Tool Support in HPC++*

"Strawman" Proposal

Lars T. Hansen, Janice Cuny, Allen D. Malony,
Sameer Shende, Kurt Windisch
University of Oregon

February 26, 1996

1 Motivation

The purpose of this document is to define a "strawman" proposal for a tool interface to a running HPC++ program. It is our intention that the proposal serve as a starting point for further discussions with language implementers and tool developers, eventually resulting in a tool interface which is part of the HPC++ language definition.

A standard tool interface can greatly aid the development and portability of tools. The tool interface reveals selected parts of the run-time system and compiled programs to the tool, allowing the tool to inspect the running program at a level below that of the source program but with language-level information still available. For example, the tool interface might allow the tool access to descriptors for distributed arrays.

At this point we are not addressing the needs of all tools, concentrating instead on a simpler initial proposal which can support tools like simple monitors (e.g. on-line profilers) and state accessors (e.g. visualizers).

There is a danger that the requirements of the tool interface overburden language implementors by making assumptions about the language implementation that will not be true on all architectures. We therefore need some form of interaction with the language implementors in order to work out a reasonable tool interface.

In this paper we propose an event model, a small set of events, and a small set of instrumentation commands to "test the waters" and form a basis for further discussion. The proposal is based on our experience with a similar system for pC++.

We are leaving out anything having to do with startup, shutdown, attaching, or detaching of the tools to the running program at this point. We are also leaving out any mention of support for multiple concurrent tools, as we are actively working on understanding how this should be done. The design notes for our pC++-based system cover these issues in some detail.

It is our intention that this proposal be compatible with standard distributed object systems like CORBA and ILU, and that these systems can serve as implementation vehicles for the tool support, if desired.

*This work has been funded by ARPA under U.S. Army contract No. DABT63-94-C-0029, "Programming Environments, Compiler Technology and Runtime Systems for Object Oriented Parallel Processing". Contact {lth,malony}@cs.uoregon.edu.

2 Event Model

The model is client/server-like where the clients are the tools and the server is the parallel program. The messaging semantics are not RPC-like but could be implemented on top of RPC.

The event model uses the HPC++ computational model. The latter is not well defined in the previous (December 1995) draft of the working papers, so we make some assumptions here based on earlier discussions within the working group.[1]

The computation runs on a number of *contexts*, essentially address spaces each with one or more processors allocated to it. Each context has one or more threads of computation, one of which is denoted the *distinguished* thread. Threads which are not the distinguished thread participate only in the execution of parallel loops.

Contexts are grouped by system objects called *collectors*, which in purpose are supersets of the processor maps proposed in section four of the December 1995 draft. Collectors serve as foci for data distribution and barrier synchronization. Each distributed data structure is controlled by a particular collector; it should be possible for the tools to discover the data structure's collector given a pointer to the data structure. A barrier synchronization among the contexts in a collector's context set can be effected by a call to the collector member function `barrier`.

Events occur at those (thread,location) pairs in the running program for which event signalling has been enabled by the client, and the thread suspends its computation. When an event occurs, a notification of this fact is signalled to the client. Events are purely local to a thread and carry local data only. The thread enters a command loop in which it will receive *commands* from the client and execute them. One command is `continue`, which will cause the thread to resume its computation. Events are described in section 3, commands in section 4.

3 Events

All events carry a context ID, a thread ID relative to the context ID, and a time stamp. The context ID and the thread ID are both nonnegative integers. The time stamp is as-yet undefined.

3.1 Collector events

The `create-collector` event is signalled when a collector has been created. The `destroy-collector` event is signalled when a collector is about to be destroyed. They are parameterized by the local address of the collector object.

The `begin-barrier` event is signalled when a thread is about to go into a barrier. The `end-barrier` event is signalled when a thread exits a barrier. The events are parameterized by the local address of the collector.

3.2 Loop events

The `begin-parallel-loop` event is signalled when the program is about to enter a parallel loop. The event is parameterized by the program location of the start of the loop. The `end-parallel-loop` event is signalled when the program is about to exit a parallel loop. The event is parameterized by the program location of the end of the loop.

While the loop is in progress, the `begin-iteration` event is signalled when the loop enters a new iteration, and the `end-iteration` event is signalled when the loop ends an iteration. The events are parameterized by the program location of the start of the loop.

[1] In general the computational model needs to be specified in much greater detail.

While the loop is in progress, the `begin-reduction` event is signalled when a reduction starts, and the `end-reduction` event is signalled when the reduction ends. Both events are parameterized by the program location of the start of the loop and the program location of the start of the reduction statement.

3.3 Function events

The `enter-function` event is signalled when a function is entered. The `exit-function` event is signalled when a function is exited. Both events are parameterized by the memory location of the start of the function.

3.4 Array Events

The `create-array` event is signalled when an array has been created. The `delete-array` event is signalled when an array is about to be destroyed. The events are parameterized by the local address of the array object.

The `change-array` event is signalled when an array has changed in a significant way (redistribution, for example). The event is parameterized by the local address of the array object and a code which details the nature of the change (currently unspecified).

4 Commands

All commands take a context ID and a thread ID. In the following, an event type is one of the events defined in section 3. Commands are executed non-overlapping in the order issued, but asynchronously; we do not yet specify any mechanism for determining that a command which has a reply (`read`, `invoke`) has completed.[2]

The `enable-event` command sets up an event at *all* program location where the event applies. It takes an event type and enables the event for the given context and thread.

The `disable-event` command disables a previously enabled event for the given context and thread. It takes an event type.

The `read` command takes a pointer valid in the given context and a number of bytes and returns the contents of memory at the indicated locations. The read is unsynchronized with any other threads which might be running in the same context.

The `write` command takes a pointer valid in the given context, a number of bytes, and a pointer to client-local memory and overwrites the indicated locations in the target context with the indicated data. The write is unsynchronized with any other thread which might be running in the same context.

The `continue` command takes no arguments and causes the indicated thread to continue execution.

The `invoke` command takes a function address, a client-local pointer to a parameter list structure (not currently defined), and a client-local pointer to a return value structure (not currently defined) and calls the indicated function with the given parameters, storing the return value in the indicated location.

Other commands will be needed: there needs to be a way to access the attributes of an array or descriptor, given a pointer to the array, and there needs to be a way to access the attributes of a collector, given a pointer to the collector.

[2] Our prototype system uses a special `reply` event and allows one outstanding reply for each thread.

5 Discussion

The intention behind using program or data locations as parameters to events is that tools which have access to the symbol table for the program can relate these locations back to the source code for the program.

The current proposal has several problems:

- The current proposal defines event instrumentation to be an *en masse* action; either all points for the events of a given type calls are instrumented or none at all. This is unlikely to be adequate for all types of tools, as the volume of events will be very large. Rather, it should be possible to enable only some event points, based on program location or on data location (for example, barriers can be enabled based either on the location of the call to the barrier function or the location (that is, identity) of the collector on which the barrier function was called.)

  Selective enabling functionality brings up the issue of how the tool discovers where to place instrumentation at run-time. We think this issue requires the attention of the working group, since event points may be hidden in libraries, for example, and even if source code is available it is not always possible to know what the compiler has done to the code without being told by the compiler. To effect a fine-grained event enabling command, therefore, it may be necessary to define some manner in which the tool can discover the locations of event points, probably from the compiler/linker via extended symbol tables.

- Including a time stamp with all events may be wasteful, as it may be expensive to obtain the time stamps. Perhaps a better functionality is to enable events with or without time stamps.

- There are no general breakpoints, and no access to the contents of the CPU registers. Such functionality is certainly necessary for general debuggers, and can be helpful to other tools.

- Our current prototype splits many events into two parts, a `begin` part and an `end` part. For example, there could be a `begin-create-array` and an `end-create-array`. The advantage of this scheme is that event filtering is made simpler in the face of all-or-nothing event instrumentation. We have left it out of the current proposal for the sake of simplicity, and because we anticipate that the event model will be changed to allow selective instrumentation.

- There is no support for data watchpoints.