Language-Based Parallel Program Interaction: The Breezy Approach*

Darryl I. Brown Allen D. Malony
Bernd Mohr

Department of Computer And Information Science
University of Oregon
Eugene, Oregon 97403
{darrylb, malony, mohr}@cs.uoregon.edu

Abstract.

This paper presents a general architecture for runtime interaction with a data-parallel program. We have applied this architecture in the development of the *Breezy* tool for the pC++ language. *Breezy* grants application programs convenient and efficient access to higher-level external services (e.g., databases, visualization systems, and distributed resources) and allows external access to the application's state (e.g., for program state display or computational steering). Although such support can be developed on an ad-hoc basis for each application, a general approach to the problem of parallel program interaction is preferred. A general approach makes tools more portable and retargetable to different language systems.

There are two main conclusions from this work. First, interaction support should be integrated with a language system facilitating an implementation of a model that is consistent with the language design. This aids application developers or the tool builders that require this interaction. Second, as the implementation of *Breezy* shows, the development of interaction support can leverage off the language itself as well as its compiler and runtime systems.

1 Introduction

It is increasingly the case in high-performance parallel applications that interaction with an external computing environment is necessary for a computational problem's solution. Certainly, this has always been true from the point of view of file I/O for reading program input data and writing computation output results, and traditional state-based debugging tools have always had interaction with a halted program as a fundamental feature. However, these interface examples are rudimentary relative to the support required for application programs to access higher-level external services (e.g., databases, visualization systems[1], and distributed resources) and to allow external access to the application's state (e.g., for program state display or computational steering[2]). Although such support can be developed on an ad-hoc basis for each application, a general approach to the problem of parallel program interaction is preferred. In particular, high-level parallel languages require an approach that is portable with language implementations and that can accommodate language-level interaction with external applications and tools.

The notion of program interaction support has long been a concern in the distributed computing and software engineering domains where interactions are either out of necessity or are the basis for improved application design. In a high-performance computing context, program interaction typically implies a performance loss. However, the development of interaction support, when required, will become more problematic as the sophistication of the application and the parallel computing environment increases. For this reason, the inclusion of interaction support early in the design and development of a parallel language system can lead to an integrated solution where external access to a parallel program will be natural and convenient, and where performance concerns, when they arise, can be addressed within the particular computing environment.

*This research is supported by ARPA under Rome Labs contract AF 30602-92-C-0135 and Fort Huachuca contract ARMY DABT63-94-C-0029.

In this paper, we describe the design approach used to implement parallel program interaction support in a parallel object-oriented language system. The unique aspect of this research work is the use of the language itself and its associated compiler resources for generating the interaction infrastructure. The remainder of the introduction describes the features of the interaction system and the language platform where it was developed. More details of the architecture and implementation are given in the following sections. Several examples are then briefly described, followed by future directions and conclusions.

1.1 Breezy and pC++

The program interaction system we developed, *Breezy* (**BR**eakpoint **E**xecutive **E**nvironment for visuali**Z**ation and data **D**ispla**Y**), is a tool that provides the infrastructure for a client application to attach to a data-parallel application at runtime. It creates a partnership between the client application and the parallel program. This partnership gives the client several capabilities.

- The client can control the execution of the program.
- The client can retrieve data from parallel data structures created in the program.
- The client can invoke certain functions (or class methods) in the parallel program.
- The client can retrieve specific information about the program's execution state.
- The client can retrieve meta-information about the program such as type descriptions.
- And, *Breezy* allows general communication between the client application and parallel program.

Here we describe the *Breezy* implementation for the data-parallel language pC++[3]. pC++ is a language extension to C++ designed to allow programmers to create distributed data structures that have parallel execution semantics. The basic concept behind pC++ is the notion of a distributed *collection*, a structured set of objects which are distributed across the processing elements of the computer in a manner designed to be consistent with HPF[4]. To accomplish this, pC++ provides a very simple mechanism to build collections of objects from a base *element* class. Member functions of this element class can be applied to the entire collection (or a subset) in parallel. This mechanism provides the programmer with a clean interface to data-parallel style operations by simply calling member functions of the element class. To help the programmer build collections, the pC++ language includes a library of standard collection classes that may be used direct or subclassed. This includes classes such as `DistributedArray`, `DistributedMatrix`, `DistributedVector`, and `DistributedGrid`. pC++ also includes an environment of tuning and analysis utilities (TAU)[5], of which *Breezy* is a part. This implementation of *Breezy* in pC++ is a concrete example of how the *Breezy* architecture has been applied successfully to a data-parallel language environment.

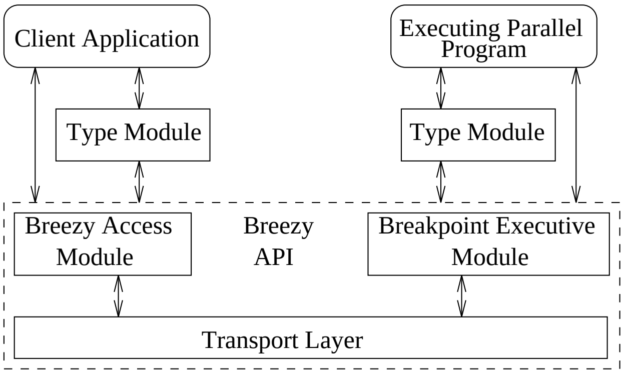

Figure 1: Breezy Architecture

2 Breezy Architecture

Breezy is an architecture that could profitably be reapplied to other data-parallel languages. The *Breezy* architecture consists of several modules (see Fig. 1). The key modules are the *Breakpoint Executive* module and the *Breezy Access* module. The *Type* module and the *Transport Layer* module are components utilized by the Access and Executive modules.

2.1 The Breakpoint Executive Module

The *Breakpoint Executive* module is primarily a request handler. It maintains information about program state, such as current breakpoint location in source code and the list of currently instantiated parallel data objects. For meta-information such as type descriptions of the parallel data structures or lists of all user-defined functions that can be called, the *Breakpoint Executive* module must consult the Type module. To serve requests for parallel data, the *Breakpoint Executive* calls access functions in the executing program. These access functions reside in the (modified) user program in order to have access to the program variables and functions.

2.2 The Breezy Access Module

The *Breezy Access* module is currently implemented as a library of C routines. This library is linked with a client application to give that application access to the *Breezy* API. The following list relates this functionality in detail.

The client can control the execution of the program. The parallel program stops at each synchronization barrier and waits for a request from the client. This request specifies one of the functions described below, or it directs *Breezy* to continue to the next breakpoint. *Breezy* guarantees a consistent state of data in the program by allowing breakpoints only during these barriers.

The client can retrieve data from parallel data structures. The client specifies the variable from the program that holds the parallel data object of interest. If this object is a structured object with fields (such as a class), then the client can further specify a particular field within that structure. The user can retrieve this data from all of the distributed elements of the parallel data object, or from a single element in that object.

The client can call specified user-defined functions (or method invocations on classes) in the parallel program. By prepending function names in the user program with a particular string (e.g. `UserDefined`), the *Breezy* instrumentation process notes these particular functions, and adds code that will make them available to the client during runtime. Note that these can be methods defined on the elements of a parallel object as well as regular global functions.

The client can retrieve specific information about the program state. This includes the current location in the source code where the program has paused, and also a list of variables that are currently instantiated parallel objects. The client can retrieve meta-information about the program. This consists of type information for all the parallel structures, and also the names of user-defined functions that are available to be called.

Lastly, *Breezy* supports communication between the client application and parallel program. This may be desirable for instance if the programmer wants a user-defined function to return a value(s). This can be done in a straightforward manner with a high level communication interface which accesses the transport layer directly, bypassing the *Breakpoint Executive* and the *Breezy Access* module.

One of the functions that might be of less obvious use is the ability to get type information about the parallel data objects in the program. This type information may be of interest in itself, as in a debugger application. Also, the client program can make use of type information to interact with the *Breezy Access* module. Using type information, client applications can be generic, adapting to different parallel programs and data within those programs.

2.3 Retrieving Data In Breezy

One way of using type information is in requesting data. For structures, these type descriptions can be used to specify a particular field of interest. Note that nested structures can be accessed this way also. A small example will help explain. Let's assume *Breezy* gives the client application the following type information:

```
class valAttributes {
      char *color;
      float threeDPosition[3];
}
class simpleElem {
      int i;
      class valAttributes *attr;
      float vals[100][100];
}
```

Assume the client further finds that there is a variable `myDistArray` that is a distributed two-dimensional array of `simpleElem` elements (by retrieving program state information using *Breezy*). *Breezy* would represent such a structure as follows:

```
DistributedArray (simpleElem)
                  myDistArray[20][20];
```

We can now retrieve all of the elements in the variable `myDistArray` or a particular element (by specifying the indices in the distributed array of the particular element). We can also retrieve a specific field of the element(s) by specifying its name.

```
retrieveData "myDistArray", "vals"
```

The above call would retrieve the data pointed to by the `vals` field of the `myDistArray` variable. This specificity is recursive, so we could further grab the `threeDPosition` field of the `attr` field, which is a class itself.

```
retrieveData "myDistArray", "attr",
             "threeDPosition"
```

This returns the values in the float array pointed to by the `threeDPosition` field in the `valAttributes` class. All of these requests could be repeated for a particular element by specifying the index of the element. For example to retrieve element indexed by (4,5):

```
retrieveDataFromElem "myDistArray",
                     "vals", 4,  5
```

3 Breezy Discussion

There are several features that make *Breezy* a unique tool for its purpose in data-parallel computing analysis.

- It has a high level interface, practical and intuitive to use.
- Its modular design allows for reuse of components, and clean substitution of new technologies (such as substituting CORBA/IDL[6] for the transport layer).
- It can be used as is with minimal effort, or it could be built on to achieve much more complex functionality, such as computational steering.
- It allows the programmer to make functions available to be called by the client (via the *Breezy* API), giving the client the power to alter the course of the program or perform specific computations.
- Almost all of the implementation is done *in the target language.*

This last point is particularly interesting because it allows the client application to reference data objects just as they were defined in the program, not at some lower level which the data may have been transformed into by the compiler. Also, a new implementation of *Breezy* is not required for each new architecture that the language system is ported to. Because *Breezy* is implemented in the language, *Breezy* runs on any architecture supported by the language implementation. There is a caveat to this argument in that there is at least one and possibly two necessary modifications that needs to take place in the runtime system for *Breezy* to work. The one necessary change is in the implementation of the synchronization barrier. *Breezy* allows the client to access the program information during these barriers only, to ensure a consistent state in the program. The runtime system must modify the implementation of this barrier function to accommodate *Breezy* by having a single thread of execution call the *Breakpoint Executive* module before entering the barrier. All other threads enter the barrier and wait for the last one before continuing on. While they are there, they serve requests from the one thread that is in the *Breakpoint Executive* module. Thus, another requirement of the runtime system would be active messages (the ability to interrupt other threads to answer requests). This may or may not be implemented in the runtime system. In the case that it is not, it would need to be simulated during these barriers.

3.1 How does Breezy Differ from a Debugger?

Comparing *Breezy* to a traditional parallel debugger of high-performance applications such as gdb or dbx-like debuggers helps illustrate its purpose. *Breezy* operates one layer below a debugger. A debugger does not provide a programmable interface. Since *Breezy* makes the parallel program accessible to the client through an API, it is much more flexible than a debugger. In fact, the first use of *Breezy* was in a simple parallel debugger. The implementation was trivial, basically a GUI built on top of the *Breezy* API.

It also differs from a debugger in that it provides different functionality. It is specifically geared toward parallel data. Thus, only data of this type is known and can be access using *Breezy*, whereas a debugger keeps track of *all* data. *Breezy* streamlines extraction of parallel data and allows interesting interactions with that data.

Breezy also differs from most parallel debuggers in philosophy. Debuggers typically deal with symbol tables and pointers to all the data on each node or thread of execution. Thus, for each thread, a debugger window appears to address the variables in that thread. *Breezy* accesses data *using the language.* The philosophy of *Breezy* is to use the language constructs that exist already to get to parallel data. A single thread using these language-level constructs can access data from all other threads, just as any thread in the program itself would access data from other threads. This is how a single point of control is maintained, while allowing access to data on all nodes.

4 Program Analysis and Instrumentation

This section discusses what happens during the precompilation process of *Breezy*. This process is important in that it describes how the data access is designed and how user functions are made available. As mentioned above, these two important functions of *Breezy* are implemented at the language level; the program analysis and instrumentation is where that implementation takes place. Program analysis is accomplished using a utility called Sage++[7], a compiler toolkit that provides an API for browsing the syntax tree of the program and modifying that tree as desired. Once modified, the new syntax tree can be unparsed to C++ source code, which can subsequently be compiled. To take advantage of *Breezy*, other languages would have to provide similar information about the program either from a compiler toolkit, or from the compiler itself.

The user program is first analyzed to generate type information. This type information is passed on to the instrumentor as well as saved to a separate type description file. This file contains "interesting" types (e.g., types involved in parallel data structures), and will be read by *Breezy* during the initialization phase of the execution to make the type information available at runtime. The second step in the process is instrumentation of the user program.

The instrumentor also makes use of the type information. It must add code to the user program that will allow access to the parallel data objects at runtime. The first step in this process is creating functions that extract the data from the parallel structures. If the distributed elements of the parallel object are instances of a class, then we must add methods to that class to get to that data. In generating these new functions and methods, the instrumentor uses the type information. It then must make a table that correlates the string type name of each interesting data type with the function that accesses (extracts data from) that data type. This table, and others that are created during the instrumentation process, are accessible by *Breezy* at runtime. Note that the access functions and methods have fixed argument types so that the access function table entries need not include the parameter types.

The next operation that the instrumentor must perform is detecting all lines in the code where parallel data objects are created. At each of these points, the instrumentor inserts code which will add the new variable's name, the pointer to that variable, and the type of that variable to a table. At each new allocation of a parallel structure, a new table entry is created, making the new variable available to *Breezy*. Note that at all points where parallel structures are deallocated, there must also be code added which deletes the table entry for the object being deallocated.

The last step the instrumentor takes is to detect user-defined access functions. This consists of searching all function names in the user program for a certain prefix, such as `UserDefined`. As in the previous step, the instrumentor again must construct a table relating the name of the function with a pointer to the function for runtime access.

Thus, by accomplishing these steps, *Breezy* has runtime access to tables from which it can do the following.

- relate a name of a type (or detailed fields within that type) to an access function that can extract the data from that type given a pointer to a variable of that type;
- relate a variable name to its type and to a pointer to that variable's data; and
- relate the name of a user-defined function to the pointer to that function.

Given access to this information at runtime, *Breezy* can forward variable, type and functions names from these tables to the client. In return, the client can specify what it is interested in by using these names as arguments to basic calls in the *Breezy* API. The result is a high-level interface based on the language and the program itself.

5 Example Applications of Breezy

The following are three applications that used *Breezy* as a basis for parallel program interaction. Space limitation limit their descriptions here; more information can be found at [11].

The first use of *Breezy* was a simple parallel debugger[5]. This consisted of building a GUI on top of the *Breezy Access* module. This interface allows basic control of the program and access to data and type information. Using the GUI, the user can choose parallel objects from the program, and specify which elements of those objects are of interest. For elements that are structures, the user can choose a particular field of the structure from a display of the structure type. The data from these selected data structures can be processed in two ways: it can be displayed in a scrollable text window or piped to a separate process.

The next application of *Breezy* was as a utility for extracting data from a specific parallel pC++ program for visualization. The parallel application dealt with objects in three dimensional space. These objects were visualized using a visualization language, VIZ, which is a STk[8] based language designed for building visualization tools and prototyping application specific visualizations.

The latest project applying *Breezy* is a Distributed Array Visualizer Environment (DAVE)[9][10]. DAVE acts as a database front-end to program data and information. DAVE, in turn, relies on *Breezy* to actually retrieve that data. DAVE may have several data analysis/visualization applications available. A user specifies through DAVE's GUI what data is to be retrieved (utilizing information from *Breezy*) and to which of these applications that data is to be sent.

6 Future work

There are currently many projects underway in areas of modifying and extending *Breezy* as well as in using *Breezy* as a tool to build on. The network communication of *Breezy* has been implemented using sockets. A new version of *Breezy* will use CORBA/IDL[6] for its transportation layer. The *Breakpoint Executive* module will be a CORBA compliant object, from which clients can request data from the program. This data will be encoded as IDL structures.

Our research team is currently working to develop program analysis tools for HPF. *Breezy* will be one of the tools that will be incorporated into this HPF environment.

DAVE[9][10] will be extended to deal with CORBA objects and communicate directly with *Breezy* via the CORBA interface.

7 Conclusion

The result of the research presented here is a general architecture for runtime interaction with a data-parallel program. We have applied this architecture in the development of the *Breezy* tool for the pC++ language.

There are two main conclusions from this work. First, when interaction support is integrated with a language system, the opportunity exists to implement a model that is consistent with the language design, aiding application developers or the tool builders that require this interaction. Second, the development of interaction support can leverage off the language itself as well as the compiler and runtime systems to implement it.

References

[1] B. Topol, J. T. Stasko, Integrating Visualization Support Into Distributed Computing Systems, Georgia Institute of Technology, Tech. Rep. GIT-GVU-94-38, Oct., 1994.

[2] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, N. Mallavarupu, Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs, *Proc. Frontiers of Massively Parallel Computation*, pp. 442-429, Feb. 1995.

[3] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, B. Mohr, Implementing a Parallel C++ Runtime System for Scalable Parallel Systems, *Proc. 1993 Supercomputing Conference*, Portland, pp. 588-597, Nov. 1993.

[4] High Performance Fortran Forum. *High Performance Language Specification (Version 1.0).* Rice University, May 3, 1993.

[5] D. Brown, S. Hackstadt, A. Malony, B. Mohr, Program Analysis Environments for Parallel Language Systems: The TAU Environment, *Proc. 2nd Workshop on Environments and Tools For Parallel Scientific Computing*, Townsend, Tennessee, pp. 162-171, May 1994.

[6] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Version 1.2.*

[7] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, B. Winnicka, Sage++: An Object Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools, *Proc. Oonski '94*, Oregon, 1994.

[8] E. Gallesio, *STk Reference Manual*, version 2.1.6. Universite de Nice, Feb. 1995.

[9] "Distributed Array Query and Visualization," *Parallel Tools Consortium Working Document*, December 14, 1994.

[10] Parallel Tools Consortium Working Group on Distributed Array Visualization, URL: http://www.llnl.gov/ptools/.

[11] Breezy: A Tool for Runtime Program Interaction With Data Parallel Programs, URL: http://www.cs.uoregon.edu/paracomp/tau/breezy/.