Native Code Execution and Snapshot on AgentTeamwork

CSS497 Autumn 07 – Winter 08: Cooperate Education: Henry Sia

Intermediate Report (10/12/2007)

1. Overview

AgentTeamwork is a grid-computing middleware system that deploys mobile agents to remote computers where they launch, monitor, and resume a parallel-computing user job in their hierarchy. To simplify the implementation, Java has been used for all AgentTeamwork's components and only Java programs are assumed as our applications. However, needless to say, C/C++ programs should be allowed to run on top of AgentTeamwork. This project main goal is to add native-code execution functionality to AgentTeamwork and conduct performance evaluation with some C/C++ parallel applications.

2. Progress

This report contains information about the work done during the first half of the schedule. During the first 3 weeks, I learned about how a Java program can make calls to C/C++ functions and vice versa. This can be done through Java Native Interface (JNI) through a set of API that can be called through JNIenv pointer. As outlined by the schedule, I implemented several programs such as:

- Program 1.1: C/C++ functions from Java and returning a value to Java
- Program 1.2: Accessing Java objects from C/C++ functions
- Program 1.3: Creating heap memory space from C/C++ functions into a JVM
- Program 1.4: Serializing and de-serializing heap memory space C/C++ functions have allocated
- Program 2.1: designing a Java function that calls C/C++ main function
- Program 2.2: designing a C/C++ function that allocates a byte array in a JVM and returns to the calling C/C++ program a void pointer to this array.

This project is done under Linux OS with Java Development Toolkit 6. First of all, of course I learn the basic of JNI. In order for the JVM to call a function written in C/C++, the function name has to be declared in the .java class by using the keyword `native`. For example: `public native void callThisMethod()`. In order to create a header file for this function, the .java class has to be compiled using `javah –jni` option. The compiler will create a machine generated .h file where it contains the native functions' signature. Then, I implemented the .c / .cpp file based on the .h.

Next, I need to compile the C/C++ file into a loadable library .so using `g++ -rdynamic filename.cpp -o libraryname.so_ -shared -ldl` option. These file then can be loaded by the Java class through a system function call `System.loadLibrary(libraryname)`. Only after this library is loaded, the C/C++ function can be called.

In program 1.2, to access java objects from C/C++ functions can be done by calling `<jniType>Get<type>Field( JNIEnv, *env, jobject obj, jfieldID fieldID )`. The jobject defines what type of object we want; it can be a primitive type or an object. Another important thing here is the jfieldID where we need to get the ID of the object we want by calling `jfieldID GetFieldID( JNIEnv *env, jclass clazz, const char *name, const char *sig )`. The jclass defines which class we want to get the variable from, we can get it by calling jclass

`GetObjectClass ( JNIEnv *env, jobject obj )`, const char *name defines the name of the java object, and the const char *sig defines the type of the java object.

For program 2.1, the C/C++ program also has to be compiled into a loadable library which will be called by the JNI function. Example of the compile command is: `g++ -shared -o ProgramName.so ProgramFile.cpp`. This will then be loaded into the system by calling dlopen, dlsym, and dlclose.

Special note on program 2.2 is that in order to create primitive type arrays in JVM from C/C++ function, it has to be done by calling New<pType>Array(JNIEnv *env, jsize length) through JNIenv pointer. So, an array of a primitive type can only be read as an array of that particular primitive type. For example, an array of bytes cannot be read as an array of int, or other primitive types.

For the rest of the quarter, I utilize everything I've learned from program 1.1 through 2.2 in order to allow AgentTeamwork to work with C/C++ program which include designing these programs:

- Program 2.3: designing C/C++ functions that call Ateam's all methods including saveSnapshot( ).
- Program 2.4: designing C/C++ functions that call mpiJava functions. Those functions must have the same interface as the original MPICH functions.
- Program 2.5: designing C/C++ functions that call GridFile functions. Those functions must have the same interface as the UNIX open/read/write/close system calls.

I haven't been able to completely finish writing program 2.4 and 2.5 but should be finished during the winter holiday. Moreover, I have not been able to test the code thoroughly but it shouldn't be too hard as most of the code is just calling existing functions.

3. Conclusions

I think I'm pretty much in good shape in terms of the quality of work and the work done during this first half. The only setback was caused by the surgery where I couldn't do anything for 3 days because of the medicine I took in order to reduce the pain I felt. I'm very glad to have been able to learn so much in a very short time. Not only that this work is very interesting but it also stimulates my brain and allow me to learn many ways to solve problems.