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

Interner Bericht

EARL - Language Reference

Felix Wolf, Bernd Mohr

FZJ-ZAM-IB-2000-01

Februar 2000

(letzte Änderung: 07.08.2000)

EARL – Language Reference

Version 2.0 / August 3, 2000

Felix Wolf and Bernd Mohr
Research Centre Jülich

Contents

Part I
Fundamentals

1 Introduction

The development of parallel applications is still a very complex and expensive process. This is essentially a result of the close relationship between the algorithm to be implemented on the one hand and the properties of the target platform in conjunction with the employed programming model on the other hand. Only when all three components of a parallel solution fit together in the right way, the application is able to achieve the desired performance.

However, the way in which these components interact is quite sophisticated and often difficult to understand. Especially using the message passing programming model, one of the most popular programming models on todays widespread massive parallel systems, often results in mysterious program behavior making the creation of scalable and fast applications a challenging task.

The complexity of current systems involves incremental performance tuning through successive observations and code refinements. A critical step in this procedure is transforming the collected data into a useful hypothesis about inefficient program behavior. Both the kind of data available and the tools we can use to interpret the data have major impact on the quality of our hypothesis.

We can get the most detailed view on the behavior of the parallel application by tracing runtime events that determine its performance properties. This task can be performed by tools like PAT [4]. Then, a performance problem can be considered as an event pattern or compound event which has to be detected in the event trace after program termination. The compound event is built from primitive events such as those associated with entering a program region or sending a message. One advantage of this technique is the existence of many powerful graphical visualization tools like VAMPIR [1] which may help to identify the desired patterns manually. But automatically locating and classifying performance problems would accelerate this process considerably.

The **E**vent **A**nalysis and **R**ecognition **L**anguage EARL was designed to provide a basic building block for automatic analysis of traces generated from message passing programs. It is actually a new high-level trace analysis language allowing to easily construct new trace analysis tools by writing scripts in the EARL language. These are then executed by the EARL interpreter.

In the context of EARL a performance problem can be considered as an event pattern occurring in the event trace produced by the parallel application. The EARL trace analysis language helps to easily specify an appropriate search algorithm by providing useful abstractions allowing the algorithm to have a very simple structure even in case of complex event patterns.

For example, one important feature of EARL is that frequently used higher-order events such as region instances or message transfers are represented as links between their constituent events which can easily be traversed by a search script. Furthermore, EARL supports navigation through function call stacks and message queues at a given execution state of the parallel program, enabling compact specification and efficient detection of the requested compound event.

An EARL performance analysis script usually takes one or more trace files as input. These are automatically mapped to the EARL event trace model by the EARL interpreter, independently of the underlying trace format, thereby allowing efficient random access to the events recorded in the file. Currently, EARL supports the VAMPIR [1], ALOG, and CLOG [5] trace formats.

Whereas the initial version of EARL [10] was implemented as extension of the Tcl interpreter [9] only, the current version also allows writing EARL scripts in Perl [8] and Python [3], so each EARL programmer can choose the scripting language that fits his needs best and can benefit from the full range of newest scripting technology. In addition, a redesign of the user interface resulted in a smaller set of simpler but still powerful orthogonal commands.

The idea of embedding the EARL language in a general purpose scripting language offers many interesting possibilities for tool design based on EARL. The multitude of powerful built-in features or extensions available on the Internet, e.g. for constructing graphical user interfaces, provides additional support for creating valuable performance tools. In particular the facilities for interprocess communication allow conducting experiments under tool control, e.g. running a parallel application with changing input parameters or hardware configuration.

The first part of this document presents the basic concepts and gives a summary of the language's semantics and its implementation. The second part provides a detailed language reference. In the last part we present three script examples to give an impression of how to use EARL in practice.

2 The EARL Event Trace Model

Much of the power of EARL comes through its very high-level abstraction of an event trace allowing a programmer to concentrate on the trace analysis and let EARL take care of the different trace formats and their encoding of functions and event types, of input handling and buffering, and of keeping track of message queues and call stacks.

The EARL event trace model defines the way an EARL programmer views an event trace. The EARL event trace model describes event types and system states and how they are related. An event trace is considered as a sequence of events. The events are sorted according to their timestamp and numbered starting at 1. There are different event types. EARL defines four predefined event types: entering (named *enter*), and leaving (*exit*) a region, and sending (*send*) as well as receiving (*recv*) a message. There may be more event types defined depending on the underlying trace format or depending on a specific event trace. A region is a named section of the traced program, e.g. it could be a loop or basic block, but in most cases it is a function or subroutine. If supported by the trace format, regions may be organized in groups, e.g. user or system functions.

2.1 Event Types

An event type is represented by an n-tuple of attributes. An event instance is defined by the values assigned to these attributes. The number of attributes depends on the type of the event.

All event types share a set of typical attributes such as a timestamp (*time*) or the location (*loc*) where the event happened. A location can be a process, thread, or ma-

chine, or a combination thereof [1]. The event type is explicitly given as a string attribute (*type*). However, the most important attribute is the position (*pos*) of the event within the event trace. It is needed to uniquely identify an event and is assigned according to the chronological order of the events.

The *enter* and *exit* event types have an additional *region* attribute specifying the name of the region entered or left. *send* and *recv* have attributes describing the destination (*dest*), source (*src*), tag (*tag*), length (*len*), and communicator (*com*) of the message.

In addition to these four standard event types, the EARL event trace model provides a template for event types that are not part of the basic event trace model. These types may be types depending on a special trace file format or even types defined for a specific event trace. Besides the five basic attributes *pos*, *loc*, *time*, *type* and *enterptr* the template provides two generic attributes without predefined semantics (*data1* and *data2*).

The concepts of region instances and messages are realized by two special attributes (Fig. 1). The *enterptr* attribute which is common to all event types points to the *enter* event that determines the region instance in which the event happened. In particular *enterptr* links two matching *enter* and *exit* events together. Apart from that, *recv* events provide an additional *sendptr* attribute to identify the corresponding *send* event. In general, references to other events are expressed by specifying the positions of the respective events.

The EARL event types and their attributes are summarized in Tab. 1.

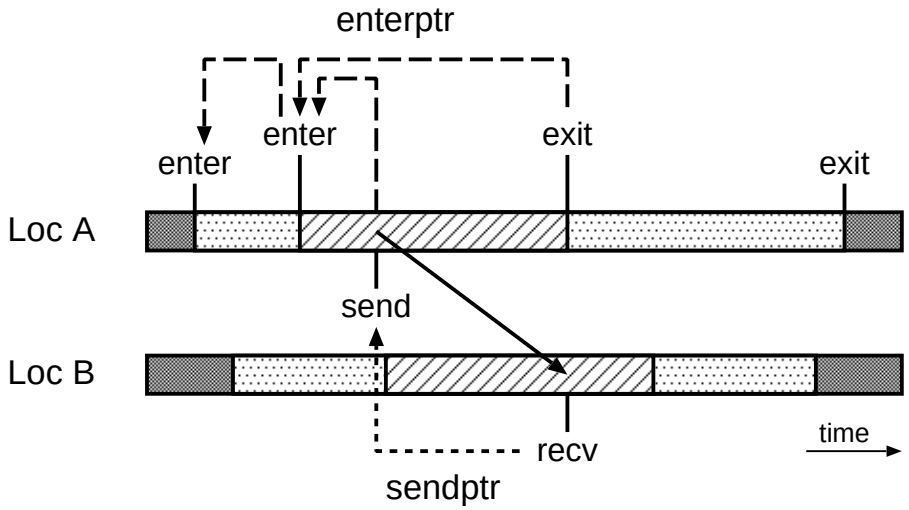

Figure 1: References provided by *enterptr* and *sendptr*

2.2 System States

For each position in the event trace, EARL also defines a system state reflecting the state in which the system was after the event at this position took place. A system state consists of one region stack per location and a message queue. The region stack is defined as the set of *enter* events that determine the regions in which the program executes at a given moment, and the message queue is defined as the set of *send* events of the messages sent but not yet received at that time. The message queue can be partitioned according to the

[1] In the context of MPI the location usually represents the rank of a process in MPI_COMM_WORLD.

source and destination locations of each individual message. The system state as a whole is just the union of the region stacks and the message queue.

In addition to querying theses structures directly, it is possible to navigate step-by-step through the region stack using the *enterptr* attribute or to trace back messages by following the *sendptr* attribute.

Table 1: The EARL event types

All types	
pos	Position of the event within the event trace. Positions are integer values starting at 1
loc	Location where the event happened. Locations are numbered from 0 to $n-1$ where n is the total number of locations used by the program
time	Timestamp in seconds as floating point value
type	Type of the event as a string value. This is either one of *enter*, *exit*, *send*, *recv*, or the identifier of a type that is not part of the basic model
enterptr	A reference (position) pointing to the *enter* event of the region instance in which the program was executing at the moment just before the event happened. If the event happened at top level, the attribute value is 0.

Entering a region (enter)	
region	Name of the region entered

Leaving a region (exit)	
region	Name of the region left

Sending a message (send)	
dest	Target location of the message
tag	Message tag
com	Communicator of the send operation
len	Message length in bytes

Receiving a message (recv)	
src	Source location of the message
tag	Message tag
com	Communicator of the receive operation
len	Message length in bytes
sendptr	Reference to the corresponding send event

Template for additional types	
data1	A generic attribute which can either have a string, an integer, or a floating point value
data2	A generic attribute which can either have a string, an integer, or a floating point value

3 The EARL Language

The core of EARL consists mainly of two wrapped C++ classes whose operations are mapped to corresponding commands of the supported scripting languages. Whereas the initial version of EARL used a hand made Tcl interface, now all interfaces are uniformly generated by SWIG [2]. The EARL extensions follow the object-oriented style also used by the Tk extension [9] which is available for all three languages.

The first class is named `EventTrace`. This class represents an event trace, i.e. a sequence of events in chronological order. It provides a mapping of the events from a trace file to the EARL event trace model. The second class `P2Statistic` can be used to calculate quantiles of a very large number of values like the execution times of a region or the transfer rates of messages.

3.1 EventTrace

Objects of this class are created with a trace file as input. Now, the trace file can be accessed according to the EARL event trace model.

An event can be considered as set of (*key, value*) pairs, where *key* denotes the name of an attribute and *value* its corresponding value. A natural way of representing such an object in a programming language would be a hash index structure h in a way that $h(key) = value$. All supported scripting languages provide such a structure in form of hash values (Perl), dictionaries (Python), or arrays (Tcl).

`EventTrace` provides several operations for accessing events: `event()` returns a hash value or a dictionary (Perl and Python) or assigns the event's attribute values to the elements of an array variable (Tcl). However, if you want to get a literal representation of an event, this operation does not suffice, because it does not preserve any order of the event attributes. Therefore, you can alternatively get a list containing the attribute names of the event in the order specified by the EARL event trace model by calling the `attributes()` operation. For convenience, the `values()` operation provides a list with the attribute values in the same order.

EARL automatically calculates the state of the region stacks and the message queue for a given event. The `stack()` operation returns the stack of a specified location as a list of the positions of the corresponding *enter* events. The `queue()` operation returns the message queue as a list of positions that point to the corresponding *send* events. If only messages with a distinct source or destination location are considered, the source and destination location can by specified as argument to the `queue()` operation.

All operations for processing an event can be used in two ways. Either you can specify the position of the event explicitly or you can implicitly use the position of the *iterator* by omitting the position argument. The iterator is owned by the EventTrace object and can be used to walk sequentially through an event trace without keeping track of the position yourself. If you want to jump back during your walk, you can do so by simply specifying an explicit target to the respective operation. The iterator remains unchanged and remembers your position. The iterator can be moved forward (`next()`), backward (`prev()`), or moved to an arbitrary position (`jump()`).

There are also several operations to access general information about the event trace. EARL allows to get a list of all defined event types (`types()`), the filename (`file()`) and format (`format()`) of the trace file, the number of locations used in the parallel application (`nrlocs()`), and a list of all defined regions (`regions()`) and groups (`groups()`). If you

a interested in the regions belonging to a certain group, you can supply the name of that group as optional argument to the `regions()` operation. The `group()` operation maps a given region to its group. It is also possible to ask for location symbols defined in the trace file with the `locsym()` operation.

3.1.1 Implementation Notes

During runtime of an EARL script the EARL interpreter dynamically builds up a sparse index structure on the trace file. At fixed intervals important trace state information (including the region stacks and the message queue) is stored in so-called *bookmarks* to speed up random access to events. If an event i is requested by an EARL script, the EARL interpreter usually does not have to start reading from the beginning of the trace file to find it. Instead, the interpreter looks for the nearest bookmark and takes the system state information from there which is required to correctly interpret the subsequent events from the file. Then it starts reading the events from there until it reaches the desired event.

To gain further efficiency, EARL automatically caches the most recently processed events in the *history* buffer. The history buffer always contains a contiguous subsequence of the event trace and the system state referring to the beginning of this subsequence. So all information related to events in the history buffer can completely generated from the buffer including region stacks and message queues.

3.2 P2Statistic

An object of this class represents a set of numeric values, e.g. observations.

P2Statistic provides an operation `add()` for adding values to the data set. An operation `count()` gives you the number of values added so far.

At any point, you can ask for different quantiles such as the median (`med()`), the 25% (`q25()`) and the 75% (`q75()`) quantile. The quantiles are actually estimates computed with the P^2 algorithm [7] which makes it unnecessary to store the complete data set. So the size of an `P2Statistic` object is very small and always constant. Using the P^2 algorithm is the reason for naming the class `P2Statistic`.

You can also request statistical standard information like the mean value (`mean()`), extreme values (`min()`, `max()`), the sum (`sum()`), and the variance (`var()`) of the values in the data set.

Part II
Language Reference

This part contains a detailed description of the EARL language. It is assumed that the reader is already familiar with the syntax of the scripting language she or he intends to use. Please consult the corresponding language manuals.

For each class and operation we first give a short description, explain the arguments if necessary, and then describe how the operation is used in each of the supported scripting languages. For each language mapping we illustrate the usage with a short code sequence.

4 EventTrace

- Life Cycle Management
  - Object creation: `EventTrace()`
  - Object deletion: See section 6
- Operations
  - Event Access
    - Get an event: `event()`
    - Attribute names: `attributes()`
    - Attribute values: `values()`
  - State Access
    - Region stack: `stack()`
    - Message queue: `queue()`
  - Iterator Control
    - Jump to a position: `jump()`
    - Move forward: `next()`
    - Move backward: `prev()`
    - Reinitialization: `reset()`
  - Miscellaneous
    - Name of the trace file: `file()`
    - Trace format: `format()`
    - Group of a region: `group()`
    - Groups of program regions: `groups()`
    - Location symbol: `locsym()`
    - Number of locations: `nrlocs()`
    - Program regions: `regions()`
    - Event types: `types()`

4.1 Object Creation

EventTrace()

Creates an EventTrace object from a trace file which has to be specified as input parameter.

Arguments

handle (Tcl only)
: Name of the command used to represent the object

file
: Name of the trace file

format (optional)
: Format of the trace file. Currently this may be one of `alog`, `clog`, or `vampir` (default)

bdist (optional)
: Distance between indexed positions (bookmarks) within the trace file (default 10000)

hsize (optional)
: Size of the history buffer in events (default 1000)

Perl

Arguments are specified in hash initializer syntax. All keys except for `file` can be omitted. The created object is returned.

```
$e = new EventTrace(file => "filename",
                    format => "vampir",
                    bdist => 10000,
                    hsize => 1000);
```

Python

All arguments except for `file` can be omitted. The created object is returned. (The example is shown in keyword argument syntax. Normal syntax is also possible with the arguments supplied in the order given below.)

```
e = EventTrace(file = "filename",
               format = "vampir",
               bdist = 10000,
               hsize = 1000)
```

Tcl

Only the `-file` switch is required. The handle name `e` is returned.

```
EventTrace e -file filename \
             -format vampir \
             -bdist 10000 \
             -hsize 1000
```

4.2 Event Access

event()

Gets an event from a trace file. The event is considered as a list of (*key*, *value*) pairs, where *key* is the name of an event attribute and *value* is the corresponding attribute value. The event is returned as a hash index structure h where $h(key)$ contains the corresponding value. The position of the requested event is supplied as argument. If the event does not exist, an exception is thrown. If the event position is omitted, the event at the current iterator position is chosen.

Arguments

arr (Tcl only)
: Name of the array variable to which the event is assigned

pos (optional)
: Position of the requested event.

Perl

The event is returned as a reference to a hash value.

```
$h = $e->event(102);
$time = $$h{"time"};

# or

%h = %{$e->event(102)};
$time = $h{"time"};
```

Python

The event is returned as a dictionary.

```
d = e.event(102)
time = d["time"]
```

Tcl

In Tcl complete arrays cannot be assigned to a variable. Instead, only scalar values can be assigned to individual elements of an array variable. Therefore, the name of the array variable to which the attribute values of the event are assigned has to be supplied as first argument. The position of the event is returned.

```
e event a 102
set time $a(time)
```

attributes()

Returns the attribute names of an event. The position of the requested event is supplied as argument. The attribute names are returned in the same order in which they are defined in the introductory part. If the event does not exist, an exception is thrown. If the event position is omitted, the event at the current iterator position is chosen.

Arguments

pos (optional)
: Position of the requested event.

Perl

Returns a reference to an array containing the attribute names.

```
$a = e->attributes();
print(@$a);
```

Python

Returns a list containing the attribute names.

```
a = e.attributes()
print a
```

Tcl

Returns a list containing the attribute names.

```
set a [e attributes]
puts $a
```

values()

Returns the attribute values of an event. The position of the requested event is supplied as argument. The values are returned in the same order in which they are defined in the introductory part and which is also used by the `attributes()` operation. If the event does not exist, an exception is thrown. If the event position is omitted, the event at the current iterator position is chosen.

Arguments

pos (optional)
: Position of the requested event

Perl

Returns a reference to an array containing the attribute values.

```
$v = e->values();
print(@$v);
```

Python

Returns a list containing the attribute values.

```
v = e.values()
print v
```

Tcl

Returns a list containing the attribute values.

```
set v [e values]
puts $v
```

4.3 State Access

stack()

Returns the region stack state at the time immediately after the event with the supplied position. If the position is omitted, the stack state refers to the current iterator position. The stack state can be queried for each location. `stack()` returns a list of positions pointing to *enter* events. These *enter* events mark the beginning of the region instances in which the program is executing at that moment. The returned positions are sorted in ascending order, that means, the most enclosing region comes last.

In contrast to the `event()` operation, specifying 0 as position parameter is valid. In this case the result is an empty list.

Arguments

loc The location for which the stack state is requested

pos (optional)
The position within the trace for which the stack state is requested

---

Perl

The stack is returned as a reference to an array containing the event positions.

```
$s = e->stack(1,102);
print(@$s);
```

Python

The stack is returned as a list containing the event positions.

```
s = e.stack(1,102)
print s
```

Tcl

The stack is returned as a list containing the event positions.

```
set s [e stack 1 102]
puts $s
```

queue()

Returns the state of the message queue at the time immediately after the event with the supplied position. If the position is omitted, the queue state refers to the current iterator position. The state of the message queue can be queried for each combination of source and destination location. `queue()` returns a list of positions pointing to *send* events by which the messages waiting in the queue have been sent. 'Waiting in the queue' means sent but not received yet. The returned positions are sorted in ascending order, that means, the oldest *send* event comes first.

In contrast to the `event()` operation, specifying 0 as position parameter is valid. In this case the result is an empty list.

Arguments

src
: The source location of the messages. If you want to consider messages from all possible locations, you can set src to -1.

dest
: The destination location of the messages. If you want to consider messages to all possible locations, you can set dest to -1.

pos (optional)
: The position to which the requested queue state refers.

Perl

The queue is returned as a reference to an array containing the event positions.

```
$q = e->queue(1,2,102);
print(@$q);
$q = e->queue(-1,-1,102); # all messages currently underway
print(@$q);
```

Python

The queue is returned as a list containing the event positions.

```
q = e.queue(1,2,102)
print q
q = e.queue(-1,-1,102)
print q
```

Tcl

The queue is returned as a list containing the event positions.

```
set q [e queue 1 2 102]
puts $q
set q [e queue -1 -1 102]
puts $q
```

4.4 Iterator Control

jump()

Moves the iterator to the position supplied as argument. The new position is returned. Only positions greater than 0 are valid targets of `jump()`. If the new position is not valid, 0 is returned and the iterator remains unchanged. So you can use `jump()` as a predicate to test if there is an event with the supplied position.

The initial position of the iterator is 0. Note that you cannot get an event with position 0, so jumping to this position would fail. But querying the system state relative to this position is valid, although the result would not be very interesting. To restore the initial position use `reset()`.

Arguments

pos	Position where to jump to.

Perl

```
$e->jump(102);
$h = e->event();
```

Python

```
e.jump(102)
d = e.event()
```

Tcl

```
e jump 102
e event a
```

next()

Increments the iterator by 1. If the new position is not valid, 0 is returned and the iterator remains unchanged. So you can use `next()` as a predicate to test if a next position (event) does exist.

Perl

```
$e->next();
$h = e->event();
```

Python

```
e.next()
d = e.event()
```

Tcl

```
e next
e event a
```

prev()

Decrements the iterator by 1. If the new position is not valid, 0 is returned and the iterator remains unchanged. So you can use `prev()` as a predicate to test if a previous position (event) does exist. Note that you can only reach positions with valid events by calling this operation. The zero position can only be reached by `reset()`. Otherwise, `prev()` could not be used as a predicate for testing valid event positions.

Perl

```
$e->prev();
$h = e->event();
```

Python

```
e.prev()
d = e.event()
```

Tcl

```
e prev
e event a
```

reset()

Resets the iterator to 0.

Perl

```
e->reset();
```

Python

```
e.reset()
```

Tcl

```
e reset
```

4.5 Miscellaneous

file()

Returns the name of the trace file.

Perl

```
$f = e->file();
```

Python

```
f = e.file()
```

Tcl

```
set f [e file]
```

format()

Returns the name of the format of the trace file. This is currently one of `alog`, `clog` or `vampir`.

Perl

```
$f = e->format();
```

Python

```
f = e.format()
```

Tcl

```
set f [e format]
```

group()

Returns the name of the group to which a given region belongs.

Arguments

region The region of which you want to know the group

Perl

```
$g = e->group("MPI_Send");
```

Python

```
g = e.group("MPI_Send")
```

Tcl

```
set g [e group MPI_Send]
```

groups()

Returns all groups of regions defined in the trace file.

Perl

Returns a reference to an array containing the group names.

```
$g = e->groups();
print(@$g);
```

Python

Returns a list containing the group names.

```
g = e.groups()
print g
```

Tcl

Returns a list containing the group names.

```
set g [e groups]
puts $g
```

locsym()

Returns the symbol for a given location defined in the trace file. If there is no symbol defined, the location number is returned.

Arguments

loc The location for which you want to get the symbol.

Perl

```
$l = e->locsym(1);
```

Python

```
l = e.locsym(1)
```

Tcl

```
set l [e locsym 1]
```

nrlocs()

Returns the number of locations used to run the parallel program.

Perl

```
$n = e->nrlocs();
```

Python

```
n = e.nrlocs()
```

Tcl

```
set n [e nrlocs]
```

regions()

Returns the names of the program regions defined in the trace file. If you only want to know the regions of a certain group, you can specify this group as optional argument.

Arguments

group (optional)
The group of which you want to know the regions

Perl

Returns a reference to an array containing the region names.

```
$all = e->regions();
print(@$all);
$mpi = e->regions("MPI");
print(@$mpi);
```

Python

Returns a list containing the region names.

```
all = e.regions()
print all
mpi = e.regions("MPI")
print mpi
```

Tcl

Returns a list containing the region names.

```
set all [e regions]
puts $all
set mpi [e regions MPI]
puts $mpi
```

types()

Returns the event types defined for the trace file. Usually, these are the standard event types (*enter*, *exit*, *send*, *recv*) and additional trace file or trace format specific event types.

Perl

Returns a reference to an array containing the event type names.

```
$t = e->eventTypes();
print(@$t);
```

Python

Returns a list containing the event type names.

```
t = e.eventTypes()
print t
```

Tcl

Returns a list containing the event type names.

```
set t [e types]
puts $t
```

5 P2Statistic

- Life Cycle
  - Object Creation:
  - Object Destruction: See section 6
- Operations
  - Managing the data set
    - Add a value to the data set: `add()`
    - Cardinality of the data set: `count()`
    - Reinitialization: `reset()`
    - Get a literal representation: `state()`
  - Quantiles
    - 25% quantile: `q25()`
    - Median: `med()`
    - 75% quantile: `q75()`
  - Miscellaneous
    - Minimum: `min()`
    - Maximum: `max()`
    - Mean value: `mean()`
    - Sum: `sum()`
    - Variance: `var()`

5.1 Object Creation

P2Statistic()

Creates a P2Statistic object. If you want, you can initialize the object with another object by providing its literal representation as constructor argument. You get the literal representation of a P2Statistic object by calling the `state()` operation.

Arguments

handle (Tcl only)
: Name of the command used to represent the object

state (optional)
: A list containing the values which define the state of the source object

---

Perl

The created object is returned.

```
$p = new P2Statistic();
$q = new P2Statistic($p->state());  # q is copy of p
```

Python

The created object is returned.

```
p = P2Statistic()
q = P2Statistic(p.state())  # q is copy of p
```

Tcl

The handle name `p` resp. `q` is returned.

```
P2Statistic p
P2Statistic q [p state] ; # q is copy of p
```

5.2 Managing the Data Set

add()

Adds a numeric value to the data set.

Arguments

value The floating point value to be added to the data set

Perl

```
p->add(3.1415);
```

Python

```
p.add(3.1415)
```

Tcl

```
p add 3.1415
```

count()

Returns the cardinality of the data set, i.e. the number of values added so far.

Perl

```
$c = $p->count();
```

Python

```
c = p.count()
```

Tcl

```
set c [p count]
```

reset()

Reinitializes the object. After applying this operation the data set is empty again.

Perl

```
p->reset();
```

Python

```
p.reset()
```

Tcl

```
p reset
```

state()

Returns the literal representation (state) of the object as list of numeric values. The list can be stored e.g. in a file and can later be used as constructor argument to recreate the object.

Perl

Returns a reference to an array containing the values which represent the object state.

```
$s = p->state();
print(@$s);
```

Python

Returns a list containing the values which represent the object state.

```
s = p.state()
print s
```

Tcl

Returns a list containing the values which represent the object state.

```
set s [p state]
puts $s
```

5.3 Quantiles

q25()

Returns the 25% quantile of the data set. The return value is an estimate computed with the P^2 algorithm. Requires at least five elements in the data set.

Perl

```
q = p->q25();
```

Python

```
q = p.q25()
```

Tcl

```
set q [p q25]
```

med()

Returns the median of the data set. The return value is an estimate computed with the P^2 algorithm. Requires at least five elements in the data set.

Perl

```
m = p->med();
```

Python

```
m = p.med()
```

Tcl

```
set m [p med]
```

q75()

Returns the 75% quantile of the data set. The return value is an estimate computed with the P^2 algorithm. Requires at least five elements in the data set.

Perl

```
q = p->q75();
```

Python

```
q = p.q75()
```

Tcl

```
set q [p q75]
```

5.4 Miscellaneous

min()

Returns the minimum of the data set. This operation requires at least one element in the data set.

Perl

```
$m = p->min();
```

Python

```
m = p.min()
```

Tcl

```
set m [p min]
```

max()

Returns the maximum of the data set. This operation requires at least one element in the data set.

Perl

```
$m = p->max();
```

Python

```
m = p.max()
```

Tcl

```
set m [p max]
```

mean()

Returns the mean value of the data set. This operation requires at least one element in the data set.

Perl

```
$m = p->mean();
```

Python

```
m = p.mean()
```

Tcl

```
set m [p mean]
```

sum()

Returns the sum of the elements in the data set. This operation requires at least one element in the data set.

Perl

```
$s = p->sum();
```

Python

```
s = p.sum()
```

Tcl

```
set s [p sum]
```

var()

Returns the variance of the elements in the data set. This operation requires at least one element in the data set.

Perl

```
$v = p->var();
```

Python

```
v = p.var()
```

Tcl

```
set v [p var]
```

6 Object Destruction

Deleting an object of an EARL class in the scripting language also calls the destructor of the corresponding C++ class and releases all allocated resources.

Perl

Normally, if the program is running out of the scope in which the variable holding the EARL object is defined, the object is deleted if no other reference to the object exists. Because of the way in which the Perl garbage collection interacts with wrapped C++ classes, it is recommended to explicitly release the object by assigning `undef` to the variable holding the object.

```
$e = SomeEarlClass(...);
# some code
$e = undef;
```

Python

Here nothing special needs to be considered. The EARL classes provide the usual destructor operation `del`.

```
e = SomeEarlClass(...)
# some code
del e
```

Tcl

Since EARL objects are represented by Tcl procedures, they can be deleted with `rename`. This also frees all allocated resources.

```
SomeEarlClass e ...
# some code
rename e {}
```

Part III
Examples

This part of the document describes three EARL script examples. Each of them is generic in the sense that it can be used with any message passing trace supported by EARL. Although simple (all are around 20 lines of code) they perform quite complex calculations. The simplicity comes from the abstractions defined in the EARL event trace model and the high-level nature of the scripting languages used.

Each example is written in another scripting language. This should help to understand how EARL is mapped to each of the supported languages.

- Compute Wasted Time of MPI_Recv (Perl)
- Passing Messages out of Order (Python)
- Region Statistics (Tcl)

7 Compute Wasted Time of MPI_Recv (Perl)

The first example demonstrates the capabilities of EARL to solve nonstandard problems, especially recognizing complex event patterns. Consider the following: For a set of event traces from a parallel MPI program, determine the time which is wasted when an MPI_Recv is posted before the corresponding MPI_Send was executed (see Fig. 2).

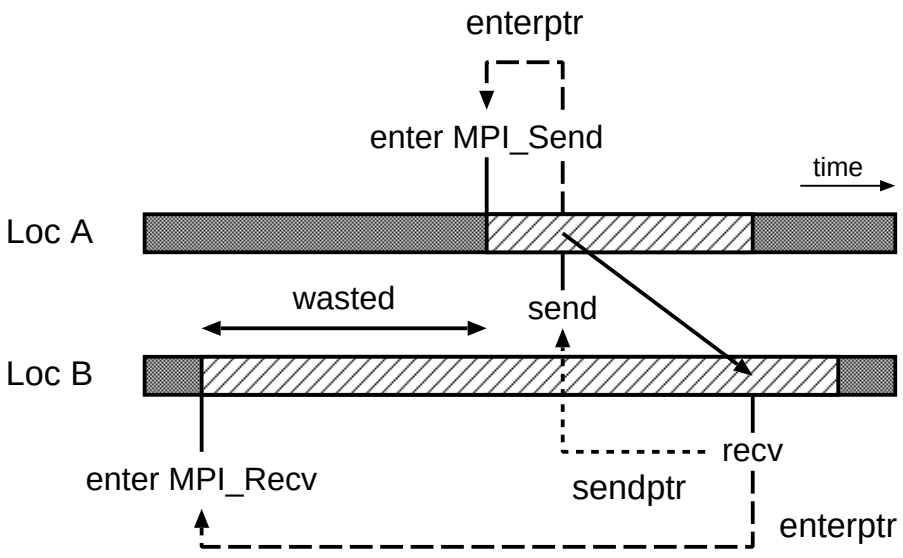

Figure 2: Wasted time in message passing programs

The complete EARL script is depicted in Fig. 3 [2]. Line 1 is a special comment which tells a Unix system which command to use to execute the following script file. Line 2 loads the earl module. Line 4 opens the trace file which is specified by the first command line parameter $ARGV[0] and stores the created EventTrace object in variable $e. The `while`

[2]The line numbers are not part of the source code.

```
 1: #!/usr/local/bin/perl
 2: use earl;
 3:
 4: $e = new EventTrace(file => $ARGV[0]);
 5:
 6: $sum_wasted = 0;
 7:
 8: while ($e->next())
 9: {
10:     %curr = %{$e->event()};
11:     if ($curr{"type"} eq "recv") {
12:
13:         %recv_start = %{$e->event($curr{"enterptr"})};
14:         next unless ($recv_start{"region"} eq "MPI_Recv")
15:
16:         %send       = %{$e->event($curr{"sendptr"})};
17:         %send_start = %{$e->event($send{"enterptr"})};
18:         next unless ($send_start{"region"} eq "MPI_Send")
19:
20:         $wasted =  $send_start{"time"} - $recv_start{"time"};
21:         if ($wasted > 0) {
22:             $sum_wasted += $wasted;
23:         }
24:     }
25: }
26: print($e->file(),": ",$sum_wasted, " sec wasted.\n");
27: $e = undef
```

Figure 3: Compute Wasted Time of MPI_Recv

in line 8 steps sequentially through the event trace, in each iteration moving the iterator forward by one position. In line 10 the event at the current iterator position is assigned to the hash %curr. If we find a *recv* event (line 11), we fill the hash %recv_start with the *enter* event of the enclosing region instance (line 13). If the enclosing region is not MPI_Recv (the message could have been sent from another routine, e.g., MPI_Broadcast), we skip the rest of the loop and continue the search (line 14). Next, we set hash %send to the corresponding *send* event (line 17), and again check whether it originated from an MPI_Send (line 18). We compute the difference between the beginning of MPI_Send and MPI_Recv (line 20) and add it to the variable $sum_wasted if MPI_Recv was executed before MPI_Send (line 22). Finally, we print the result (line 26) and close the trace file (line 29) by assigning `undef` to the variable holding the EventTrace object.

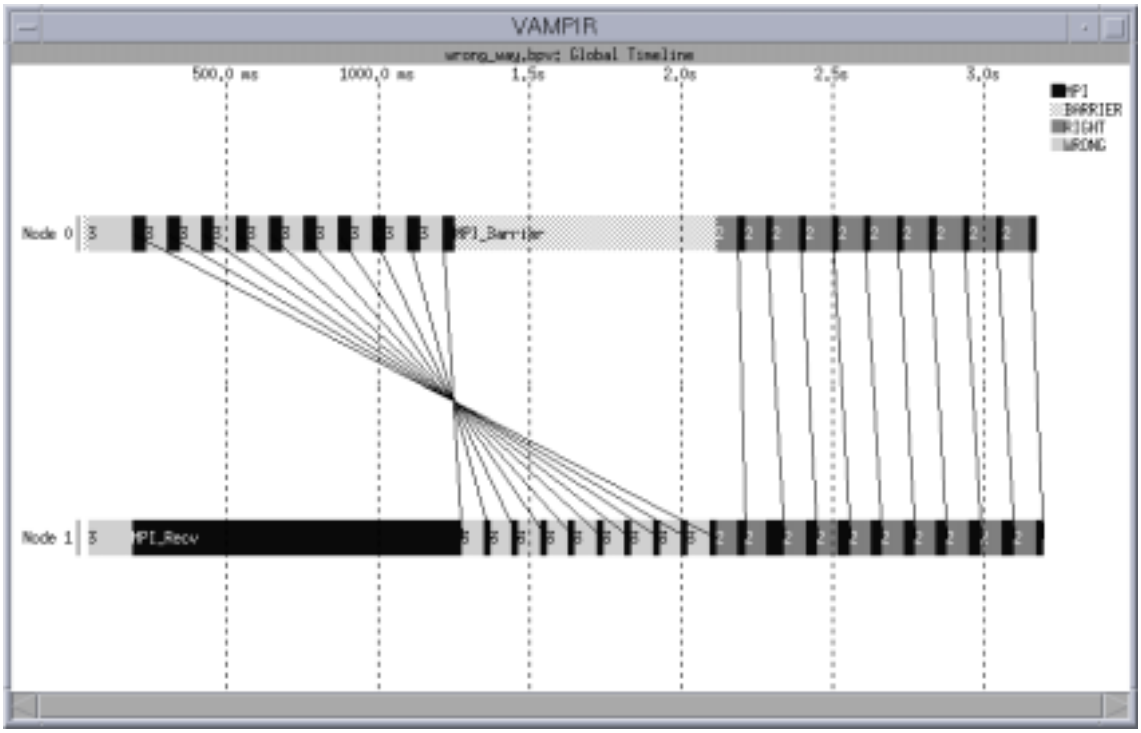

Figure 4: Passing messages out of order

8 Passing Messages out of Order (Python)

The second example demonstrates how EARL can be used to find programming errors in message passing programs. The example is taken from the *Grindstone test suite for parallel performance tools* [6] and highlights the problem of passing messages out-of-order. This problem could arise if one process is expecting messages in a certain order, but another process is sending messages which are not in the expected order. In Fig. 3 an extreme example is shown: In the left part of the picture Node 1 is processing incoming messages in the opposite order they were sent from Node 0. Processing them in the order they were sent would not only speed up the program but also requires much less buffer space for storing unprocessed messages. This is shown in the right part of the picture.

The EARL code for this example is quite simple. We open the trace file specified as first command line parameter (line 5) and sequentially loop through the events of the trace (line 7). If we find a *recv* event (line 10), we check for all messages still in the message queue sent to the current location from the same source location as the current message (line 12), whether the corresponding *send* event happened before the *send* event of the current message (line 14). As we only have to determine the order of these events in time and EARL provides references to other events by their positions, the necessary comparison can be done by comparing the references since the positions comply with the chronological order. If we find such an outstanding message, an error message is printed (line 15).

```
 1: #!/usr/local/bin/python
 2: from earl import *
 3: from sys  import *
 4:
 5: e = EventTrace(file=argv[1])
 6:
 7: while e.next():
 8:
 9:    curr = e.event()
10:    if curr["type"] == "recv" :
11:
12:        for send in e.queue(curr["src"], curr["loc"]) :
13:
14:            if  (send < curr["sendptr"]) :
15:                print "Received message in wrong order"
16:
17: del e
```

Figure 5: Passing messages out of order

9 Region Statistics (Tcl)

The third problem is the standard task of computing the time which is spent in each region of the program including and excluding the time spent in contained region instances (Fig. 6).

Line 2 indicates which package is required for understanding the following lines. Line 4 opens a trace file which is given the script as first command line parameter. Now, the trace is represented by a Tcl command named `e`. Next, we get the number of locations used (line 6) and a list of all regions defined (line 8). Then we iterate over all locations and regions (lines 9 to 14) to initialize the two arrays `incl` and `excl`, where for each location we will store the total inclusive and exclusive time spent in each region.

The `while` in line 14 steps sequentially through the event trace setting the array `curr` to the next event. If we find an *exit* event (line 16), we fill the array `enter` with the corresponding *enter* event of the region we are about to leave (line 17). In line 18 we calculate the time spent in this region. In lines 20 and 21, we add this time to the corresponding values in the arrays `incl` and `excl`. Here we use the auxiliary variable `index` as key which we have computed in line 19. Lines 22 to 25 subtract the execution time of the current region instance from the total exclusive execution time of the enclosing region (if there is one). In order to find it, we use the *enterptr* attribute of the *enter* event which should point to the beginning of the enclosing region instance. If *enterptr* is not 0 there is an enclosing region instance, and we can perform the subtraction.

After finishing the while loop the arrays `incl` and `excl` contain the desired information which now can be printed or displayed using bar graphs or pie charts.

```
 1: #!/usr/bin/tclsh8.0
 2: package require earl
 3:
 4: EventTrace e -file [lindex $argv 0]
 5:
 6: set n [e nrlocs]
 7:
 8: foreach r [e regions] {
 9:    for {set i 0} {$i < $n} {incr i} {
10:       set excl($i,$r) 0
11:       set incl($i,$r) 0
12:    }
13: }
14: while {[e next]} {
15:    e event curr
16:    if {$curr(type) == "exit"} {
17:        e event enter $curr(enterptr)
18:        set diff [expr $curr(time) - $enter(time)]
19:        set index "$curr(loc),$curr(region)"
20:        set incl($index) [expr $incl($index) + $diff]
21:        set excl($index) [expr $excl($index) + $diff]
22:        if {$enter(enterptr)} {
23:           e event encl $enter(enterptr)
24:           set excl($curr(loc),$encl(region)) \
25:                   [expr $excl($curr(loc),$encl(region)) - $diff]
26:        }
27:    }
28: }
29: # print results here
30: rename e {}
```

Figure 6: Region statistics

References

[1] A. Arnold, U. Detert, and W.E. Nagel. Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding. In R. Winget and K. Winget, editors, *Proc. of Cray User Group Meeting*, pages 252–258, Denver, CO, March 1995.

[2] David M. Beazley. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proc. of the 4th Tcl/Tk Workshop*.

[3] David M. Beazley. *Python Essential Reference*. New Riders, October 1999.

[4] Jim Galarowicz and Bernd Mohr. Analyzing Message Passing Programs on the Cray T3E with PAT and VAMPIR. In H. Lederer and F. Hertwick, editors, *Proceedings of Fourth European CRAY-SGI MPP Workshop*, pages 29–49, Garching/Munich (Germany), October 1998.

[5] William Gropp and Ewing Lusk. *User's Guide for MPE: Extensions for MPI Programs*. Argonne National Laboratory, 1998. http://www-unix.mcs.anl.gov/mpi/mpich/.

[6] J.K. Hollingsworth and M. Steele. Grindstone: A Test Suite for Parallel Performance Tools. Computer Science Technical Report CS-TR-3703, University of Maryland, October 1996.

[7] R. Jain and I. Chlamtac. The P^2 Algorithm for Dynamic Calculation of Quantiles and Histograms Without Storing Observations. *Communcations of the ACM*, 28(10), October 1985.

[8] Sriram Srinivasan. *Advanced Perl Programming*. O'Reilly, August 1997.

[9] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, second edition, 1997.

[10] F. Wolf and B. Mohr. EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. In A. Hoekstra and B. Hertzberger, editors, *Proc. of the 7th International Conference on High-Performance Computing and Networking (HPCN'99)*, pages 503–512, Amsterdam (The Netherlands), 1999.