Dynamic High-Level Scripting in Parallel Applications

Filippo Gioachin, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
gioachin@uiuc.edu, kale@cs.uiuc.edu

Abstract

Parallel applications typically run in batch mode, sometimes after long waits in a scheduler queue. In some situations, it would be desirable to interactively add new functionality to the running application, without having to recompile and rerun it. For example, a debugger could upload code to perform consistency checks, or a data analyst could upload code to perform new statistical tests.

This paper presents a scalable technique to dynamically insert code into running parallel applications. We describe and evaluate an implementation of this idea that allows a user to upload Python code into running parallel applications. This uploaded code will run in concert with the main code. We prove the effectiveness of this technique in two case studies: parallel debugging to support introspection and data analysis of large cosmological datasets.

1 Introduction

When some functionality is missing in an application, and the need for it is realized only while the application is running, the typical solution is to stop the application, modify the source code, recompile, and rerun it. The recompilation process can take a significant amount of time, especially for large applications. This reiterated overhead can significantly reduce user productivity. In the case of parallel applications, there is additional overhead in the resubmission of the modified program to the scheduler queue, where it might stay for a long time before being scheduled again for execution. Moreover, sometimes the source code might not be available, making it impossible to add the desired functionality.

While debugging an application, the user might want to perform some extra checks to ensure the correctness of the data inside the application. This might be required only once, or periodically. While analyzing scientific data, intermediate results can steer the user towards new and unexpected hypotheses. Unforeseen procedures might be needed to prove or disprove these hypotheses. During a long-running simulation, the user might want to steer the simulation by modifying some parameter or internal data structure. For example, she might want to inject new molecules while studying the behavior of an enzyme. In all these situations, it would be convenient to simply write the function needed, upload it to the running application, and use it immediately.

In this paper, we present a semi-automatic solution developed inside the CHARM++ parallel runtime system. First, the programmer incorporates some basic functionality into the application to access and manipulate its data structures. Later, when the application is running on the parallel machine, the user can upload snippets of Python code. This Python code can use the basic functionality present in the application to check or modify its state. Since Python is a high-level scripting language, it can contain control flow statements, allowing great expressiveness. Moreover, Python code is typically more compact than C or Java code, and it is well established that programs written in scripting languages are easier to write than programs written in declarative languages[17, 18]. In the case of a closed-source application, if it is compiled with support for our interface, users can still add new functionality on demand, even if they have no access to the original source code.

This paper proceeds by presenting the different ways in which the CHARM++ application and an inserted Python script can interact. This will be aided by explanatory examples. Following this, two case studies using this interface are described in Section 3 and Section 4. Performance of our implementation is analyzed in Section 5. We compare our approach to other works in the field in Section 6. Final remarks and future work are in the last section.

2 The CHARM++/Python Interface

We integrated the CHARM++ runtime system with the Python interpreter. Since CHARM++ is internally written in C++, we utilized the Python/C API[20] to make the two languages interact and exchange data. As we shall see, this allows the user to write Python code which will be up-

loaded to a running CHARM++ parallel application. Here, the Python code will be executed and it will be able to interact with the main CHARM++ code.

2.1 Background

CHARM++[15] is a popular runtime system for developing parallel applications. Several applications have been developed based on the CHARM++ runtime system. Among these are NAMD[1], a molecular dynamics code and winner of the Gordon Bell award in 2002, OpenAtom[2] for Car-Parrinello ab-initio molecular dynamics, and ChaNGa[12] for cosmological simulations. The combined workload of these applications accounts for more than 15% of the time spent executing jobs on several NSF funded supercomputers in the United States, thus proving its significant presence.

The primary concept in CHARM++ is object virtualization[14]. In CHARM++, the user divides the computation into small objects, called *chares*. These chares are assigned to the available processors by the runtime system itself, thus allowing load balancing[26] and other automatic performance optimizations, such as communication optimization[16].

These chares communicate with each other via asynchronous messages. Messages trigger function calls on the destination chare. These functions are called *entry methods*. The computation performed by an entry method upon receipt of a message depends on the information carried by the message and the internal state of the chare receiving the message. Chares performing the same operation are typically grouped into indexable collections of chares for convenience. The most frequently used collection type is an *array of chares*, or simply *array*, where each element of the collection is indexable with an index of up to six dimensions. Another common type of collection is a *group*, where exactly one element of the collection is present on each processor. The declarations of the chare collections and their entry methods are contained in a *charm interface* file, or simply *ci* file. Figure 1 shows an example of a ci file.

```
module MyCharmModule {
  array [1D] MyArray {
    entry MyArray();
    entry void MyMethod();
    entry void MyMethod2(int);
  }
}
```

Figure 1. Example of a charm interface (ci) file. Definition of a chare array type `MyArray` of dimension one, with its constructor and methods.

Converse Client-Server (CCS)[6] is a communication protocol to allow parallel applications to receive requests from remote clients. This protocol is built into CHARM++ and can be used by any CHARM++ application. It follows the CHARM++ semantics: whenever a request arrives through CCS, a message is generated inside the application, and computation is triggered by the delivery of this message. As such, CCS requests are serviced asynchronously with respect to the rest of the application. If an application desires to use the CCS protocol, it has to associate one or more signature strings to entry methods. This is performed through a function call into the CCS framework. Moreover, at startup (whether it is through batch scheduling or interactive shell), a flag must be passed to the application. This will request that the runtime system opens a socket to listen for incoming connections. The connection parameters are then printed to the standard output. Subsequently, remote clients can send requests to the parallel application using this information. A signature string present in the request will be matched to identify which entry methods should be invoked on the application. The application can perform any kind of operation as a consequence of such a request. Finally, a reply can be returned to the client via the CCS protocol.

2.2 The Interface

We used the CCS protocol as the basic communication mechanism between remote clients and the parallel application, also referred to as server. Upon CCS, we implemented our interface to facilitate the programmer's task to augment the parallel application to interact with uploaded Python code, and to create clients capable of generating Python requests.

A typical control flow for inserting a Python code is illustrated in Figure 2. At the beginning of the execution, as with any other application using the CCS protocol, the server registers a string identifying Python requests (step 1 in the Figure). Subsequently, a remote client can send an *Execute* request containing Python code. The server will, at this point, encapsulate the code into a message and schedule it together with the other messages present in the system. Upon delivery of the message, the server will create a new Python interpreter using the Python/C API, initialize it with some basic information, and execute the user code inside this interpreter (step 2). An ID representing the interpreter will be sent back to the client for later usage (step 3). The client can then probe the server with *Print* request to see if the code running on the server printed anything (step 4 and 5), possibly multiple times. Depending on the client setup, the server can finally be queried for completion of the uploaded code with a *Finished* request (step 6 and 7). In the default configuration, the server destroys the Python interpreter at the end of the Python code. This destruction can be overridden by the client, as we shall see. Notice,

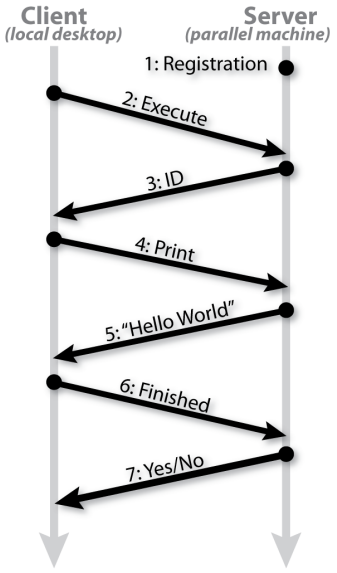

Figure 2. Execution flow of a Python code insertion.

the application does not need to be stopped, and can continue with normal execution. The request is delivered to the application as a message, and is scheduled as any other message present in the application. Therefore, insertion points are between message deliveries. In some scenario, the user might also want to submit more than one request simultaneously.

Each of the three CCS messages—Execute, Print, Finished—is encapsulated by a C++ and Java object. Client programs written in these two languages can use the provided classes to communicate with the parallel application. These classes contain a series of flags that the user can set to modify the behavior of the server. For example, a Request can return an ID for the interpreter as soon as it is created, or wait on the server until the entire Python code has run and return a reply only at that point. In this second case, an Execute request also provides the functionality of a Finished request. The client can be implemented in multiple ways. It can maintain a CCS connection for each Python request and use them to retrieve the prints from different requests concurrently, or maintain a single CCS connection to the server, and periodically probe for prints from the various active Python requests.

As described, the server returns an ID for the interpreter used to serve a particular Execute request. This ID can be used by the client in multiple ways. First, it is needed to retrieve the finish status, or any print generated by the Python code. Second, interpreters can be set to be persistent. In this case, the parallel application will not discard the interpreter at the end of the execution of the Python code. The client can then use this ID to issue a new Execute request on the existing interpreter. In this way, the server will internally maintain the environment set by a previous request, and build upon it. This can be used to upload Python routines and modules, and subsequent requests can contain code using these modules. As we will see in the results, the reuse of an interpreter also has performance benefits.

In the context of CHARM++, all computation is performed inside entry methods, and within the scope of the chare to whom the message was delivered. Dynamically uploaded Python code is not an exception. When the code is uploaded and executed, it runs inside the scope of a chare, determined during step 1 of Figure 2. Notice that during registration either a single chare or a collections of chares can be registered; in the latter case, the same script runs independently in each chare of the collection. In the following section, we will describe three ways in which the Python script can interact with the CHARM++ application: (1) *low-level*, to allow the Python script to perform simple queries on the hosting chare; (2) *high-level*, to allow the Python script to perform more complicated parallel operations on the entire application; and (3) *iterative*, to apply a Python method to a set of objects provided by the hosting chare.

```
module MyPython {
  array [1D] [python] MyArray {
    entry MyArray();
  }
}
```

Figure 3. Definition of a chare array using the Python interface.

2.3 Cross communication

Figure 3 shows the *ci* file for the server definition of a CHARM++ array that can receive Python requests. As can be seen by comparing it with Figure 1, the only addition to a normal definition of a CHARM++ array is the keyword "[python]" in the definition of the chare array MyArray. The other necessary change to the user code is the registration of a string for CCS requests to be identified as Python requests for the chare array MyArray. The registration is a simple function call, made by processor zero, into a registration routine with the string as parameter. These two simple modifications are sufficient to have Python code execute inside interpreters bound to the chare array MyArray. Naturally, if the Python script could not interact with the chare object itself, it would not be very useful. There are three interfaces to allow interaction between the code running in the Python interpreter and the chare object linked to it.

The simplest way for the Python script to interact with the parallel application is through the *ck* module. This Python module is imported into the interpreter before the user script is allowed to run (by executing a default code), and allows Python to query some properties of the CHARM++ environment. These are defined by the system, and include some standard properties, like the processor and the node it is running on, and some specific data about the chare running the interpreter, like the index of the chare inside the collection. In addition, there are two other methods, namely *read* and *write*, through which the Python script can read and write variables with the same access privileges as the containing chare has. The user-defined C++ class (MyArray) inherits these virtual methods from a system-defined C++ class. MyArray can redefine them overriding the default empty behavior.

The *read* method accepts as input parameter a single object, representing where the data should be read from. This object can be a tuple or a list, thus allowing multiple values to be passed in. An example of usage is illustrated in Figure 4 (which will later be described in more detail). Here, we pass a tuple of two values, a string and an integer, as input parameter, and return a single integer as output. To handle input and output from/to Python, the programmer can use the standard Python/C API as well as an extra API provided by our interface (not described here). The *write* method accepts two parameters, one representing where the data should be written to, the other what data to write. It is up to the programmer to define the *read* and *write* methods to correctly interpret the parameters passed as input. A mismatch between definition and usage of these methods generates an exception. If these methods do not give access to some portion of the data, the Python code will not have access to it. This allows some control on what Python can access. Similarly, some data can be made read-only by having it accessible through the *read* methods but not the *write* method.

```
size = ck.read((”numparticles”, 0))
for i in range(0, size):
   vel = ck.read((”velocity”, i))
   mass = ck.read((”mass”, i))
   mass = mass ∗ 2
   if (vel > 1): ck.write((”mass”, i), mass)
```

Figure 4. Python code using only the low-level interface, without the iterate mode.

Nevertheless, the information gathered through the *ck* module is limited to the scope of the processor and the chare that executes the script. If the script requires information generated from a combined operation on all the chares in a collection, say the maximum value of a variable, then another Python module called *charm* can be used. This Python module is constructed to contain all the methods in the *ci* file declared as *python*. The example in Figure 5 shows the method `run` declared as such.

```
module MyPython {
  array [1D] [python] MyArray {
    entry MyArray();
    entry [python] void run();
  }
}
```

Figure 5. Definition of a chare array using the high-level Python interface.

There are two differences between the methods of the *ck* module and those of the *charm* module. The first is that the “python” keyword can be used in conjunction with as many entry methods as needed, thus augmenting at will the set of functions available through the *charm* module. Each of these functions can accept any number of input parameters (the input parameters of the Python calls are always passed by the Python/C API into the C function as a single tuple object), and provide different functionality. The second is that methods of the *charm* module are run inside a user-level thread. This allows the method to issue parallel operations and suspend ifself while waiting for the results. On the other hand, creating a user-level thread has a small but not insignificant cost[27], making the high-level interface slightly more expensive. Notice that parallel operations are initiated by the C++ functions defined in the user class (MyArray), and not by the Python script itself, which can always communicate only with its enclosing chare.

Finally, while the above modules allow the Python script to access the underlying chare and parallel application, there are situations when this is not the best approach. Sometimes a small operation needs to be applied to large sets of homogeneous structures in the parallel application. An example is shown in Figure 4. Here, we want to double the mass of all particles with high velocity, but the user may want to apply many different operations to such particles. One way to solve this problem is by utilizing the previously described interface. The user can write a loop over the desired particles, and by using the low-level or high-level routines, access all the needed information. Each call to `ck.read` and `ck.write` in the script invokes the *read* and *write* methods, respectively, of the user-defined class MyPython. These methods will retrieve/store the information from/to the appropriate locations. Most likely, these locations will be some variable declared inside the MyPython class itself. The other way would be to have the CHARM++ application iterate over the available particles, and call a simple update method with each particle as input. This is

the third method of interaction between the Python script and the CHARM++ application. The user defines two functions in the chare to provide a *begin* and *next* iterator over the particles. CHARM++ uses these two functions to iterate over all the particles, and the user-provided Python method will be applied to each particle. The Python code for this iterative mode is shown in Figure 6. This approach can significantly reduce the complexity of the code that the user has to write (in our example from six lines of code to two), and therefore reduce the probability of making a mistake.

```
def increase(p):
  if (p.velocity > 1): p.mass = p.mass * 2
```

Figure 6. Python code when using iterate mode.

2.4 Error Handling

There are two possible situations in which an error is raised while running an uploaded fragment of code: (1) the uploaded code has an error, (2) the interface code written inside the parallel application is buggy.

In the second case, there is not much that can be done to prevent the application from terminating. While it is possible to capture most signals and errors, the application will likely be in an inconsistent state. If this happens, the application should be corrected.

The case where the uploaded script is erroneous should be tolerated to the extent possible. In our implementation, if the Python code raises an error, this error will be captured and reported to the client as a regular printed string. The user will be able to see the problem, and possibly correct the script and upload a new request with a corrected code. In the implementation presented in this paper, if the erroneous script modified the state of the application before raising the exception, these changes could not be undone automatically. Therefore, it is up to the user to consider what has executed, and take appropriate actions.

We are looking to integrate a live checkpoint-restart scheme, which is currently under development, to expand the coverage of automatically recoverable errors. With this improvement, the state of the application will be saved before executing the Python script, and restored upon failure of the script. Even with the current limitation, we believe that our technique is still valuable. Moreover, an appropriate definition of the atomicity of operations that modify the application (e.g low-level vs. high-level) can help the final user to better recover from mistakes.

3 Case Study: Parallel Debugging

As the first example of a use of our interface, we present CHARMDEBUG[13], a debugger for parallel applications written in CHARM++. While CHARM++ can be run on top of MPI, tools specific to MPI applications, such as TotalView[24], will provide the user with more information regarding the CHARM++ internal implementation, than the user's code. CHARMDEBUG, instead, targets the CHARM++ level. It provides information pertinent to the user, such as the messages queued in the system, the state of a chare, and allows the user to set breakpoints at the beginning of entry methods.

CHARMDEBUG is composed of two modules. The first of these is a graphical tool written in Java. This module can launch parallel CHARM++ applications on remote systems and monitor them. The second module is a plugin inside CHARM++ itself. This module interacts with the parallel application providing the information described above, as well as many other features not described here, to the first module. They communicate over the network, through the CCS protocol.

We built upon the existing CHARMDEBUG system. We used our interface to provide an introspection platform to the user. The user can upload Python code to run only once, after every message processed by the program, or selectively only after a subset of messages. This code can perform checks on the status of the system and identify problems at an early stage. The script is bound to a chare collection selected by the user, and has access to any variable accessible to that chare.

Figure 7 shows a screenshot taken from CHARMDEBUG. On the right side the user can select the entry methods after which the introspection code should be run, or if None is selected, the code will run only once. On the left side, the user can choose the instantiated chare collection that will be hosting the script (at the top), and enter the actual Python code below. Once the code is sent, the CHARMDEBUG plugin inside the parallel application will receive the code, and either execute it immediately and only once, or install it for repeated use. If installed, the code will then automatically be triggered when the specified entry methods are called. If the Python script returns any value other than "None", the parallel application will be suspended, and the user will be notified. He can then use the other views of CHARMDEBUG to inspect the application state in more detail.

As we have seen in Section 2.2, to have a Python script delivered to a particular chare collection, the *ci* file requires that collection be defined as "python". Nevertheless, we did not want to require the programmer to declare every chare collection as "python".

To solve these problems, we used a chare group as target for the Python script. This chare group is part of the

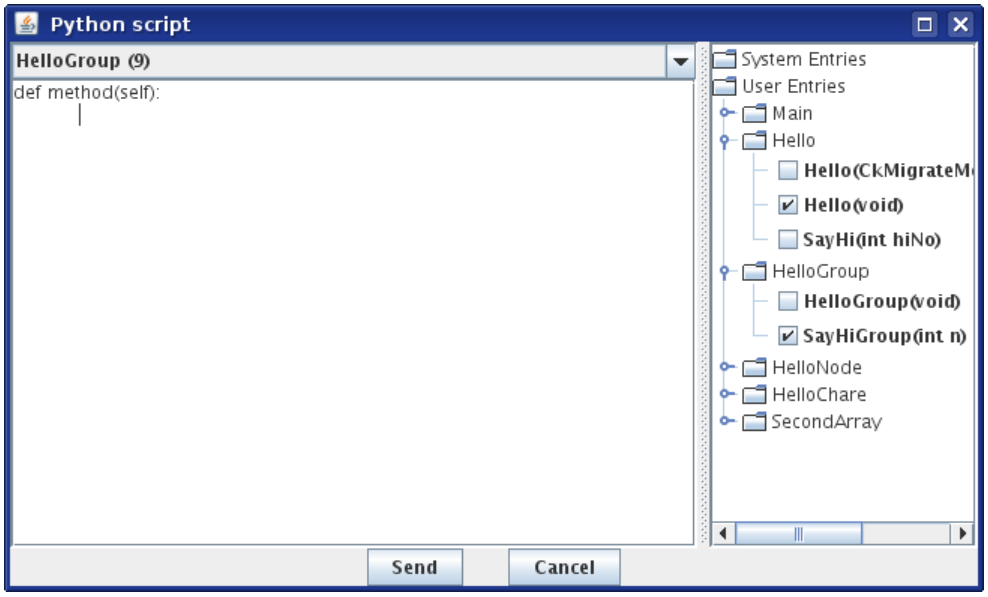

Figure 7. Screenshot of CHARMDEBUG.

CHARMDEBUG plugin module and is called *CpdPythonGroup*. CpdPythonGroup uses the *iterative* method, as described in Section 2.2, to iterate over all the chares in the chare collection selected by the user from the dialog box in Figure 7. For each chare in the collection, the user-specified Python code is executed in conjunction with that chare. This Python code can access a variable inside that chare by using three helper functions. These functions are exported by CpdPythonGroup through the high-level interface into the *charm* module. They are: *getArrayElement*, to browse through arrays, *getValue*, to return a specific field of a data structure, and *getCast*, to perform dynamic casts between objects. All three functions return either opaque objects or simple type objects, such as int or float. Opaque objects can represent any complex data structure in the application, similar to a void pointer in C++. Simple type objects represent primitive data types in C++. To start browsing, the Python code receives an opaque object representing the chare on which it is running as input.

Another major challenge was the fact that while we wanted to have full introspection capability, where the user would be able to read and modify all variables, C++ does not support reflection. This means that at runtime, the application alone cannot identify its data layout. Therefore, the opaque object alone is not enough for the helper functions to provide the desired functionality. Our solution was to require the user to specify the type of every object when calling any of the helper functions. The CHARMDEBUG graphical tool modifies every call to the helper functions, and adds the additional information needed at runtime to browse through the data structures. For example, in a call to getValue, CHARMDEBUG adds to the parameters (1) the type of the resulting value and (2) the offset of the requested field from the beginning of the requested class type. This information is available to CHARMDEBUG since it internally constructs a representation of the class hierarchy of the running application. This representation is needed by CHARMDEBUG for other purposes, therefore its creation is not an overhead.

```
def check(self):
    length = charm.getValue(self, MyArray, len)
    arr = charm.getValue(self, MyArray, data)
    for i in range(0, length):
        value = charm.getArray(arr, double, i)
        if (value > 10 or value < -10):
            print "Error: value ", i, " = ", value
            return i
```

Figure 8. Introspection code to check range of an array.

Figure 8 shows an example of code that can be issued through CHARMDEBUG to perform introspection checks on the running application. Here, we perform a simple check on an array of doubles, to check if their values are between some bounds. Initially, we load the size of the array into the integer value "length" and the opaque value representing the C++ array "data" into the value "arr". Then we loop through all the values in the array, retrieve the i^{th} element, and check if it is within range. If not, we return a value to stop the parallel application.

```
charm.loadSimulation('bhmerger2.3.00006')
charm.createGroup_AttributeRange('blackholes', 'All', 'formationtime', -1e38, -1.0e-38)
print charm.getNumParticles('blackholes', 'star')
```

Figure 9. Example of script uploaded into Salsa. Load a simulation, and print the number of black holes in it (known to have a negative formation time)

4 Case Study: Cosmological Data Analysis

Salsa[21] is another example of an application using the Python/CHARM++ interface. Salsa is part of a larger project called NChilada devoted to the simulation and analysis of cosmological systems. Salsa performs the analysis part. During a typical analysis of a cosmological system, the analyst will use a parallel machine to load the data. He will then proceed iteratively: after having obtained some intermediate results from the analysis, he will decide the next step in the analysis process depending on these intermediate results. In this situation, the flexibility of the system to allow the greatest possible variety of operations is essential for an effective analysis.

A certain number of commonly used operations to analyze a cosmological dataset have been identified. Salsa exports these operations to Python scripts through the high-level interface. Some of these operations include the definition of subsets of particles in the system (grouping), and the definition of new attributes associated with each particle. Moreover, it is possible to iterate over the user created particle groups and apply a user-defined function on both pre-existing attributes and newly created ones.

To be completed, these operations require the collaboration of the entire application. Since they are defined through the high-level interface, they can request the parallel operation and suspend the Python script until the operation completes. When the operation is completed, the execution of the Python script can be resumed. The Python script can contain multiple instructions containing parallel operations, each operation will be executed sequentially by the parallel system.

For the sake of better coordination, and imposing a total ordering on the operations requested through the Python script, the Python code runs inside a single interpreter attached to a single chare, and not inside a collection. This chare issues the parallel requests and gathers the results. Figure 9 shows an example of a Python script uploaded into Salsa.

The fact that only some pre-determined operations are available to the analyst can be seen as a limitation to the capabilities of the application. Nevertheless, the normal user of Salsa is not a computer scientist and may not be too familiar with parallel programming. Therefore, giving access to low-level details of Salsa, which would be possible using other tools, could have a negative impact on the usability. The ease of use of scripting language over declarative languages argued in other papers, is of especial importance in this case. It is one of the reasons why we chose Python over other languages. Another reason was the familiarity of the initial users of Salsa with Python.

5 Performance

The time the user has to wait between sending the code and receiving a response is important for the success of an interactive system. Response time is a known problem in existing tools. The benchmarks show that the time is very short (milliseconds). While evaluating the performance of our implementation, we focused on the overhead we incur. We did not consider the time spent to satisfy the user request (e.g the time spent in the loop to check the array correctness in Figure 8), since this can take as much as needed, and is not part of our interface. We also did not consider memory overhead.

We created two benchmarks with the same behavior as the two case studies described[1]. We ran our benchmarks on the NCSA Linux Cluster Abe, which consists of dual socket quad core Intel 64 2.33 GHz nodes interconnected with Infiniband OFED 1.2, through the batch scheduling queue. We used the net-linux ibverbs build of CHARM++ v6.0.1 (publicly available), compiled with gcc 3.4.6 and optimization -O3. The Python interpreter available was v2.5.2. We used the default CHARM++ timers which, for this platform, is gettimeofday.

The first benchmark creates a single chare where the Python requests are processed. The Python code contains a call to a high-level function. This C++ function broadcasts to all processors, which perform a certain amount of computation in parallel. The computation is defined by a simple loop with timer. At the end, a return value is reduced from all processors, and returned to the Python client, which prints it. The amount of computation performed by each processor in parallel is specified as an input parameter. We measured the time on the client, from when the Execute request is sent, until the ID of the interpreter used is returned back to the client. We made the Execute request wait for the completion of the Python code on the server. The total request time thus consists of (1) round-trip time of the message between the client and the server (within

[1]The benchmarks are available as part of the CHARM++ distribution under the directory tests/charm++/python

	Execution time in ms							
#procs	1	2	4	8	16	32	64	128
no reuse	41	69	222	474	503	1904	2905	1844
with reuse	0.7	0.8	0.9	0.9	1.3	1.2	1.9	1.9

Table 1. Client request processing time results in milliseconds with varying number of processors. The Python script runs inside an interpreter connected to a chare group.

	Execution time in ms									
#calls	0	1	2	4	8	16	32	64	128	1000
time	0.141	0.156	0.175	0.187	0.225	0.307	0.460	0.766	1.480	10.181

Table 2. Time to execute the script with varying number of calls to the high-level interface.

the same cluster); (2) creation of a user-level thread inside CHARM++; (3) creation of a new Python interpreter by the server (optional); and (4) execution of the Python script itself. For each execution, we sent 30 requests to the same server.

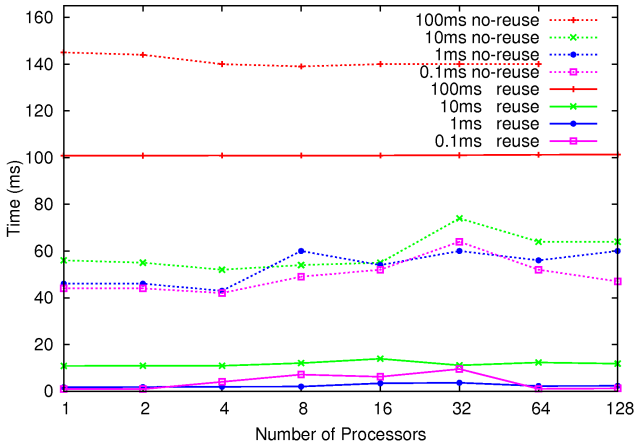

Figure 10. Client request processing time results in milliseconds with varying amount of computation (different color) and number of processors (X axis). The Python script runs inside an interpreter connected to a single chare. Dotted lines have a new interpreter created every request; solid lines have the same interpreter reused over multiple requests.

The results are shown in Figure 10 with varying number of processors and amount of computation performed by each processor. Dotted lines represent each request allocating a new Python interpreter on the server, while solid lines represent the same interpreter reused. The difference between corresponding lines show that the creation of a new Python interpreter (point 3) takes between forty to fifty milliseconds. This number is independent of the number of processors as expected, since the interpreter is created only on processor zero by a single chare. From the solid lines, by subtracting the amount of computation performed by the Python script which is known, we obtain the overhead of creating a new Python interpreter and a user-level thread. This is in the range of one to two milliseconds.

In the second benchmark, we used the CpdPythonGroup group, and sent a Python request to it. This request did not perform any work. Again we ran this test with varying number of processors, both creating new Python interpreters every request, and reusing the old one. Table 1 shows the results. As in the previous test, by reusing the same Python interpreter, we suffer only a few milliseconds of overhead. On the other hand, the overhead of creating new Python interpreters at every request grows to about two seconds for more than 32 processors. We do not understand this behavior completely, and we are still investigating it.

In all situations, having the client reuse the same Python interpreter for multiple requests reduces the overhead of the interface to below two milliseconds. This overhead can be tolerated both in a scenario of a user interactively writing code to upload, and in the scenario of a batch process uploading requests. Moreover, the performance results show that our implementation scales well up to at least 128 processors. This proves that our technique of uploading a high-level scripting language such as Python into a running parallel application is not only desirable, but also practicable.

Furthermore, we analyzed in greater detail the time spent to execute the Python script. We first tested the time taken to make a call from Python to CHARM++. We used the second benchmark, and increased the amount of work performed by the Python script by adding calls to the charm.getValue method. We ran the benchmark on a single processor to avoid pollution from the parallel environment. We collected and averaged ten requests, excluding the first one. Since the Python script runs on each processor independently from the others, the results reflect on the multiprocessor case as well. Table 2 shows the time taken to execute the script with varying number of calls to the high-level interface. By linearly interpolating the results, we can see that one function call accounts for about 10μs. This value is independent of

	Execution time in ms							
#elements	1	4	16	25	100	400	2500	10000
total	10.04	39.95	160.5	248.3	1017	4010	25158	100926
per element	10.04	9.99	10.03	9.93	10.20	10.03	10.06	10.09

Table 3. Time to execute the script with varying number of elements over which to iterate.

	Execution time in ms							
size	8000					1000		
#chares	64	256	1024	4096	25600	100	400	10000
original	186	156	163	315	1144	9.8	67	297
with Python	188	155	159	312	1151	9.9	64	289

Table 4. Execution time of a 5-point 2D Jacobi application on a matrix with dimension *size*X*size* decomposed into *#chares* chares on 32 processors.

our implementation and depends on the Python/C library.

Secondly, we tested the overhead of the iterative interface to apply the same Python operation to multiple input elements (see Figure 6). We again used the second benchmark. Table 3 shows the results with varying number of elements over which iterating. It can be seen that the time scales linearly with the number of elements, therefore the overhead of repeatedly calling Python for each element is virtually zero.

Finally, we experimented with a real application to see the impact of repeatedly running Python scripts to check for application bugs. We used a 5-point 2D Jacobi application. Through CHARMDEBUG, we installed a lightweight version of the code in Figure 8 stripped of the time consuming loop (since we are interested in the overhead only). This checking code ran after every message exchanged by the application (roughly four times the number of chares). We ran this on a 4-node Linux cluster, each node composed of dual socket quad core Intel Xeon 2.0 GHz, against the "original" program (which does not even contain the [python] keyword). Table 4 shows the performance results with varying amount of computation, determined by the matrix size, and granularity, determined by the number of chares. The overhead to link the Python interface and run the checking code is negligible in all scenarios, even in the extreme ones with thousands of chares.

6 Related work

Other tools performing dynamic insertion of code into a running application include DynInst[3]. While an application is running, they allow an external program, called *mutator*, to attach to the running application, and modify its code image. After the image has been modified, the application will continue running the new code. This approach allows great flexibility in how the code is modified. Nevertheless, DynInst is not meant to be used directly by the user to write the new code, but through other tools that will simplify the modification process, which is otherwise tedious and potentially error prone. More recently, Dyner[25] has provided a TCL interface to the DynInst library to allow any modification to the user code in a simpler way. While this approach allows any modification at the source level, it does not provide the right level of abstraction for some kind of applications, like data analysis. Our aim is to allow the user to easily write a snippet of code to perform the desired operation while the application is running, and having it run immediately. In our approach the application developer retains the faculty to provide operations at the desired level of abstraction, and deny others that should not be used directly. This is an advantage for closed-source codes, where the user otherwise has to step down to the assembly level.

Other tools[7, 10] are used to patch non-stop applications to update them from one version to the following. These programs, like DynInst, provide low-level patching mechanisms, which again are not suitable for some kind of applications. Moreover, the patching mechanism is only for expert programmers, as the uploaded code is supposed to have passed all correctness tests.

GDB[8] provides the capability to inspect variables when the program is suspended at a breakpoint, as well as suspend execution when a condition is satisfied. A breakpoint in GDB can be set at any instruction line in the source code. While this is a powerful tool for debugging, if the condition is complicated, this approach might not be practical. For parallel distributed applications written in MPI, TotalView[24] can provide similar functionality. Again, if the checking code to run is complicated, writing it correctly might be challenging. It is agreed that scripting languages such as Python, Lua or Ruby are easier to use than programming languages like C/C++ or Fortran. In our approach, we focused on the usage of scripting languages to simplify the on-the-fly writing of checking code.

On the topic of introspection within application written in C++, many tools have been built[4, 5, 9, 22, 23]. The main scope of these works is to provide the program itself

access to its data types. In our approach, we used the information already collected by CHARMDEBUG to provide this capability. Nevertheless, these other approaches are also viable implementations, and might be considered in future work.

Tools like VASE[11] allow the user to interactively visualize the progress of an application and steer it when necessary. In VASE, all the code that the application can use has to be written and compiled in the application. The steering mechanism will allow the selection of which code to execute. Instead, we wanted to let the user write new code even when then application is already running.

7 Conclusion and Future Work

In this paper, we presented an interface to support dynamic insertion of a high-level scripting language inside a running parallel application. We described its implementation in the context of the Python language and the CHARM++ parallel runtime system. We showed how this interface has been successfully used by two applications to provide the user flexibility to run arbitrary Python code in two very different contexts: parallel debugging, where access to low-level information is essential, and data analysis, where high-level parallel procedures are needed. Many other classes of application can benefit from our technique, in particular those involving computational steering. For these, a user can probe periodically the application, and steer it with ad-hoc Python scripts when needed. We have also shown that the performance overhead to upload and run Python code inside a parallel application is very small and negligible in a context of interactive usage.

This same approach can be applied to other programming paradigms to provide flexibility at runtime. One such paradigm is MPI, very commonly used in parallel computing. In this environment, there are two possible implementations. At the user level, the user can probe for incoming requests, and use a Python library derived from the one described in this paper to execute Python scripts on demand. At the MPI library level, the MPI runtime system itself can receive incoming requests and process them inside Python interpreters, similarly to how CHARM++ behaves. In this latter case, a request could be executed anytime the application calls a function in the MPI library, or specifically when the application triggers Python request handling, say by calling a function like MPI_Python.

A natural future direction to extend this interface in the context of CHARM++ is to integrate other scripting languages, so that the user can choose which one to use. In this direction, another project not involving the authors of this paper is integrating Lua into the CHARM++ runtime system[19].

Within the context of Python, a direction to explore is the use of a preprocessor or static analyzer to provide default read/write methods. This will facilitate the developer's task to create possibly complex methods. For finer access control, the user could annotate the source code to specify which fields should be accessible, and with what privileges. The default identifier used in Python to identify the variables could be a simple string with the name of the variable to access. A more ambitious extension could use an instrumentation tool, like DynInst, to dynamically modify the interface an application exports to Python. This will allow the developer to add functionality unforeseen at compile time, while maintaining a simple interface for final users. This freedom entails an even greater risk to corrupt the application, thus requiring a careful design against programmer mistakes.

Acknowledgements: This work have been made possible in part by grants NSF OCI-0725070, NASA NNX08AD19G, and NSF ITR-0205611. We would like to thank TeraGrid for the compute time granted through allocation TG-ASC050039N. The authors are grateful to Prof. Tom Quinn (Univ. of Washington), who is a collaborator with them on the cosmology application.

References

[1] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[2] E. Bohm, G. J. Martyna, A. Bhatele, S. Kumar, L. V. Kale, J. A. Gunnels, and M. E. Tuckerman. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[3] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[4] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. *Lecture Notes in Computer Science*, 707:482–??, 1993.

[5] T.-R. Chuang, Y. S. Kuo, and C.-M. Wang. Non-intrusive object introspection in C++. *Software– Practice and Experience*, 32(2):191–207, 2002.

[6] Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL. *The CONVERSE programming language manual*, 2006.

[7] P. Falcarin and G. Alonso. Software architecture evolution through dynamic aop. In *AOP , European Workshop on Software Architecture (EWSA 2004*, pages 57–73. Springer-Verlag, 2004.

[8] Free Software Foundation. GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/.

[9] J. Hamilton, R. Klarer, M. Mendell, and B. Thomson. Using SOM with C++. C++ report, August 1995.

[10] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, 2005.

[11] D. Jablonowski, J. Bruner, B. Bliss, and R. Haber. Vase: The visualization and application steering environment. *Supercomputing '93. Proceedings*, pages 560–569, 15-19 Nov. 1993.

[12] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[13] R. Jyothi. Debugging support for charm++. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[14] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[15] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[16] S. Kumar. *Optimizing Communication for Massively Parallel Processing*. PhD thesis, University of Illinois at Urbana-Champaign, May 2005.

[17] O. Nierstrasz, R. Bergel, M. Denker, S. Ducasse, M. Glli, and R. Wuyts. On the revival of dynamic languages. In *Proceedings of Software Composition 2005. LNCS*, pages 1–13, 2005.

[18] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31:23–30, 1998.

[19] T. Ponte and N. Rodriguez. LuaCharm: Implementing Chares in a High-Level Scripting Language. Presentation at 6th Annual Workshop on Charm++ and its Applications (http://charm.cs.uiuc.edu/workshops/charmWorkshop2008/), May 2008.

[20] Python Software Foundation. Python/C API Reference Manual, 2008. http://docs.python.org/api/api.html.

[21] T. Quinn, L. Kale, F. Gioachin, O. Lawlor, G. Lufkin, and G. Stinson. Salsa: a parallel, interactive, particle-based analysis tool. Poster at Supercomputing 2004.

[22] H. Singh. Introspective c++. Master's thesis, Computer Science Department, Virginia Polytechnic Institute and State University, 2004.

[23] K. Stephens. Xvf: C++ introspection by extensible visitation. *SIGPLAN Not.*, 38(8):55–59, 2003.

[24] TotalView Technologies. TotalView® debugger. http://www.totalviewtech.com/TotalView.

[25] C. Williams and J. Hollingsworth. Interactive binary instrumentation. *IEE Seminar Digests*, 2004(915):25–28, 2004.

[26] G. Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[27] G. Zheng, O. S. Lawlor, and L. V. Kalé. Multiple flows of control in migratable parallel programs. In *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*, pages 435–444, Columbus, Ohio, August 2006. IEEE Computer Society.