Parallel-C++ for the Java Virtual Machine

Timothy E. Denehy
Department of Computer Science
University of Dayton
Dayton, OH 45469
denehyte@flyernet.udayton.edu

Chang-Hyun Jo
Department of Computer Science
University of North Dakota
Grand Forks, ND 58202-9015
jo@cs.und.edu

ABSTRACT

Object-oriented modeling and design is a way of solving problems using models based upon real-world concepts. In this paradigm, the fundamental construct is the object, an entity that encompasses both data structure and behavior. Parallel computing strives to increase computational power through task subdivision and concurrent execution. The combination of these two areas has led to active research in concurrent object-oriented programming languages. The Parallel-C++ language was designed to support concurrent object-oriented programming in parallel and distributed computing environments. The language provides support for fine-grained parallelism, concurrent process execution, inter-process communication, and object migration. *A prototype Parallel-C++ to Java Byte Code compiler is implemented to provide program execution on any architecture that supports the Java Virtual Machine*. This mechanism could serve as the basis for an environment that facilitates the implementation of distributed applications and object migration across heterogeneous systems.

Keywords

Parallel-C++, Java Virtual Machine, Java Byte Code

1 INTRODUCTION

The evolution of parallel computers has offered the potential of greater speed-up and increased efficiency. Concurrent processes are processes whose execution can overlap in time. Parallel processing achieves this simultaneous execution by distributing work across several processing elements. Many real-world applications that involve overlapping tasks can be implemented and modeled using concurrent programming.

This research was supported by the NSF REU Grant #9619805 to the University of North Dakota in 1999.

--
ACM SAC 2000, Como, Italy, 843-848, March 2000.

Object-oriented modeling and design is a way of solving problems using models based upon real-world concepts. In this paradigm, the fundamental construct is the object, an entity that encompasses both data and associated operations. Objects are useful constructs for modeling real-world entities because they store an internal state (that distinguishes them from other objects) and they interact through well-defined interfaces. Object-oriented programming also provides many other benefits, such as encapsulation, modularity, and code reuse. The combination of these two fields has lead to research in concurrent object-oriented programming languages [1]. Parallel-C++ is one of the pioneer concurrent programming languages that supports mobile objects [2, 3].

This paper presents our experience in implementing a compiler to translate Parallel-C++ source code to Java Byte Code to be executed on various machines supporting the Java Virtual Machine.

The next section of this paper describes Parallel-C++, a concurrent object-oriented programming language. Section 3 introduces the Java Virtual Machine, its instruction set, and its binary file format. Section 4 describes an implementation of a prototype Parallel-C++ compiler that targets the Java Virtual Machine, and section 5 provides an example compilation. The final section summarizes the work that has been done and presents ideas for the future work.

2 PARALLEL-C++

Parallel-C++ [2, 3] is an extension to the C++ language that preserves the object-oriented language features and adds new constructs to support parallel programming. C++ was chosen as a base language because of its familiar syntax and widespread use. Several new keywords are added to the C++ language to define the parallel extensions. These keywords are listed in Table 1.

parbegin	parend
autobegin	autoend
parallel_commands	end_parallel_commands
process	end_process
parallel_functions	end_parallel_functions
send	recv
export	import

Table 1: Keywords Added by Parallel-C++

The first extension to the language is the "parallel statements" construct. Its syntax is shown in Figure 1. The statements within this construct will be executed in parallel at run-time, achieving fine-grained parallelism. The programmer is responsible for insuring the independence of these statements. Alternatively, use of the "automatic processor allocation" construct causes the compiler to analyze the dependencies between the concurrent statements and parallelize them accordingly. Its syntax is illustrated in Figure 2.

```
parbegin { <statements> } parend;
```
Figure 1: Parallel Statements Syntax

```
autobegin { <statements> }autoend;
```
Figure 2: Automatic Processor Allocation Syntax

Several extensions also provide for medium and large-grained parallelism. The "parallel function call" construct, shown in Figure 3, provides the ability to call several functions in parallel. The function calls are executed concurrently, and the parallel processes are synchronized at the end of the construct. The "explicit process allocation" construct allows the user to define several processes to be run concurrently. Its syntax is shown in Figure 4.

```
parallel_functions {
    <function-call-statements>
} end parallel_functions;
```
Figure 3: Parallel Function Call Syntax

```
parallel_commands
<number-of-processes>) {
        process(<process-id>)
                <compound-statement>
        end_process;
        …
} end parallel_commands;
```
Figure 4: Explicit Process Allocation Syntax

The "synchronization" statements, shown in Figure 5, are provided to facilitate communication between concurrently executing processes. These statements provide both synchronous and asynchronous communication. The "object migration" constructs allow for the transfer of objects between other objects and processes. Their syntax is shown in Figure 6. The exporting object or process continues its execution even if nothing imports its object. The importing object, however, must wait until an object becomes available. These constructs implement the "copy-and-delete" rule. The exported object is copied to the import list of the specified object or process, and the object is deleted from the memory of the exporting object.

```
send(<msg-id>, <destin-id>, <msg-type>);
send(<msg-id>, <destin-id>);
recv(<source-id>, <msg-id>, <msg-type>);
recv(<source-id>, <msg-id>);
```
Figure 5: Synchronization Syntax

```
export(<object-id>, <destination-id>);
<object-variable> = import(<class-id>);
<object-variable> = import(<class-id>,
                           <source-id>);
```
Figure 6: Object Migration Syntax

3 THE JAVA VIRTUAL MACHINE

The *Java Virtual Machine* [4, 5] is an abstract computer. Like other computers, the Java Virtual Machine has an instruction set, *Java Byte Code*, and a recognized binary file format, the class file. The class file is an architecture and operating system independent binary file format that defines a class or interface along with its methods and data members.

The Java Virtual Machine supports primitive types, such as char, int, long, float, and double, as well as reference types, which are used to access objects in memory. During run-time, the Java Virtual Machine maintains several data areas to support the executing program. A program counter is used to record the address of the currently executing instruction. A stack is used to store frames related to method invocation. Each frame contains space for local variables and for an instruction operand stack. The machine also manages a heap for the allocation of object and array instances and a run-time constant pool to store values such as literals and object specific information.

Java Byte Code includes instructions typical of any machine instruction set. Load and store instructions for the different types are available along with specialized instructions to access object members. Arithmetic instructions specific to their operand types are included, as well as instructions to convert values between the primitive types. Other instructions are provided to allocate and manage object instances and to manipulate the operand stack. There are also several comparison and branching instructions and instructions to invoke named methods.

Programs written in Java Byte Code are converted into the binary class file format. The resulting file contains the binary version of the program instructions along with supporting data such as constant pool values and other information. Before executing a program, the Java Virtual Machine verifies that the class file conforms to the correct format and validates the instructions it contains. Upon successful verification, the machine creates and links an instance of the specified class, initializes it, and executes its "main" method. The machine dynamically loads, links, and initializes other classes, as they are needed.

4 TRANSLATING PARALLEL-C++ TO JAVA BYTE CODE

A prototype compiler from Parallel-C++ to Java Byte Code has been implemented. The Java Byte Code produced by the compiler can be assembled, and the resulting Java class file can be executed on the Java Virtual Machine.

4.1 Implementation

The lexical analyzer for the compiler was generated using *lex*, a Unix compiler tool. The input to lex is a file that defines the tokens of the language using regular expressions. Lex outputs C code, scanner, that recognizes and identifies tokens on the input stream, and this code is accessed by the parser.

The parser for the compiler was generated with another Unix compiler tool, *yacc*. The input to yacc is a grammar definition for the language along with code to be executed in coordination with the firing of the various grammar productions. These code fragments are used to generate the appropriate Java Byte Code during parsing. Yacc generates C code, parser that parses the input and executes the code specified with each rule. Because there are several ambiguities in the C++ language, a special version of yacc called backtracking yacc [6] was used. When confronted with a shift/reduce or reduce/reduce conflict in the grammar, the generated parser performs several test parses until a successful path is found. At this point, the parser backtracks and executes the code fragments associated with the path's productions. The Java Byte Code generated by the compiler is converted into the Java class file format using *jasmin* [5], a Java Byte Code assembler.

A symbol table was implemented in C++ to maintain information about the various identifiers in the input source. The table supports insertion and retrieval of identifiers, and it stores the identifiers in a hash table for faster access. The symbol table also includes routines to record the current scope in the source file and to generate and manage program labels for the output.

4.2 Translation Scheme

The primitive numeric types in Parallel-C++ (int, long, float, double) are translated directly to their counterparts in Java Byte Code. Mathematical, logical and comparison operations are implemented using the appropriate Java Byte Code instructions.

Because the format of the Java Virtual Machine requires programs to be associated with classes, a default class is created by the compiler with the same name as the input file. Functions in the source code are translated into static methods of the default class, and local variables are implemented using the local frame stack provided by the Java Virtual Machine. An outline for a default class is shown Figure 7.

Parallel-C++ classes and structures are translated into their own class files as recognized by the Java Virtual Machine. This scheme provides the ability to allocate class instances and access class members the facilities and instructions present in the Java Virtual Machine. Data members are translated into class fields, and function members are translated into class methods. These class fields and methods can be accessed by name with an object reference and the appropriate Java Byte Code instructions. An outline for a class translation is shown in Figure 8.

In order to implement the "explicit process allocation" construct, a distinct class file is generated for each process. These classes are derived from the Java thread class, and thus inherit the concurrent execution abilities represented by Java threads. The statements for each process are placed in their class's "run" method, and the execution of each thread is started in turn to achieve the parallel execution desired. Figure 9 shows an outline for the translation of this construct.

```
.class public <input-filename>
; class definition
.super java/lang/Object
; superclass
.method public <init>()V
; standard initializer
  aload_0
  invokenonvirtual
    java/lang/Object/<init>()V
  return
.end method

; static function
.method public static <function-name>
   (<parameter-specifiers>)<return-type-
   specifier>
  .limit stack  100
  .limit locals 100
  <function-instructions>
.end method
```

Figure 7: Translation for Default Class

```
.class public <class-name>
; class definition
.super java/lang/Object
; superclass

; class field definition
.field public <field-name> <type-
    specifier>

; class method definition
.method public <method-name>(<parameter-
    specifiers>) <return-type-specifier>
  .limit stack  100
  .limit locals 100
  <function-instructions>
.end method
```

Figure 8: Translation for Classes

```
.class public class<process-name>
; class definition
.super java/lang/Thread
; superclass

.method public run()V
; class method definition
   .limit stack  100
   .limit locals 100
   <function-instructions>
.end method
```

Figure 9: Translation for Explicit Process Allocation

For the implementation of the "object migration" constructs, an object migration manager class was implemented in Java to coordinate the export and import of objects. The manager class contains a collection of queues for each object and process involved in migration. The queues are stored in a hash table based on their object's or process's identifier so that they can be accessed quickly. In the case of an import call, the compiler generates code to call the object migration manager's "dequeue" method. If there is an appropriate object available on the caller's import queue, it removed and returned to the caller. If the import queue is empty, the object's or process's "wait" method is called to halt its execution. When an export call is made, the compiler generates code to call the object migration manager's "enqueue" method. The object is placed on the import queue of the specified object or process, and its "notify" method is called if it is waiting to import an object. The object is also deleted from the exporting object, and the exporting object continues its execution. The translation schemes for these constructs are outlined in Figure 10 and Figure 11.

```
aload <local-number>
; load reference to object
ldc "<class-name>"
; load string constant for class identifier
ldc "<process-name>"
; load string constant for process id.
invokestatic
  OMManager/enqueue(Ljava/lang/Object;
  Ljava/lang/String;Ljava/lang/String;)V
; call migration manager method for export
aconst_null
; load null reference
astore <local-number>
; store reference to object
```

Figure 10: Translation for Object Migration Export Construct

```
aload <local-number>
; load reference to object
aload_0
; load self reference
ldc "<class-name>"
; load string constant for class
identifier
invokestatic
    OMManager/dequeue(Ljava/lang/Object;
    Ljava/lang/String;)Ljava/lang/Object;
; call migration manager method for import
astore <local-number>
; store reference to object
```

Figure 11: Translation for Object Migration Import Construct

5 EXAMPLE COMPILATION

The Parallel-C++ source code in Figure 12 illustrates the use of the "explicit process allocation" and "object migration" constructs to model two airport processes transferring control of airplane objects. Each airport process creates a new airplane object, sets its flight number, exports it to the other process, and imports a new airplane object. The Java Byte Code generated by the compiler for this example is shown in Figure 13, Figure 14, and Figure 15. The code generated for the London process is extremely similar to that for the Paris process, and thus is omitted. Comments have been added to the generated code for explanatory purposes. The symbol table used during compilation is also shown in Figure 16.

```
class Airplane {
   int   flight;
   void  setFlight(int i);
};

void main() {

   parallel_commands(2) {

      process(Paris) {
         Airplane   f88;
         Airplane   y;

         f88.setFlight(88);
         export(f88, process(London));

         y = import(Airplane);

      } end_process;

      process(London) {
         Airplane   f144;
         Airplane   x;

         f144.setFlight(144);
         export(f144, process(Paris));

         x = import(Airplane);

      } end_process;

   } end_parallel_commands;

   return;
}

void Airplane::setFlight(int i) {
   flight = i;
}
```

Figure 12: Example Parallel-C++ File main.cc

```
.class public main
; class definition

.super java/lang/Object
; superclass

.method public <init>()V
; standard initializer
   aload_0
   invokenonvirtual
     java/lang/Object/<init>()V
   return
.end method


.method public static
   main([Ljava/lang/String;)V
   .limit stack  100
   .limit locals 100

   new classParis
   ; create new class instance
   dup
   ; duplicate reference
   invokespecial classParis/<init>()V
   ; call class initializer
   dup
   ; duplicate reference
   astore 0
   ; store reference locally
   dup
   ; duplicate reference
   ldc "Paris"
   ; load string constant for object id.
   invokestatic
   OMManager/addObject(Ljava/lang/Object;
   Ljava/lang/String;)V
   ; add this object to the migration mgr

   new classLondon
   ; create new class instance
   dup
   ; duplicate reference
   invokespecial classLondon/<init>()V
   ; call class initializer
   dup
   ; duplicate reference
   astore 1
   ; store reference locally
   dup
   ; duplicate reference
   ldc "London"
   ; load string constant for object id.
   invokestatic
   OMManager/addObject(Ljava/lang/Object;
   Ljava/lang/String;)V
   ; add this object to the migration mgr
   invokevirtual classLondon/start()V
   ; begin thread execution
   invokevirtual classParis/start()V
   ; begin thread execution
   return
   return
.end method
```

Figure 13: Example Java Byte Code File main.j Generated by Compilation of main.cc

```
.class public Airplane
; class definition
.super java/lang/Object
; superclass

.field public flight I
; class field definition

.method public <init>()V
   ; standard initializer
   aload_0
   invokenonvirtual
     java/lang/Object/<init>()V
   return
.end method

.method public setFlight(I)V
   ; class method definition
   .limit stack  100
   .limit locals 100

   aload 0
   ; load self reference
   getfield Airplane/flight I
   ; load member flight
   iload 1
   ; load local i
   dup
   ; duplicate value
   aload 0
   ; load self reference
   swap
   ; swap operands
   putfield Airplane/flight I
   ; store member flight
   return
.end method
```

Figure 14: Example Java Byte Code File Airplane.j Generated by Compilation of main.cc

6 CONCLUSION

A prototype Parallel-C++ compiler has been implemented to target Java Byte Code. This allows the Parallel-C++ source program to be run on any architecture that supports an implementation of the Java Virtual Machine.

In the future, extensions could be added to the existing system to facilitate the distribution of processes and objects across heterogeneous systems. Because the binary Java class file format is uniform across all systems, the Java Virtual Machine would serve as an ideal host for the use of object migration across different architectures. The Java Virtual Machine would also be an ideal platform for the execution and exchange of mobile agents.

```
.class public classParis
; class definition
.super java/lang/Thread
; superclass

.method public <init>()V
; standard initializer
   aload_0
   invokenonvirtual
     java/lang/Thread/<init>()V
   return
.end method

.method public run()V
; class method definition
   .limit stack  100
   .limit locals 100

   new Airplane
   ; create new class instance
   dup
   ; duplicate reference
   invokespecial Airplane/<init>()V
   ; call class initializer
   dup
   ; duplicate reference
   ldc "f88"
   ; load string constant for object id.
   invokestatic
   OMManager/addObject(Ljava/lang/Object;
   Ljava/lang/String;)V
   ; add this object to the migration mgr
   astore 1
   ; store local reference

   new Airplane
   ; create new class instance
   dup
   ; duplicate reference
   invokespecial Airplane/<init>()V
   ; call class initializer
   dup
   ; duplicate reference
   ldc "y"
   ; load string constant for object id.
   invokestatic
   OMManager/addObject(Ljava/lang/Object;
   Ljava/lang/String;)V
   ; add this object to the migration mgr
   astore 2
   ; store local reference

   aload 1
   ; load reference to f88
   ldc 88
   ; load constant 88
   invokevirtual Airplane/setFlight(I)V
   ; call setFlight class method

   aload 1
   ; load reference to f88
   ldc "Airplane"
   ; load string constant for class id.
   ldc "London"
   ; load string constant for process id.
   invokestatic
   OMManager/enqueue(Ljava/lang/Object;
   Ljava/lang/String;Ljava/lang/String;)V
   ; call migration mgr method for export

   aconst_null
   ; load null reference
   astore 1
   ; store reference to f88

   aload 2
   ; load reference to y
   aload_0
   ; load self reference
   ldc "Airplane"
   ; load string constant for class id.
   invokestatic
   OMManager/dequeue(Ljava/lang/Object;
   Ljava/lang/String;)Ljava/lang/Object;
   ; call migration mgr method for import
   astore 2
   ; store reference to y
   return
.end method
```

Figure 15: Example Java Byte Code File classParis.j Generated by Compilation of main.cc

Name	Type	Descriptors
Airplane		class
Airplane::flight	int	
Airplane::setFlight	void	function
main	void	function
classParis		process class
main::Paris	classParis	local 0 process class
classParis::run::f88	Airplane	local 1 class
classParis::run::y	Airplane	local 2 class
classLondon		process class
main::London	classLondon	local 1 process class
classLondon::run::f144	Airplane	local 1 class
classLondon::run::x	Airplane	local 2 class
Airplane::setFlight	void	function
Airplane::setFlight::i	int	local 1

Figure 16: Example Symbol Table Generated by Compilation of main.cc

REFERENCES

1. Agha, G., P Wegner and A. Yonezawa. “Research Directions in Concurrent Object-Oriented Programming”, MIT Press, (1993).
2. Jo, Chang-Hyun. “The Design and Implementation of an Object-Oriented Parallel Programming Language”. Ph.D. Dissertation, Oklahoma State University, (1991).
3. Jo, Chang-Hyun et al. "A realization of a concurrent object-oriented programming", ACM 1998 Symposium on Applied Computing (SAC'98), 558-563, (Feb. 1998).
4. Lindholm, Tim and Frank Yellin. “The Java Virtual Machine Specification”, Sun Microsystems, Addison-Wesley, (1999).
5. Meyer, J. and T. Downing. “Java Virtual Machine”, Addison-Wesley, (1999).
6. Siber Systems. “BtYacc”, <www.siber.com/btyacc/>, (1999).