cube
scalasca

CubeLib 4.10 | Derived Metrics

Intoduction in CubePL and Cube's derived metrics

November 2026
The Scalasca Development Team
scalasca@fz-juelich.de

JÜLICH
FORSCHUNGSZENTRUM

Attention

The Cube Derived Metrics User Guide is currently being rewritten and still incomplete. However, it should already contain enough information to get you started and avoid the most common pitfalls.

Contents

1 Introduction into Cube Derived Metrics **3**
1.1 Introduction . . . 3
1.2 Creation of a derived metric in Cube . . . 3
1.2.1 Using the C writer . . . 3
1.2.2 Using the C++ library . . . 4
1.3 The three kinds of derived metrics . . . 4

2 Reference of CubePL **7**
2.1 Supported calls in CubePL . . . 7

3 Syntax of CubePL **11**
3.1 Syntax step by step . . . 11
3.1.1 Expressions . . . 11
3.1.1.1 Constants . . . 11
3.1.1.2 Arithmetical Expressions . . . 11
3.1.1.3 Boolean Expressions . . . 11
3.1.1.4 Function call expressions . . . 11
3.1.2 Lambda function (In-place function definition) . . . 12
3.1.3 Control structures . . . 12
3.1.3.1 Condition IF-ELSE . . . 12
3.1.3.2 Loop WHILE . . . 12
3.1.4 Variables . . . 13
3.1.4.1 User defined Variables . . . 13
3.1.4.2 Predefined Variables . . . 13
3.1.5 Different ways to refer an another metric . . . 16
3.1.5.1 Context sensitive reference to another metric . . . 16
3.1.5.2 Context insensitive reference to another metric . . . 17
3.1.5.3 Direct reference to another metric . . . 17
3.1.5.4 Definition of an encapsulated calculation within CubePL expression using metrics of Cube. . . . 18
3.1.5.5 Definition of an initialization expression for ghost metrics within CubePL expression. . . . 19
3.1.5.6 Definition of an initialization expression for ghost metrics within CubePL expression. . . . 19
3.2 Grammar of CubePL . . . 20

4 Examples of CubePL expressions **21**
4.1 Simple Examples . . . 21
4.2 Complex Examples . . . 21
4.2.1 Different made-up expressions . . . 21
4.2.2 Special metrics of Scalasca . . . 22

Bibliography **23**

Copyright © 1998–2026 Forschungszentrum Jülich GmbH, Germany

Copyright © 2009–2015 German Research School for Simulation Sciences GmbH, Jülich/Aachen, Germany

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the names of Forschungszentrum Jülich GmbH or German Research School for Simulation Sciences GmbH, Jülich/Aachen, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1 Introduction into Cube Derived Metrics

1.1 Introduction

With Cube 4.1.0 derived metrics are introduced and provide a powerful tool for performance data processing.

Unlike normal metrics of the cube, derived metrics do not store any data inside of the cube report, but derive their value according to an expression, formulated using CubePL (Cube Processing Language) syntax ('3').

Cube supports different kinds ('1.3') of derived metrics. Derived metric can be a child metric of any metric of cube or other derived metrics.

Derived metrics are naturally metrics with the value `DOUBLE`

1.2 Creation of a derived metric in Cube

Derived metrics play the very same role as a normal metric in cube, therefore one proceeds with creation of derived metrics as usual. The only difference is the specification of the CubePL expression.

1.2.1 Using the C writer

To create a derived metric in a C program, one calls `cube_metric_set_expression(...)` subsequently to `cube_def_met(...)`. Here is a code example:

```
cube_metric * met = cube_def_met(   cube,
                                    "Derived metric",
                                    "derivedmetric1",
                                    "DOUBLE",
                                    "sec",
                                    "",
                                    "https://doc.web.org/metrics.html#derived",
                                    "Average execution time",
                                    NULL,
                                    CUBE_METRIC_PREDERIVED_EXCLUSIVE
                                );
  cube_metric_set_expression(met, "metric::time(i)/metric::visits(e)");
```

Derived metric specification allows to define an CubePL expression, which will be executed only once before the actual metric expression is executed. To specify an initalization expression, one uses the call `cube_metric_set_init_expression(...)`.

Prederived metrics allow to redefine operators "+" and "-" in the aggregation formula by any user defined expression using same CubePL syntax.

Any such expression should be a formula of "arg1" and "arg2" parameters. E.g. `max(arg1, arg2)`.

To specify an expression for an aggregation operator one uses the call `cube_metric_set_aggr_expression(me` where operator can be `CUBE_METRIC_AGGR_PLUS` or `CUBE_METRIC_AGGR_MINUS`

1.2.2 Using the C++ library

To create a derived metric in C++ programs, one specifies the expression as the last parameter of the metric definition. This parameter can be omitted, in this case the derived metric returns always 0.

Here is a code example:

```
Metric * met = cube.def_met(  "Derived metric",
                              "derivedmetric1",
                              "DOUBLE",
                              "sec",
                              "",
                              "https://doc.web.org/metrics.html#derived",
                              "Average execution time",
                              NULL,
                              CUBE_METRIC_PREDERIVED_EXCLUSIVE,
                              "${unit}*metric::time(i)/metric::visits(e)",
                              "{ ${unit}=1000; }",
                              "min(arg1,arg2)",
                              ""
                           );
```

`CUBE_METRIC_PREDERIVED_EXCLUSIVE` defines the kind of the derived metric (see '1.3' below).

Derived metric specification allows to define an CubePL expression, which will be executed only once before the actual metric expression is executed. This expression is the last argument in the parameter `{ ${unit}=1000; }`.

Operator "+" is redefined like `min(arg1,arg2)`.

Note, that the CubePL expression should be valid at the time of execution. It means, if there is a reference to metric, this metric should be defined before.

1.3 The three kinds of derived metrics

Cube provides an API to calculate different inclusive and exclusive values aggregated over different dimensions.

In the context of derived metrics it is sometimes useful, when evaluation of the expression is done after the aggregation of the metrics in the expression, sometimes it is needed to aggregate values of the derived metrics.

Therefor Cube provides three kinds of derived metrics:

1. **Prederived exclusive metric** - metric, which value of which expression is observed

as "being stored" inside of cube report and it behaves as a usual exclusive (along call tree) metric.

One specifies this type of metric using constant

`CUBE_METRIC_PREDERIVED_EXCLUSIVE`

For this metric one can redefine only operator "+". Operator "-" will be ignored.

2. **Prederived inclusive metric** - metric, which value of which expression is observed as "being stored" inside of cube report and it behaves as a usual inclusive (along call tree) metric.

   One specifies this type of metric using constant

   `CUBE_METRIC_PREDERIVED_INCLUSIVE`

   For this metric one can redefine both operators, "+" and "-".

3. **Postderived metric** - metric, which expression evaluated only after metrics, which were used inside of the expression, get aggregated according to the aggregation context (along system tree or along call tree).

   If expression doesn't contain references to another metrics, no aggregation is done. In this case the expression gets calculated and its value returned.

   One specifies this type of metric using constant

   `CUBE_METRIC_POSTDERIVE`

   This metric doesn't allow to redefine anyoperators, because calculation of this metric is performed AFTER aggregation within another metric.

2 Reference of CubePL

2.1 Supported calls in CubePL

Here is the table of currently supported calls in CubePL:

Operation	**Explanation**
123.34	a numerical constant
"STRING"	a string constant
-x	negative value of x
()	grouping (priority)
| |	absolute value
+ - / *	arithmetical operations
A^x	A in power of x
> < >= <= != ==	numerical comparison
and or xor not	boolean operations
eq	string comparison
seq	case insensitive string comparison
=~ /expression/	matching of regular *expression*
sin(x) cos(x) asin(x) acos(x) tan(x) cot(x) atan(x) acot(x)	trigonometrical functions
ln(x)	natural logarithm of x
exp(x)	natural exponent of x
sqrt(x)	square root value of x
random(x)	random number from 0 to x
pos(x)	returns x, if x>, or 0
neg(x)	returns x, if x<0, or 0
sgn(x)	returns -1, if x<0, 0 if x is 0 and 1 is x>0
abs(x)	returns absolut value
ceil(x)	returns smallest integer value, which is bigger than x
floor(x)	returns biggest integer value, which is smaller than x
min(x,y)	returns smaller value of x and y
max(x,y)	returns bigger value of x and y

uppercase(x)	returns uppercase version of x
lowercase(x)	returns lowercase version of x
env("x")	returns a value of the environment variable x
local(var)	Declares (to compile time) a variable *var* as local. It cannot be redeclared later
print(expression)	Prints out in to the standard output the value of *expression*.
global(var)	Declares (to compile time) a variable *var* as global. It cannot be redeclared later
${var}	0th element of the variable with name *var*
${var}[x]	x-th element of the variable with name *var*
sizeof(var)	number of elements in the variable *var*
defined(var)	returns true (1), if the variable *var* is defined, otherwise it returns false (0)
metric::[uniq_name](i|e|∗, i|e|∗)	context sensetive value of metric *uniq_name*
metric::context::[uniq_name](i|e|∗, i|e|∗)	context sensetive value of metric *uniq_name*
metric::fixed::[uniq_name](i|e|∗, i|e|∗)	context independend value of metric *uniq_name*
metric::call::[uniq_name](c_id, i|e|∗, s_id, i|e|∗)	value of metric *uniq_name* for the call path with id *c_id* and system resource with an id *s_id*
cube::metric::set::[uniq_name]("X", "Y")	sets a property X of a metric *uniq_name* to value Y. Corrently supported are only "value" with value "VOID" or else
cube::metric::get::[uniq_name]("X")	returns a property X of a metric *uniq_name*. Corrently supported are 1. *unique name* - returns unique name of a metric *uniq_name* 2. *display name* - returns display name of a metric *uniq_name* 3. *uom* - returns unit of measurement of a metric *uniq_name* 4. *dtype* - returns data type of a metric *uniq_name* as a string 5. *url* - returns url of online help for the metric *uniq_name* 6. *description* - returns description of a metric *uniq_name* 7. *value* - returns property "value" for the metric *uniq_name*. Its value "VOID" deactivates metric and it returns always a zero.

cube::metric::get::[uniq_name]("X", "Y")	Extended version of cube::metric::get::. Returns a property X of a metric *uniq_name* with its value "Y". Currently supported values for X are 1. *attribute* - returns value of the attribute Y.
cube::callpath::get(id, "X", "Y")	Returns a property X of callpath with "id" with its value "Y". Currently supported values for X are 1. *attribute* - returns value of the attribute Y.
cube::region::get(id, "X", "Y")	Returns a property X of region with "id" with its value "Y". Currently supported values for X are 1. *attribute* - returns value of the attribute Y.
cube::stn::get(id, "X", "Y") cube::locationgroup::get(id, "X", "Y") cube::location::get(id, "X", "Y")	Returns a property X of a system tree node, or of a location group or of a location with "sysid" with its value "Y".
cube::metric::set::[uniq_name]("X", "Y")	Sets the property X of a metric *uniq_name* with the value "Y". Currently supported values for X are 1. *attribute* - returns value of the attribute Y.
cube::callpath::set(id, "X", "Y")	Sets the property X of callpath with "id" with its value "Y". Currently supported values for X are 1. *attribute* - returns value of the attribute Y.

cube::region::set(id, "X", "Y")	Sets the property X of region with "id" with its value "Y". Currently supported values for X are 1. *attribute* - returns value of the attribute Y.
cube::stn::set(id, "X", "Y") cube::locationgroup::set(id, "X", "Y") cube::location::set(id, "X", "Y")	Sets the property X of a system tree node, or of a location group or of a location with "sysid" with its value "Y". Currently supported values for X are 1. *attribute* - returns value of the attribute Y.

3 Syntax of CubePL

3.1 Syntax step by step

This chapter introduced the CubePL syntax of the most common programming structures.

3.1.1 Expressions

3.1.1.1 Constants

Any constant expression is an `expression`. eg. 123.0, "someString"

3.1.1.2 Arithmetical Expressions

Common mathematical notation of an arithmetical expression is valid in CubePL. Any arithmetical expression is an `expression`. eg. 123.0 + 23.6, sin(23)

3.1.1.3 Boolean Expressions

Common literal notation of a boolean expression is valid in CubePL. Any boolean expression is an `expression`.

```
[expression] or|and|xor [expression]
```

or

```
not [expression]
```

Any `expression` can be a term of a boolean expression. Non zero value is observed as `TRUE`, zero value is `FALSE`.

3.1.1.4 Function call expressions

Function calls have the following syntax:

```
name ( [expression ] )
```

3.1.2 Lambda function (In-place function definition)

To define a function in-place, one uses the following syntax:

```
{
  [statement];
  [statement];
  ...
  return [expression];
}
```

In-place definition of a function is an `expression`. It means, it can appear everywhere, where one can use an `expression`.

3.1.3 Control structures

Control structures like *if-else* or *while* are `statements`.

3.1.3.1 Condition IF-ELSE

One can execute series of statements under a condition using the short form of the `if` statement:

```
if ( condition )
{
 [statement];
 [statement];
 ...
}
```

or full form with `else` :

```
 if ( condition )
 {
  [statement];
  [statement];
  ...
 }
 else
{
  [statement];
  [statement];
  ...
 }
```

`condition` is a boolean `expression`

3.1.3.2 Loop WHILE

One can execute a series of statements as long as a condition is true using the `while` statement:

```
while ( condition )
{
 [statement];
 [statement];
 ...
}
```

Sequence of statements will be executed till condition is not fulfilled, max 1000000000 times.

3.1.4 Variables

CubePL allows to work with memory, by using variables. All variables are multi-typed:

- In string context value of a numeric variable is presented in string format.
- In numerical context a string variable is converted to its numerical representation, if possible. Otherwise it is 0.

All variables are arrays. Indexless access to the variable assummes mutually index value 0.

3.1.4.1 User defined Variables

The user can define a variable using the syntax:

```
${ name } = [expression];
```

or

```
${ name }[index] = [expression];
```

Currently its name is a fixed string of characters. In later versions of CubePL it will allow any `expression`.

Example of a string context :

```
${name} seq "STRING"
```

Example of a numeric context :

```
${name} >= 0.34
```

Example of an array access to the variable :

```
${name}[ ${i} ] >= 0.34
```

Access to the variable is an `expression`.

Index for the access to the value is also an `expression`.

3.1.4.2 Predefined Variables

Cube provides a set of predefined variables for every calculation, which values are independend. Following predefined variables do contain the general information about the cube:

Predefined variable	Explanation
cube::#mirrors	Number of mirrors in cube file
cube::#metrics	Number of metrics in cube file
cube::#root::metrics	Number of root metrics in cube file
cube::#regions	Number of regions in cube file
cube::#callpaths	Number of call paths in cube file
cube::#root::callpaths	Number of root call paths
cube::#locations	Number of locations in cube file
cube::#locations::void	Number of void locations in cube file
cube::#locations::nonvoid	Number of nonvoid locations in cube file
cube::#locationgroups	Number of location groups in cube file
cube::#locationgroups::void	Number of void location groups in cube file
cube::#locationgroups::nonvoid	Number of nonvoid location groups in cube file
cube::#stns	Number of system tree nodes in cube file
cube::#rootstns	Number of root system tree nodes in cube file
cube::filename	Name of the cube file

CubePL engine defines a set of variables, which do depend on an index, their "id". Using call `sizeof(...)` one can run over them and inspect within CubePL expression.

cube::metric::unit::name	Unique name of the metric
cube::metric::disp::name	Display name of the metric
cube::metric::url	URL of the documentation of the metric
cube::metric::description	Description of the metric
cube::metric::dtype	Data type of the metric
cube::metric::uom	Unit of measurement of the metric
cube::metric::expression	CubePL expression of the metric
cube::metric::initexpression	CubePL initialisation expression of the metric
cube::metric::parentd::id	ID of the parent of the metric
cube::metric::#children	Number of children in the metric

cube::callpath::mod	Module of the call path
cube::callpath::line	File line of the call path
cube::calleeid	ID of the callee region
cube::callpath::#children	Number of children in the call path
cube::callpath::parent::id	ID of the parent callpath

cube::region::name	Name of the region
cube::region::mangled::name	Mangled name of the region
cube::region::paradigm	Name of the paradigm of the region
cube::region::role	Name of the role the region is playing within the paradigm
cube::region::url	URL with the description of the region of the call path

cube::region::description	Description of the region of the call path
cube::region::mod	Module of the region of the call path
cube::region::begin::line	Begin line of the region of the call path
cube::region::end::line	End line of the region of the call path

cube::stn::name	Name of the system tree node
cube::stn::class	Class of the system tree node
cube::stn::description	Description of the system tree node
cube::stn::#children	Number of children (other system tree nodes) of the system tree node
cube::stn::#locationgroups	Number of location groups of the system tree node
cube::stn::parent::id	ID (among all system tree nodes) of the parent of the system tree node
cube::stn::parent::sysid	ID (global) of the parent of the system tree node

cube::locationgroup::name	Name of the location group
cube::locationgroup::rank	Rank of the location group
cube::locationgroup::type	Type of the location group
cube::locationgroup::void	Is the this location group void?
cube::locationgroup::#locations	Number of locations of the location group
cube::locationgroup::parent::id	ID (amoung all system tree nodes) of the parent system tree node
cube::locationgroup::parent::sysid	ID (global) of the parent system tree node

cube::location::name	Name of the location
cube::location::rank	Rank of the location
cube::location::type	Type of the location
cube::location::void	Is the this location void?
cube::location::parent::id	ID (among all location groups) of the parent location group
cube::location::parent::sysid	ID (global) of the parent location group

CubePL engine sets a series of context sensetive variables, which value depends on the paramaters, for which the derived metric is being calculated. Their value can be used to refer to the values of context insensitive variables described above:

calculation::metric::id	ID of the metric, being calculated
calculation::callpath::id	ID of the callpath, for what is the metric being calculated
calculation::callpath::state	State of the callpath (inclusive=0, exclusive=1, same=2), for what is the metric being calculated
calculation::callpath::#elements	Number of selected callpath, for which is metric being calculated. Usually bigger than 1 only for postderived metrics

calculation::region::id	ID of the region, for what is the metric being calculated
calculation::region::#elements	Number of selected regions, for which is metric being calculated. Usually bigger than 1 only for postderived metrics
calculation::sysres::id	ID (local within system resource type) if the sysem resource, for what is the metric being calculated
calculation::sysres::sysid	ID (global) if the sysem resource, for what is the metric being calculated
calculation::sysres::kind	Type of the system element: • `0 = unknown` • `5 = system tree node` • `6 = location group` • `7 = location`
calculation::sysres::#elements	Number of selected system tree elements, for which is metric being calculated. Usually bigger than 1 only for postderived metrics

3.1.5 Different ways to refer an another metric

3.1.5.1 Context sensitive reference to another metric

To use values of another metric in the same calculation context, one uses syntax:

```
metric::[uniq_name]( modificator, modificator )
```

or

```
metric::[uniq_name]( modificator)
```

or

```
metric::[uniq_name]()
```

There are three version of this call:

1. with two arguments (call path and system);
2. with one argument (call path);
3. with no argument(an arguments takes as '$*$').

`modificator` specifies flavor of the calculation: `i` - inclusive, *e* - exclusive, $*$ - same like in calculation context.

Metric reference is an `expression`.

3.1.5.2 Context insensitive reference to another metric

To use values of another metric in the some fixed calculation context (e.g. aggregated over threads), one uses syntax:

```
metric::fixed::[uniq_name]( modificator, modificator )
```

or

```
metric::fixed::[uniq_name]( modificator)
```

or

```
metric::fixed::[uniq_name]()
```

There are three version of this call:

1. with two arguments (call path and system);
2. with one argument (call path);
3. with no argument(an arguments takes as '∗').

`modificator` specifies flavor of the calculation: `i` - inclusive, *e* - exclusive, ∗ - same like in calculation context.

Metric reference is an `expression`.

3.1.5.3 Direct reference to another metric

To use values of another metric with an specific call path id and system resource id (!), one uses syntax:

```
metric::call::[uniq_name]( callpath id, modificator, sysres id, modificator )
```

or

```
metric::call::[uniq_name]( callpath id, modificator )
```

There are two version of this call:

1. with four arguments (call path and system); - Only callculation of an inclusive or exclusive value is performed
2. with one argument (call path); - Aggragation over system tree is performed additionally to the calculation of the inclisive or exclusive value for the calltree id

`modificator` specifies flavor of the calculation: `i` - inclusive, *e* - exclusive.

Metric reference is an `expression`.

Note that "sysres id" is a global identificator and can be refered using ${calculation::sysres::id}.

3.1.5.4 Definition of an encapsulated calculation within CubePL expression using metrics of Cube.

One special mechanis of CubePL processing engine allows some level of calculation separation. A derived metric can be created within another CubePL expression. Such "ghost" metric gets its name and properties and exists inside of the cube object as a casual metric. Only difference to the casual metric is in the fact that ghost metric is not visible in GUI and tools and is not stored inside of the metric tree of the cube file.

One can refer such metric as a casual metric using metric references (see '3.1.5.1' , '3.1.5.2' and '3.1.5.3').

Example for definition of such metric :

```
cube::metric::nvisitors(e)
<<
{
  ${return}=0;
  if ( ${cube::locationgroup::void}[${calculation::sysres::id}]  != 1)
  {
    if (metric::visits()>0 )
    {
      ${return} = 1;
    };
  };
  return ${return};
}
>>;
${visitors} = metric::fixed::nvisitors(e);
```

where

```
cube::metric::prederived::nvisitors(e)
```

gives a type (prederives, exclusive) and unique name (nvisitors) of this metric. Unique name is used then later to refer to this metric via

```
${visitors} = metric::fixed::nvisitors(e);
```

There are kinds of metrics, which can be defined on such manner:

1. `cube::metric::prederived::name(e)` - An exclusive prederived metric with the name `name`;
2. `cube::metric::prederived::name(i)` - An inclusive prederived metric with the name `name`;
3. `cube::metric::postderived::name` - A postderived metric with the name `name`.

Notice that once the metric with some unique name created it exists whole lifetime of the cube object. Therefore one can refer to some somewhere previously defined ghost metric from any following it CubePL expressions.

Ghost Metric definition is a `statement`.

3.1.5.5 Definition of an initialization expression for ghost metrics within CubePL expression.

One can specify within of a CubePL expression an initialisation phase for previously created ghost metric. For that purpose one uses expression `cube::init::metric::[name]`. Notice that named metric should be know by the moment of compilation of the CubePL expression.

Example for definition of such metric :

```
cube::init::metric::init::nvisitors
<<
{
  global(nvisitors);
}
>>;
```

where

```
cube::init::metric::nvisitors
```

uses an unique name (nvisitors) of the metric.

Definition of the initialisation phase of a metric is a `statement`.

3.1.5.6 Definition of an initialization expression for ghost metrics within CubePL expression.

One can specify within of a CubePL expression an aggregation operator "+" or "-" for previously created ghost metric. For that purpose one uses expression `cube::metric::plus::[name]` or `cube::metric::minus::[name]`. Notice that named metric should be know by the moment of compilation of the CubePL expression.

Example for definition of such metric :

```
cube::init::metric::plus::nvisitors
<<
max( arg1, arg2)
>>;
```

or

```
cube::init::metric::minus::nvisitors
<<
min( arg1, arg2)
>>;
```

where

```
cube::init::metric:plus::nvisitors
```

or

```
cube::init::metric:minus::nvisitors
```

uses an unique name (nvisitors) of the metric.

Definition of the initialisation phase of a metric is a `statement`.

3.2 Grammar of CubePL

Here will be a full grammar of CubePL expressions (later).

4 Examples of CubePL expressions

4.1 Simple Examples

1. Calculation of an arithmetical expression

```
123.4 + 234 -( 23)^2
```

2. Calculation of an arithmetical expression with different functions

```
sin(23 + ln(12))
```

4.2 Complex Examples

4.2.1 Different made-up expressions

1. Definition of a constant function

```
{ return 24; }
```

2. Definition of a more complex function

```
{ return sin ({ return 1; }); }
```

3. Definition of a function with an access to one variable

```
{${a}=123; return ${a}; }
```

4. Definition of a function with a control structure

```
{
 ${a}=metric::visits();
 ${b}=0;
 if (${a}>100)
   { ${b}=metric::time(); };
return ${b};
}
```

5. Definition of a function with a loop structure

```
{
 ${a}=0; ${b}=0;
 while (${a}<123)
  {
     ${b}=${b}+metric::time();
     ${a}=${a}+1;
  };
return ${b};
}
```

6. Definition of a function with a access to predefined variable

```
{
 ${a}=0;
```

```
 if (${calculation::region::name}[${calculation::callpath::id}] =~ /^MPI_/)
  {
     ${a}=metric::time();
  };
return ${a};
}
```

4.2.2 Special metrics of Scalasca

1. Calculation of an average runtime of a call path (Kenobi metric, postderived)

```
metric::time(i)/metric::visits(e)
```

2. Calculation of the time, spend in MPI synchronization calls

   a) (initialization)

```
{
    global(mpi_synchronization);
    ${i}=0;
    while( ${i}  < ${cube::#callpaths} )
    {
        ${mpi_synchronization}[${i}] = 0;
        ${regionid} = ${cube::callpath::calleeid}[${i}] ;
        if (
            (${cube::region::paradigm}[ ${regionid} ] seq "mpi")
            and
            (
            (${cube::region::name}[${regionid} ] seq "mpi_barrier" )
            or
            (${cube::region::name}[${regionid} ] seq "mpi_win_post" )
            or
            (${cube::region::name}[${regionid} ] seq "mpi_win_wait" )
            or
            (${cube::region::name}[${regionid} ] seq "mpi_win_start" )
            or
            (${cube::region::name}[${regionid} ] seq "mpi_win_complete" )
            or
            (${cube::region::name}[${regionid} ] seq "mpi_win_fence" )
            or
            (${cube::region::name}[${regionid} ] seq "mpi_win_lock" )
            or
            (${cube::region::name}[${regionid} ] seq "mpi_win_unlock" )
            )
            )
        {
                ${mpi_synchronization}[${i}] = 1;
        };
        ${i} = ${i} + 1;
    };
    return 0;
}
```

   b) (actual calculation)

```
{
    ${a}=0;
    if ( ${mpi_synchronization}[${calculation::callpath::id} ]== 1 )
    {
            ${a} = metric::time(*,*)-metric::omp_idle_threads(*,*);
    };
    return ${a};
}
```

3. Calculation of thecComputational load imbalance (single participant)

   – NO EXAMPLE YET –





www.scalasca.org