatype type, int cnt);

void free_mpi_buf(mpi_buf_t* buf);
```

The collective operations in MPI come in two flavors. A regular version where all participating processes specify the same amount of data, and an irregular version, where each communicating process pair may specify an individual amount of data. Since we also want to provide for irregular collective operations in the test suite, a more general buffer type is needed for this case providing additional fields for storing the characteristics of the buffer allocated for the root rank.

MPI collective buffer management

```
typedef struct {
  void* buf;
  void* rootbuf;
  MPI_Datatype type;
  int  cnt;
  int* rootcnt;
  int* rootdispl;
  int isroot;
} mpi_vbuf_t;

mpi_vbuf_t* alloc_mpi_vbuf(MPI_Datatype type, distr_func_t df,
                           distr_t* dd, int root, MPI_Comm c);

void free_mpi_vbuf(mpi_vbuf_t* buf);
```

To manage MPI buffers two constructor functions `alloc_mpi_buf()` and `alloc_mpi_vbuf()` are provided. For the irregular case, additional arguments use a distribution function (as in the parallel work functions) to describe the distribution of data over the processes. For the regular buffer, only MPI type and element count are given as arguments. In addition, there are destructor functions for safely releasing the buffers (`free_mpi_buf()` and `free_mpi_vbuf()`).

MPI “default” buffer size

```
void set_base_comm(MPI_Datatype type, int cnt);
```

Finally, there is a function for setting the default buffer size for MPI communication used in the MPI property test programs, which is specified by the desired number of elements of a specific MPI data type.

3.1.4 MPI Communication Patterns

For use in the construction of the actual property functions we have tried to extract some readily usable communication patterns. The two patterns described in this section are just a starting point, more work is needed in this direction. The patterns are called by all processes in a communicator, much like a collective operation. Parameters are, for each process, a send and a receive buffer, the MPI communicator, and additional control parameters. Important in the design is that patterns can be called with little context; as long as the communication buffers match pairwise, a pattern should work (that is, not deadlock or abort) regardless of the number of processors, or of other communication going on at the same time. Below are the prototypes for two *point-to-point communication patterns*:

MPI communication patterns

```
typedef enum MPI_DIR { DIR_UP, DIR_DOWN } mpi_dir_t;

void mpi_commpattern_sendrecv(mpi_buf_t* buf, mpi_dir_t dir,
                              int use_isend, int use_irecv, MPI_Comm c);

void mpi_commpattern_shift(mpi_buf_t* sbuf, mpi_buf_t* rbuf,
                           mpi_dir_t dir, int use_isend,
                           int use_irecv, MPI_Comm c);
```

The `mpi_commpattern_sendrecv` pattern performs an even-odd send-receive: processes with even ranks send to a process with an odd rank. The direction parameter `dir`, which must be the same for all calling processors, determines the direction of the communication (up or down). If called with an odd number of processes, the last process is ignored and will not take part in the communication. By the `use_isend` and `use_irecv` parameters an MPI communication mode can be selected: if set, non-blocking (immediate) send-receive operations are used, followed by an `MPI_Wait` operation to complete the communication.

The `mpi_commpattern_shift` pattern performs a cyclic shift. In this case, all processes are sending and receiving a message. The direction parameter, which again must be the same for all calling processors, determines the direction of the shift.

3.1.5 Performance Property Functions

Using the described functions for specifying work and distribution, as well as for MPI buffer management and communication, it is now quite simple to implement functions which, when executed, show a well-defined performance behavior. Ideally, for each of the performance properties listed in [7], at least one function should be defined. Therefore, we call them *performance property functions*.

Of course, these functions should be implemented as generic as possible, i.e., there should be no restrictions on the context where the functions are called (e.g., the number of processors). Also, there should be parameters to describe the exact form and the severity of the performance property.

A large portion of the performance property functions is related to an imbalance of work or communication. In this case, the property functions typically have three parameters: Through a distribution function and its corresponding distribution descriptor the imbalance of work or communication is specified. A third parameter describes how often the main body of the performance property function should be repeated. As an example, the complete source code of the performance property function `imbalance_at_omp_barrier()` is shown below:

Example: imbalance_at_omp_barrier()

```
void imbalance_at_omp_barrier(distr_func_t df, distr_t* dd, int r) {
  int i;

  #pragma omp parallel private(i)
  {
    for (i=0; i<r; ++i) {
      par_do_omp_work(df, dd, 1.0);
      #pragma omp barrier
    }
  }
}
```

Other performance property functions are based on more complex event patterns and therefore have more performance property specific parameters. As an example here we show the `late_sender()` function:

Example: late_sender()

```
void late_sender(double basework, double extrawork, int r, MPI_Comm c) {
  val2_distr_t dd;
  int i;
  mpi_buf_t* buf = alloc_mpi_buf(base_type, base_cnt);
  dd.low = basework+extrawork;
  dd.high = basework;

  for (i = 0; i<r; ++i) {
    par_do_mpi_work(df_cyclic2, &dd, 1.0, c);
    mpi_commpattern_sendrecv(buf, DIR_UP, 0, 0, c);
  }
  free_mpi_buf(buf);
}
```

In the MPI *late sender* pattern, a process executing a receive operation is blocked because the corresponding send operation is executed too late. We implement this by using the generic communication pattern `mpi_commpattern_sendrecv()` and arranging that the sending processes (on the even ranks) are always late by assigning them more work. This can easily done by using a cyclic2 distribution. Because we need a very specific distribution here, we cannot parameterize this performance property function as usual. Instead of allowing the user to specify a generic distribution, it is only possible to specify the amount of basic work for all ranks (`basework`) and extra work for the sending processes (`extrawork`).

In the two examples above, we simply repeated the same pattern as often as specified by the repetition factor `r`. Of course, more complicated implementations are possible, e.g., where the severity of the pattern is a function of the iteration number. This can easily be implemented by using the scale factor parameter of the distribution functions.

The following collection lists all performance property functions implemented in our ATS prototype. We plan to greatly enhance this list in the next months.

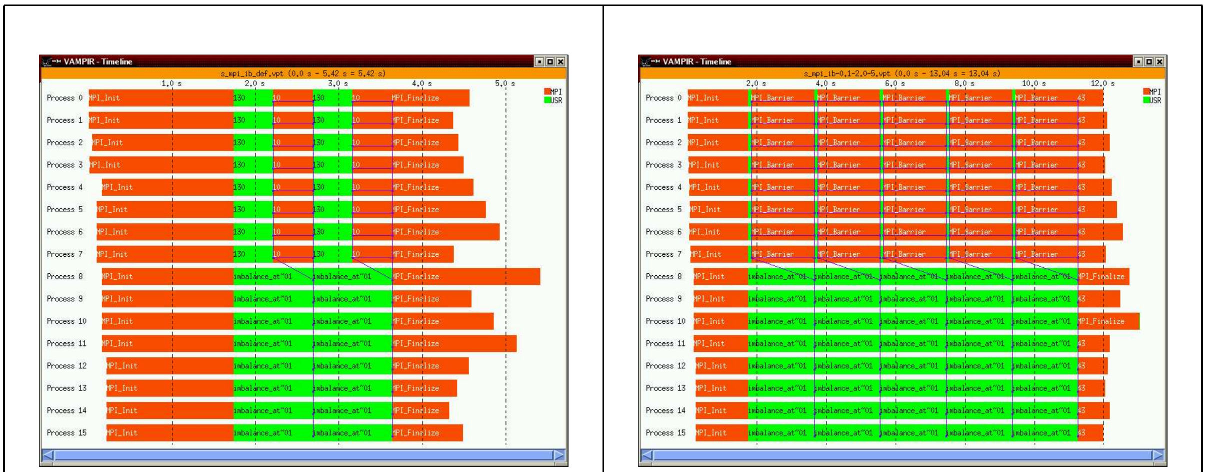
Currently implemented performance property functions

```
/* -- MPI Point-to-Point Communication Performance Properties -- */
void late_sender(double basework, double extrawork, int r, MPI_Comm c);
void late_receiver(double basework, double extrawork, int r, MPI_Comm c);

/* -- MPI Collective Communication Performance Properties -- */
void imbalance_at_mpi_barrier(distr_func_t df, distr_t* dd,
                              int r, MPI_Comm c);
void imbalance_at_mpi_alltoall(distr_func_t df, distr_t* dd,
                               int r, MPI_Comm c);
void late_broadcast(double basework, double rootextrawork, int root,
                    int r, MPI_Comm c);
void late_scatter(double basework, double rootextrawork, int root,
                  int r, MPI_Comm c);
void late_scatterv(double basework, double rootextrawork, int root,
                   int r, MPI_Comm c);
void early_reduce(double rootwork, double baseextrawork, int root,
                  int r, MPI_Comm c);
void early_gather(double rootwork, double baseextrawork, int root,
                  int r, MPI_Comm c);
void early_gatherv(double rootwork, double baseextrawork, int root,
                   int r, MPI_Comm c);

/* -- OpenMP Parallel Region Performance Properties -- */
void imbalance_in_omp_pregion(distr_func_t df, distr_t* dd, int r);
void imbalance_at_omp_barrier(distr_func_t df, distr_t* dd, int r);
void imbalance_in_omp_loop(distr_func_t df, distr_t* dd, int r);
```

3.2 Single Performance Property Testing

Based on the collection of performance property functions, a collection of test programs can be build, to allow testing of each performance property separately. We envision that these test programs can be generated automatically from the performance property function signatures, e.g., using a parser tool like PDT [9]. Using the information about the function argument types, it is easy to generate a main program skeleton which reads the necessary property parameters from the command line and then calls the property function after initializing parallel execution if necessary.

More extensive experiments based on these synthetic test programs can then be executed through scripting languages or through automatic experiment management systems, such as ZENTURIO [10].


VAMPIR - Timeline
s_mpi_ib_def.vpt (0.0 s - 5.42 s = 5.42 s)
MPI
USR
VAMPIR - Timeline
s_mpi_ib-0.1-2.0-5.vpt (0.0 s - 13.04 s = 13.04 s)
MPI
USR

ib_at_mpi_barrier block2 0.5 1.0 2

ib_at_mpi_barrier block2 0.1 2.0 5

Figure 3.2: Vampir Timeline Display of Two Executions of Single Performance Property Test Programs with Different Parameters.

Figure 3.2 shows Vampir timeline displays of an example of such a synthetic test program generated for the performance property *Imbalance at MPI Barrier*. Using different command line parameters allows testing for different distributions with different characteristics with different severities. Unfortunately, these displays also show another performance property which might be detected by good automatic performance analysis tools, namely *High MPI Initialization/Finalization Overhead*, which is hard to avoid in the view of the small sizes of the test programs.

3.3 Composite Performance Property Testing

Beyond testing for single performance properties separately, to test whether an automatic performance analysis tools works at all, it is of course possible to implement arbitrarily complex test programs just by invoking more than one performance property function in the same program. This can be used to test additional functionality, for example, whether performance problems can also be detected if they appear only in parts of the program, or when a program shows several performance properties, whether the tool can rank them correctly.

The figures on the following pages demonstrate two examples of composite performance property test programs. Figure 3.3 shows a Vampir timeline of an MPI test program which simply calls all currently defined MPI property functions with different severities and repetition factors. This program can be used to quickly determine how many different performance properties can be detected by a performance tool.

Figure 3.4 shows a slightly more complicated program. After initialization, the lower and the upper half of the participating MPI processes form different communicators. Then, the group of processors in each communicator each call a different set of performance property functions. This means that two different performance properties are active at the same time in parallel. This situation still can easily be detected using current automatic performance analysis prototypes from inside the APART group. For example, Figure 3.5 shows the result presentation component of the EXPERT tool [1, 2]. In the left (performance property) pane, it can be seen that EXPERT found (among others) the *Late Broadcast* performance property. The middle (call graph) pane shows that it located it correctly at the `MPI_Bcast()` function call inside the performance property function `late_broadcast()`. Finally, the right (location) pane shows that the performance property was located at MPI ranks 8 and 9 to 15. This is also correct, as `late_broadcast()` was executed on the communicator with the upper half of the MPI ranks with an (communicator-local) root rank 1.

The very modular design of the ATS framework allows the easy construction of very complex composite performance property test cases. For example, it is also possible to use performance property functions from different parallel programming paradigms in the same program, so that performance tools for hybrid programming can be tested. Again, these could use multiple MPI communicators and/or involve nested OpenMP parallelism resulting in several OpenMP thread groups, each executing different or the same sets of performance property functions in parallel.

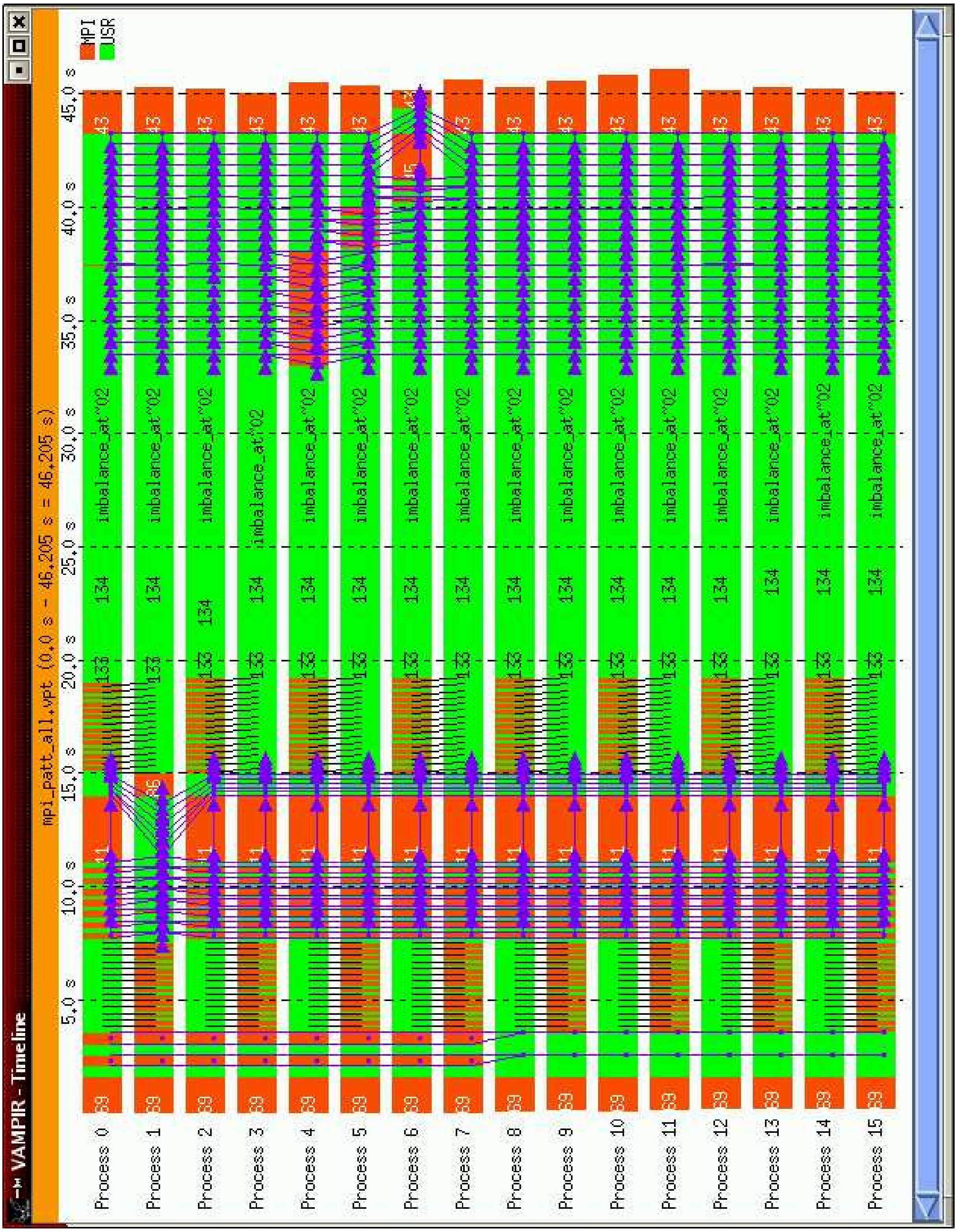


Figure 3.3: Vampir Timeline Display of Collection of MPI Performance Property Function Executions.

However, it remains to be tested how portable such complicated test cases would be, i.e., whether the performance properties in such a program behave the same on different computing platforms, or whether the differences in the platforms results in major differences in execution behavior.

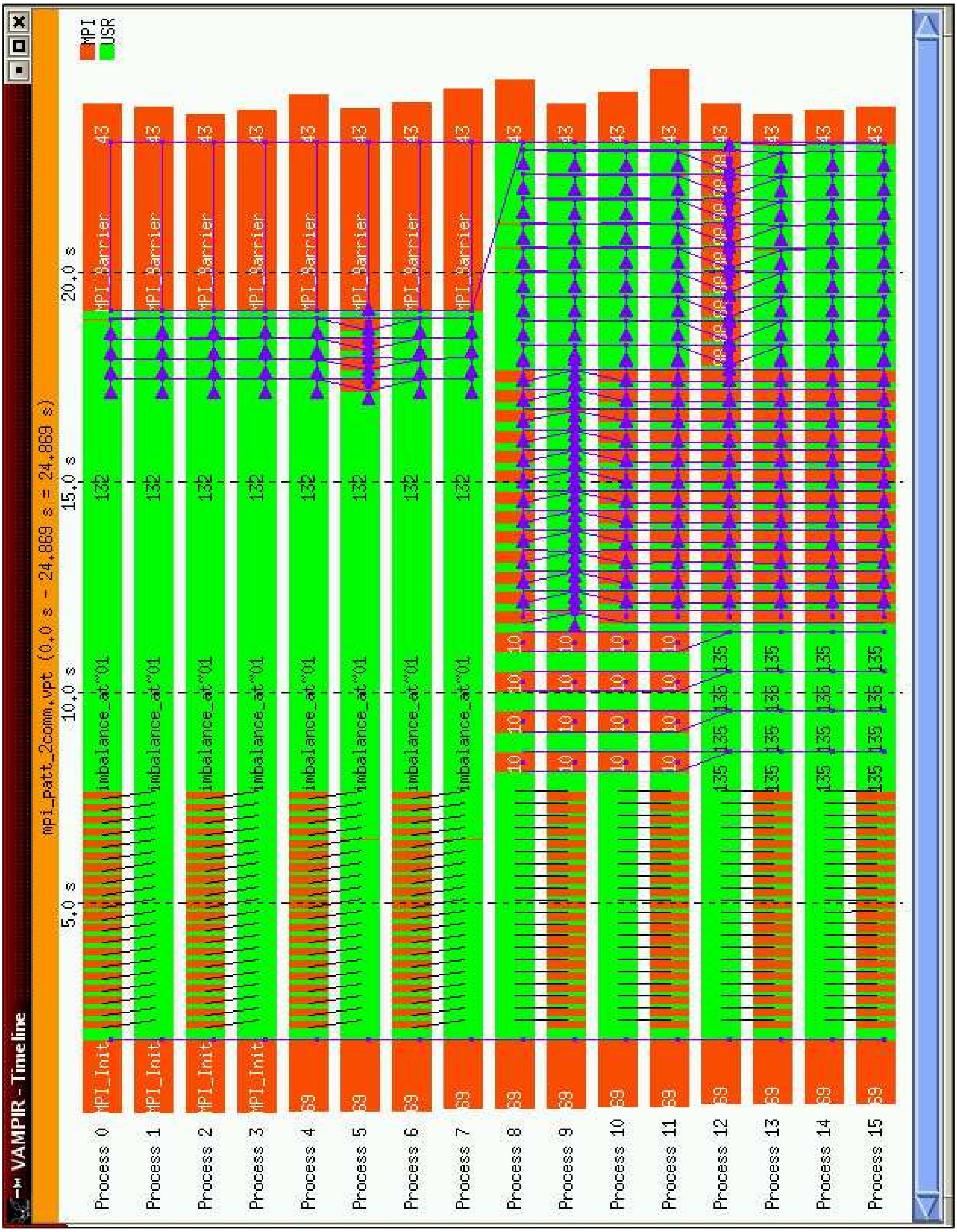
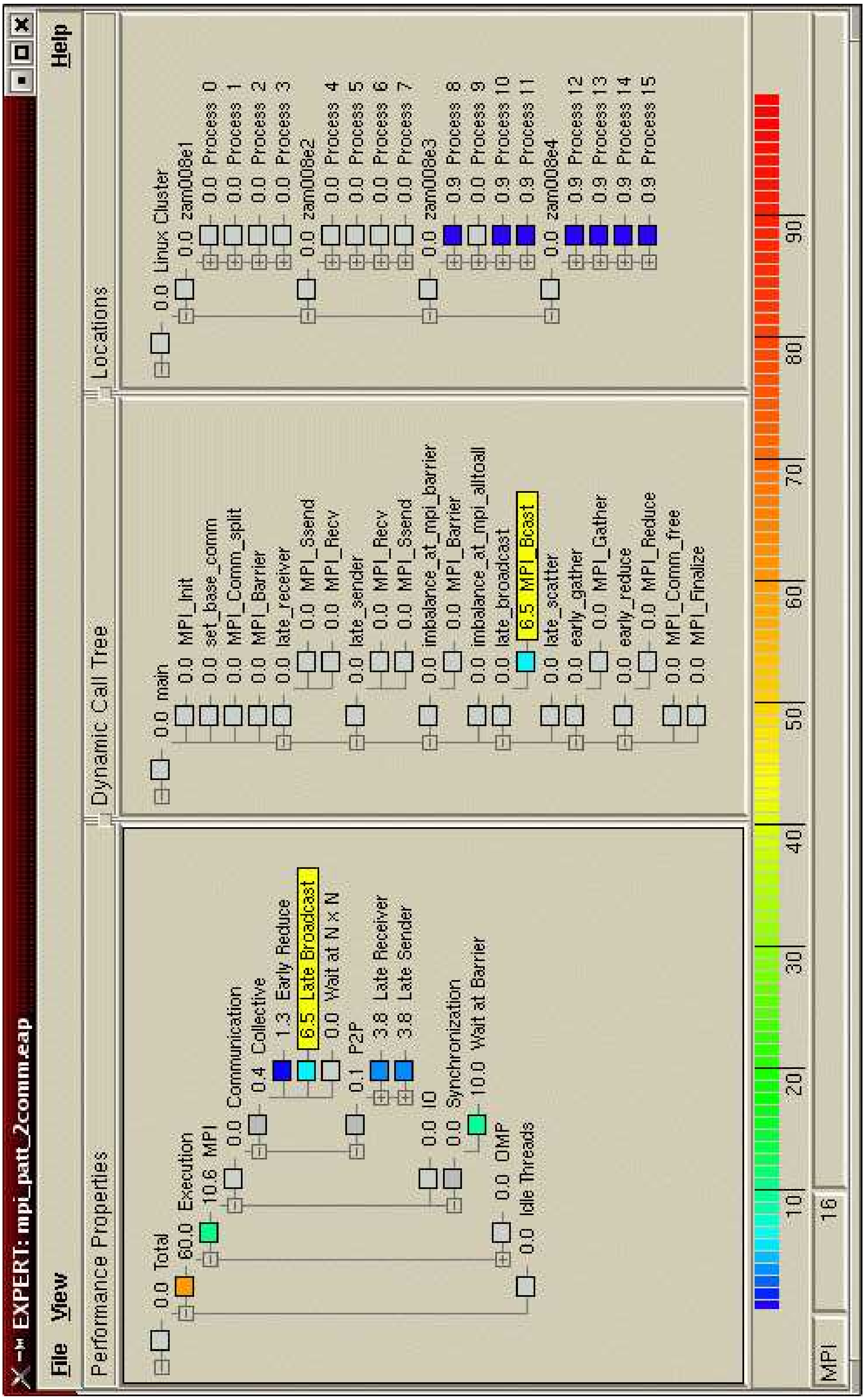


Figure 3.4: Vampir Timeline Display of Two Collections of MPI Performance Property Function Executing in Parallel in Different Communicators.



```
EXPERT: mpi_patt_2comm.eap
File  View  Help

Performance Properties
0.0 Total
    60.0 Execution
        10.6 MPI
            0.0 Communication
                0.4 Collective
                    1.3 Early Reduce
                    6.5 Late Broadcast
                    0.0 Wait at N x N
                0.1 P2P
                    3.8 Late Receiver
                    3.8 Late Sender
            0.0 IO
            0.0 Synchronization
                10.0 Wait at Barrier
        0.0 OMP
    0.0 Idle Threads

Dynamic Call Tree
0.0 main
    0.0 MPI_Init
    0.0 set_base_comm
    0.0 MPI_Comm_split
    0.0 MPI_Barrier
    0.0 late_receiver
        0.0 MPI_Ssend
        0.0 MPI_Recv
    0.0 late_sender
        0.0 MPI_Recv
        0.0 MPI_Ssend
    0.0 imbalance_at_mpi_barrier
        0.0 MPI_Barrier
    0.0 imbalance_at_mpi_alltoall
    0.0 late_broadcast
        6.5 MPI_Bcast
    0.0 late_scatter
    0.0 early_gather
        0.0 MPI_Gather
    0.0 early_reduce
        0.0 MPI_Reduce
    0.0 MPI_Comm_free
    0.0 MPI_Finalize

Locations
0.0 Linux Cluster
    0.0 zam008e1
        0.0 Process 0
        0.0 Process 1
        0.0 Process 2
        0.0 Process 3
    0.0 zam008e2
        0.0 Process 4
        0.0 Process 5
        0.0 Process 6
        0.0 Process 7
    0.0 zam008e3
        0.9 Process 8
        0.0 Process 9
        0.9 Process 10
        0.9 Process 11
    0.0 zam008e4
        0.9 Process 12
        0.9 Process 13
        0.9 Process 14
        0.9 Process 15

10  20  30  40  50  60  70  80  90
MPI  16
```

Figure 3.5: EXPERT Analysis of Two Collections of MPI Performance Property Function Executing in Parallel in Different Communicators.

Chapter 4

Applications

Finally, is has to be tested that performance analysis tools also work for large, complex applications in use on today's computer systems, not just only for carefully constructed "simple" test cases like the simple and composite performance property test programs introduced in the last sections. Here, like for validation and benchmark suites, we again propose to collect pointers to publicly available application programs together with a standardized description on the APART web site. This could include

- short description of the application,
- pointers to publications describing the program,
- installation and usage instructions,
- already tested computer systems / ports,
- descriptions of the application's performance behavior,
- and already available performance data.

It would also be possible to integrate this work with projects from the APART group on performance databases and project management, e.g., the PPerfXchange project [11] which allows the integration of heterogeneous, geographically dispersed data.
A starting point could be the collections of benchmark programs used for computer procurements like:

- The NAS Parallel Benchmarks (NPB)
  `http://www.nas.nasa.gov/Software/NPB/`
- The ASCI Purple and Blue Benchmark Codes
  `http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html`
  `http://www.llnl.gov/asci_benchmarks/asci/asci_code_list.html`

Chapter 5

Conclusion and Future Work

Automatic performance tools, like any other tools, must be tested as to whether they perform their task correctly. First, it must be ensured that that the tools do not alter the semantics of the application under investigation. Next, it must be verified that the tools collect the correct performance data as required. Finally, it must be checked that the tools indeed perform their intended tasks: For badly performing applications, relevant performance problems must be detected and reported, and, on the other hand, tools should not diagnose performance problems for well-tuned programs without such problems.

In this report we discussed the initial design and a very first prototype of the APART test suite (ATS) for automatic parallel performance analysis tools. In particular, we described a framework which allows to easily construct both simple and more complex synthetic test programs which exhibit the desired performance properties.

Lots of work remains to be done:

- Make the initial ATS prototype available on APART web site, so other tool developers inside and outside the APART group can test it and give valuable feedback. It will include
  - The collection of MPI, OpenMP, and hybrid validation and benchmark suites
  - The prototype version of the ATS framework
- The list of publicly available "real world" applications needs to be extended. Also, a more complete set of validation and benchmark suites needs to be collected.
- Implementation work on the ATS Framework needs to be continued.
  - We need a complete list of performance property test functions for MPI, OpenMP, and hybrid as described in [7].
  - We also need test functions for sequential performance properties.
  - The single property test program generator needs to be implemented as outlined in this paper.
  - Documentation!
- Currently, ATS is implemented using the C programming language. Because of its importance in the scientific computing community, we also need a Fortran version, ideally automatically generated from the C version.
- On request (or need), performance property functions for other parallel programming paradigms like HPF, PVM, or POSIX threads could be implemented.

Bibliography

[1] F. Wolf, B. Mohr. *Automatic Performance Analysis of SMP Cluster Applications*. Technical Report IB-2001-05, Forschunszentrum Jülich, 2001.

[2] F. Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. Dissertation, Forschunszentrum Jülich, 2002.

[3] A. Espinosa. *Automatic Performance Analysis of Parallel Programs*. PhD thesis, Universitat Autonoma de Barcelona, 2000.

[4] H.-L. Truong, T. Fahringer, G. Madsen, A. D. Malony, H. Moritsch, S. Shende. *n Using SCALEA for Performance Analysis of Distributed and Parallel Programs*. SC 2001, Denver, 2001.

[5] H.-L. Truong, T. Fahringer. *SCALEA: A Performance Analysis Tool for Distributed and Parallel Programs*. Euro-Par 2002, Paderborn, 2002.

[6] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvine, K. L. Karavanic, K. Kunchithapadam, T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[7] T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, J. L. Träff. *Knowledge Specification for Automatic Performance Analysis - APART Technical Report, Revised Version*. Technical Report IB-2001-08, Forschunszentrum Jülich, 2001.

[8] M. Gerndt. *Specification of Performance Properties of Hybrid Programs on Hitachi SR8000*. Peridot Technical Report, TU München, 2002

[9] K. A. Lindlan, J. Cuny, A .D. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen. *A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates*. SC 2000, Dallas, 2000.

[10] T. Fahringer, R. Prodan. *ZENTURIO: An Experiment Management System for Cluster and Grid Computing*. 4th Intl. APART Workshop, Euro-Par 2002, Paderborn, 2002.

[11] K. Karavanic. *PPerfXchange: Integrating Heterogeneous, Geographically Dispersed Data in a Parallel Performance Tool*. 4th Intl. APART Workshop, Euro-Par 2002, Paderborn, 2002.

Acknowledgments

We would like to thank all our partners in the IST Working Group APART for their contributions to this topic.