CALIPER

Enabling introspection of MPI libraries through the MPI_T interface

Authors:
Srinivasan Ramesh

August 18, 2017

Introduction

MPI libraries today involve multiple components interacting in complex ways to affect performance. Together with the heterogeneous nature of current and future architectures, this means default MPI library settings may not always be optimal for performance and scalability — significant performance gains may be achieved by tailoring MPI library behaviour to suit application characteristics. To understand MPI library internals, there needs to be a way to introspect the MPI_T libary at runtime. In order to modify MPI library behaviour dynamically at runtime, the library must expose a means to do so.
The MPI_T interface, introduced in the MPI 3.0 standard provides external tools an opportunity to introspect and potentially modify MPI library behaviour at runtime by means of two semantics:

- Performance Variables (PVARs): Performance variables represent MPI internal information in the form of counters, state, watermarks, etc. The MPI specification details the various classes of PVARs supported, allowed datatypes and access semantics each of class.
- Control Variables (CVARs): Control variables are the means by which an external tool can modify MPI library behaviour and fine-tune application performance. They are essentially knobs that may represent the value of a particular setting inside the MPI library.

Caliper is an application introspection tool that relies on source code annotations to collect information and perform profiling related tasks. Caliper *services* are the basic building blocks that can be combined freely to realize advanced profiling / tracing capabilities. The *MPI* service utilizes the PMPI interface to profile MPI library calls. This document describes the design of the *MPIT* service that performs MPI library introspection through the MPI_T interface. The motivation behind this document is to describe the rationale that went into the design of the service. Caliper is in active development — this document deliberately avoids description of code or filenames

used to implement the *MPIT* service. However, any design modifications to the service shall be reflected here.
We shall touch upon some Caliper concepts when required. For detailed description of Caliper design, kindly refer to Caliper documentation.

Collecting PVARs through MPI_T

0.1 Creating a performance session

In order to use MPI_T, a tool must first create a *performance session* and associate *handles* for the performance variables it wishes to read. Performance sessions allow the MPI library to distinguish between multiple tools / software modules that may be simultaneously querying the MPI_T interface.
The *MPIT* Caliper service creates an MPI_T performance session during the Caliper service registration phase.

0.2 PVAR handle allocation

Before a tool can read the value of a PVAR, it must first allocate a *handle* for the PVAR. The MPI_T interface specifies a function that allows a tool to know the number of PVARs exported by an MPI implementation at any given point in time. A couple of points need to be kept in mind when allocating PVAR handles:

- Number of PVARs can change: The number of PVARs exported by the library can change at any point in time. Typically, MPI libraries export additional PVARs after `MPI_Init`. Caliper allocates handles for PVARs from multiple places: During the registration for the *MPIT* service, inside the wrapper for `MPI_Init`, and from inside the wrapper for certain MPI calls (description follows).

- PVARs can be bound to MPI objects: The `MPI_pvar_get_info` function returns the *bind* type for the PVAR. The idea here is that PVARs

can be *associated* with a specific object such as a communicator or message. As a result, there can be multiple handles allocated for a PVAR at any given index. These handles must be allocated appropriately depending on the bind type.

- `MPI_T_BIND_NO_OBJECT`: These PVARs are not bound to any MPI object — Caliper allocates handles for such PVARs during Caliper registration and inside the wrapper for `MPI_Init`.
- `MPI_T_BIND_MPI_COMM`: These PVARs are bound to MPI communicators, and is a special case. Handles for pvars bound to `MPI_COMM_WORLD` and `MPI_COMM_SELF` are allocated during Caliper registration phase. Additionally, handles are created each time `MPI_Comm_create` is invoked, by intercepting the call through the PMPI wrapper.
- `MPI_T_BIND_WIN`: These PVARs are bound to MPI windows. As a result, handles for such pvars are allocated inside the Caliper PMPI wrapper for `MPI_Win_create`.
- `MPI_T_BIND_MPI_ERR_HANDLER`: These PVARs are bound to MPI error handlers. Handles for such pvars are allocated inside the PMPI wrapper for `MPI_Errhandler_create`.
- `MPI_T_BIND_MPI_FILE`: These PVARs are bound to file objects. Handles are allocated inside the PMPI wrapper for `MPI_File_open`.
- `MPI_T_BIND_MPI_GROUP`: These PVARs are bound to MPI group objects. Handles are allocated inside the PMPI wrapper for `MPI_COMM_GROUP`.
- `MPI_T_BIND_MPI_OP`: These PVARs are bound to MPI reduction operators. Handles are allocated inside the PMPI wrapper for `MPI_Op_create`.
- `MPI_T_BIND_MPI_INFO`: These PVARs are bound to MPI Info objects. Handles are allocated inside the PMPI wrapper for `MPI_Info_create`.
- `MPI_T_BIND_MPI_MESSAGE`, `MPI_T_BIND_MPI_REQUEST`: Not supported inside Caliper currently. Open question — how do we allocate handles for these?

0.3 PVAR classes and notion of aggregability

Depending on what they represent, PVARs are categorized into counters, state variables, watermarks, etc., and are handled differently. For this pur-

pose, we define the notion of aggregatability as follows: Any PVAR on which it is *meaningful* to apply one or more of (SUM, MAX, MIN, AVG, COUNT) operators is defined as aggregatable.
Along with other information, a call to `MPIT_pvar_get_info` returns the *CLASS* to which the PVAR belongs. The various classes, along with how Caliper handles them are:

- `MPI_T_PVAR_CLASS_TIMER`, `MPI_T_PVAR_CLASS_AGGREGATE`, `MPI_T_PVAR_CLASS_COUNTERS`: These are free-counting, monotonically increasing values. As such, they are not aggregatable, but by storing the "last" value for these counters and timers, the difference between the current and last value is a derived metric that is aggregatable by use of *SUM*, *MAX*, *MIN*, *AVG* operators. Storing this difference is more useful than just the raw counter values, as one would typically by interested in the *change* caused to any of these PVARs rather than the raw value itself.

- `MPI_T_PVAR_CLASS_STATE`: Represents MPI state at any instant in time. Non-aggregatable value.

- `MPI_T_PVAR_CLASS_SIZE`: Represents size of an MPI resource. Non-aggregatable value.

- `MPI_T_PVAR_CLASS_LEVEL`, `MPI_T_PVAR_CLASS_PERCENTAGE`: Represents the instantaneous level or percentage utilization of an MPI resource. It is meaningful to apply the *AVG*, *MIN*, *MAX* operators, and hence these classes are aggregatable.

- `MPI_T_PVAR_CLASS_HIGHWATERMARK`, `MPI_T_PVAR_CLASS_LOWWATERMARK`: As such, they are non-aggregatable. However, one can define aggregatable derived metrics out of these PVARs. Specifically, Caliper defines two derived metrics: A boolean that tells us if the watermark has gone up from the last time it was read, and a double value specifying the *change* in the value between successive reads. Both of these derived metrics are aggregatable quantities as one can apply the *COUNT* and / or *SUM* operator to them.

- `MPI_T_PVAR_CLASS_GENERIC`: Represents PVARs that do not fall into any of the above classes. These PVARs would need to handled on a case-by-case basis, and thus for now, we define these as non-aggregatable values.

0.4 Creating Caliper attributes for PVARs

The basic data unit in Caliper is an attribute. An attribute is a key value pair that has certain properties. For each PVAR exposed by the MPI library, Caliper defines an attribute with the same name as the PVAR. Each PVAR attribute has the following properties:

- `CALI_ATTR_AS_VALUE` - We do not want "stacking" for PVAR values. They should be treated much the same way as PAPI counters.
- `CALI_ATTR_SCOPE_PROCESS` - PVARs are defined on a per-rank basis
- `CALI_ATTR_SKIP_EVENTS` - We do not want callbacks to be triggered everytime the attribute for a PVAR is updated
- `Metadata (class.aggregatable)` - Boolean value specifying if the PVAR is aggregatable or not. Aggregatability is determined in accordance with the rules above.

Apart from creating a Caliper attribute for each PVAR exported, there are two additional attributes created for each watermark PVAR exported — one that represents the number of times the watermark changes, and another that represents the cumulative change in the watermark PVAR.

0.5 Querying and storing PVARs

In the current design, all PVARs exported by the MPI library are queried when a snapshot is triggered. By integrating the *MPIT* service along with the *MPI* service, this would be useful in determining how various MPI function calls contribute to changes in PVAR values. Moreover, one can gather meaningful information by aggregating using MPI function names or annotated code regions as keys. Currently, we note about 10-15% overheads in collecting PVARs during every snapshot event. This may not be a very scalable option, as this overhead would increase with a rise in number of PVARs exported.
Depending on the class of the PVAR, we either store the raw value read from the interface in the snapshot, or a derived metric.

- `MPI_T_PVAR_CLASS_TIMER`, `MPI_T_PVAR_CLASS_AGGREGATE`, `MPI_T_PVAR_CLASS_COUNTERS`: We store the difference between the current value and the "last value" for such PVARs in the snapshot. Storing and aggregating this derived value is more meaningful — it

helps us answer questions such as: How do different MPI functions contribute to this PVAR? Which MPI function is responsible for the highest value?

- `MPI_T_PVAR_CLASS_STATE`, `MPI_T_PVAR_CLASS_SIZE`: These PVARs are stored as is in the snapshot. Perhaps it may be more meaningful to view changes *over time*, such as in a trace.

- `MPI_T_PVAR_CLASS_HIGHWATERMARK`, `MPI_T_PVAR_CLASS_LOWWATERMARK`: Along with storing the raw value for watermark PVARs, we store the derived metrics that represent the number of times the watermark changed, along with how much the watermark changed in the snapshot. By aggregating across MPI functions for example, we can answer questions such as: Which function most frequently pushed up / down a watermark? Which function was responsible for the highest cumulative change in a given watermark?

- `MPI_T_PVAR_CLASS_LEVEL`, `MPI_T_PVAR_CLASS_PERCENTAGE`: We store these PVARs as is in the snapshot. It maybe meaningful to view the average, maximum or minimum value for these PVARs, aggregated across MPI functions.