PARALLELIZING TRANSLATOR FOR AN OBJECT-ORIENTED PARALLEL PROGRAMMING LANGUAGE

Chang-Hyun Jo †
K. M. George †
Keith A. Teague *

† Department of Computer Science
* School of Electrical and Computer Engineering
Oklahoma State University
Stillwater, OK 74078

ABSTRACT
This paper is concerned with the implementation of a translator for an object-oriented parallel programming language, Parallel-C++, on an Intel iPSC/2 hypercube computer system. Parallel-C++ is an extension of C++. The novel concept of object migration and the concept of explicit process allocation are added to C++. Parallel-C++ supports parallel and distributed programming within the object-oriented paradigm. The language constructs which support parallel and distributed programming are "parallel_commands" and "export / import". The constructs "export / import" are used to export and import dynamic objects that travel among static objects. Parallel-C++ provides features to represent problems such as air traffic control systems, dynamic load balancing, network transmission, and other similar problems in a natural way.

I. INTRODUCTION
Several object-oriented programming languages have been designed and implemented. Some such as C++[33] and CLOS[11] are extensions of existing conventional or functional programming languages, while others such as Smalltalk[15] and Eiffel[25] are newly designed languages. While some languages add new concepts to object-oriented programming languages, others adapt object-oriented methods to new areas of application such as distributed computing. In this paper, we discuss the implementation of an object-oriented parallel programming language which incorporates new ideas into C++.

A number of projects have reported research about parallel programming and its language evolution[2,4,16,22,23,27,28,29]. Besides the above, there are also a number of concurrent and parallel programming language models[1,8,9,13,17,24,26]. Several researchers have also introduced parallelism into object-oriented programming[32,36]. An object-oriented parallel programming language, Parallel-C++,[19,20], has been developed by the authors to support parallelism within the object-oriented programming paradigm. Parallel-C++ is based on C++ and preserves the properties of object-oriented programming while providing parallel programming. It also incorporates a new language idea to support object migration.

This paper focuses on the implementation of two important language constructs. Implementation techniques are presented for the following:

(1) The language construct "parallel commands" - which provides support for explicit concurrent process allocation. The concurrent processes can be mapped into parallel processors at run-time.
(2) The language constructs "export / import" - which provide support for object migration between objects and processes in the system.

II. RELATED WORK
Parallel-C++[19,20] is an extension of the C++ language [33]. Language constructs are added to support parallelism within the object-oriented programming paradigm. Furthermore, it incorporates a new concept and language constructs which can support object migration. Parallel-C++ preserves the properties of object-oriented programming, such as information hiding and inheritance, while providing support for parallel and distributed programming.

Incorporation of concurrency into object-oriented programming has been the topic of study by many researchers[5,6,12,14,32,34,36]. Other related work in the area of concurrent programming languages and systems[7,10,21] is concerned with load balancing, process migration and object migration. Several research projects in the distributed systems area[3,30] have used process and object migration ideas. The works of Shapiro[31] and Yin[35] focus especially on the research of parallel languages and systems based on C++.

III. PARALLEL-C++ TRANSLATOR IMPLEMENTATION
An overall structure of the translator is shown in Figure 1. Even though the translation scheme takes advantage of existing facilities, the new language features incorporated into Parallel-C++ make the translator design a difficult task. The Parallel-C++ translator can be viewed as a high level language transformer. It translates the Parallel-C++ code into a source code for a C++ compiler for the iPSC/2[18]. To facilitate parallel execution in the hypercube, the original source program written in Parallel-C++ is transformed using several steps. One of the steps is to identify parallel sections in a program. Parallel sections are transformed into node programs for the iPSC/2 that can be executed concurrently. The translation...
schemes used and corresponding algorithms are described in the following sections.

Figure 1. Structure of Parallel-C++ Translator

3.1 Concurrent Processes

In Parallel-C++, the "parallel_commands" construct is provided to support explicit process allocation. It supports the explicit generation of concurrent processes that can be executed in parallel on a multiprocessor machine. The parallel_commands construct combined with object migration (discussed in Section 3.2) provides the programming environment for distributed applications such as load balancing. Figure 2 shows the syntax and semantics of such a language construct. Figure 3 shows a simple illustrative example of the use of explicit process allocation in a program. The program consists of two processes. Each process consists of a block of sequential statements. The two processes run concurrently and they communicate by sending messages. The "end_parallel_commands" statement acts as a synchronization point. The next section illustrates how this program segment will be processed by the Parallel-C++ translator.

**Syntax:**
"parallel_commands (number_of_processes)

**Semantic Notes:**
1. All the processes may run concurrently and independently at run-time. They are capable of communicating with each other.
2. Parallel_commands can be nested.
3. Number_of_processes indicates the total number of processes in the construct.

Figure 2. Explicit Process Allocation Construct

**Example:**

```
parallel_commands (2) {

    process(0) { x = 0; send(x, process(1)); recv(process(1), y); x = y + 1; cout << x; } end_process; // x = 2
    process(1) { recv(process(0), z); z++; send(z, process(0)); cout << z; } end_process; // z = 1
}
```

Figure 3. Explicit Process Allocation Example

3.1.1 Automatic Processor Allocation Tree

In order to allocate a processor to a process, the translator generates an Automatic Processor Allocation tree (APA tree) using the APA tree generation algorithm (Algorithm 1). If there are enough processors available on the target machine, each process in the explicit process allocation may be mapped using one-to-one mapping into each physical processor. In an APA tree, each node represents a process in a parallel_commands construct. The Parallel-C++ translator generates an APA tree as shown in Figure 4.a when the parallel_commands construct in Figure 3 is parsed. Figure 4.b shows the structure of a node in the APA tree. In Figure 4.a, two processes "P0" and "P1" are placed on two nodes. Those two nodes are leaf nodes which are executable in parallel. Each node will be assigned to a processor, and the necessary code to keep this information is generated and placed in the translated program.

**Algorithm 1. APA Tree Generation**

Input: A parallel_commands construct in a source program.
Output: An APA tree representing concurrent process nodes.
Method:

Initially make a root to represent a parallel_commands construct.
Repeat while (not end_of_construct in a source)

1. Parse a source program to identify concurrent processes.
   - If (find concurrent_process)
     - Push process level into the process stack.
     - If (same process_level with current node)
       - Make a sibling node.
     - Else if (lower process_level)
       - Parse parents node to find a node of same process level.
       - Generate an appropriate node.
     - Else if (higher process_level) { generate a child node; }
   - Else if (find end_of_process)
     - Pop process_level from the process stack;
   - Keep current information in each token to use at the code generation step;

Figure 4.a. A Simple APA Tree

Figure 4.b. A Process Node Structure
programs. An example translation is shown later. This example depicts a simple case. The parallel_commands construct can be more complicated, however. Such an example is considered in the next section.

3.1.2 Nested Structures

Parallel_commands constructs can be nested within another parallel_commands construct. Figure 5 shows a skeleton example of such a situation with a maximum nesting level of four.

In Figure 5, there are 10 nested processes in a program segment. Figure 6 shows the APA tree corresponding to the code segment. The parallel_commands construct has two processes in level 1, "1a" and "1b". Process "1a" in level 1 has two child processes, "2a" and "2b", and so on. The leaf nodes, "2a", "2b", "3a", "4a", "4b" and "3c", correspond to concurrent processes at the lowest level. Since they do not have any children, they can be executed in parallel. To facilitate processor allocation, this tree is transformed into a binary APA tree form. Figure 7 shows the transformed binary APA tree.

```
parallel_commands(2)
  | process(1a)           // level 1
  |     parallel_commands(2)
  |       process(2a)    // level 2
  |             end_process;
  |       process(2b)    // level 2
  |             end_process;
  |     end_parallel_commands;
  | process(1b)           // level 1
  |     parallel_commands(1)
  |       process(2c)    // level 2
  |         parallel_commands(3)
  |           process(3a) // level 3
  |             end_process;
  |           process(3b) // level 3
  |             end_parallel_commands;
  |             process(4a) // level 4
  |               end_process;
  |             process(4b) // level 4
  |               end_process;
  |     end_parallel_commands;
  | end_process;
| end_parallel_commands;
```

```
Figure 5. Example Codes with Nested Processes
```

```
Input: An APA tree.
Output: A transformed APA tree in which parallel processors are allocated to concurrent process nodes.
Method:
Initially transform input APA tree to the binary APA tree;
Traverse the tree to find out concurrent leaf nodes;
Calculate the total number of leaf nodes (= necessary_processors);
Get the total number of available processors in the system (= available_processors);
if (necessary_processors > available_processors)
call Process Node Boxing Algorithm (Algorithm 3);
Repeat traverse until finished:
  { if(leaf_node)
    { if (node is boxed)
      assign the first processor to the leaf node;
    } else
      assign next available processor to the leaf node;
    } else
      trace processor numbers of child nodes and keep information for using at the code generation step;
}
```

```
Algorithm 2. Process Mapping
```

After transforming an APA tree into a binary APA tree, the APA routine allocates processors to process nodes in the tree using the process mapping algorithm (Algorithm 2). The process mapping algorithm starts by checking leaf nodes in a depth-first-search and determines the nodes which may run concurrently. In this particular
example, the starred nodes ("*") in Figure 8 are those leaf nodes which may initially run concurrently at run-time. The APA routine assigns available processors to these leaf nodes first followed by interior nodes. The processor allocation is illustrated in Figure 8. In Figure 8, the numbers attached to the upper-right side of each leaf node represent the processor identifiers assigned to each leaf node. Once the processors are assigned to every leaf node, these numbers are traced from the leaf nodes to the parents. This processor assignment information is provided to the translator. The Parallel-C++ translator uses the information given by the APA routine to generate appropriate parallel codes in the translated program. The parallel codes consist of node programs for the iPSC/2 and they ensure correct control flow within the parallel commands.

In Figure 8, the numbers attached to the upper-right side of each leaf node represent the processor identifiers assigned to each leaf node. Once the processors are assigned to every leaf node, these numbers are traced from the leaf nodes to the parents. This processor assignment information is provided to the translator. The Parallel-C++ translator uses the information given by the APA routine to generate appropriate parallel codes in the translated program. The parallel codes consist of node programs for the iPSC/2 and they ensure correct control flow within the parallel commands.

3.1.3 Load Distribution with Limited Processors

In the preceding discussion, it was assumed that sufficient physical processors are available to be allocated to concurrent processes in the parallel_commands construct. However, this is not possible except in the case of very small programs. Therefore, algorithms need to be developed to map processes to processors in a many-to-one fashion. Figure 9 depicts a process load distribution scheme with a limited number of processors using the preceding example of the last section (Figure 5, 6, 7, 8). It is assumed that the total number of available processors in the system is four. Using a depth-first-search, the process node boxing algorithm (Algorithm 3) examines the first three leaf nodes until the total number of assigned processors becomes the same as the total number of available processors. The boxed nodes are the first three leaf nodes to be checked. The APA routine next assigns the first available processor to these nodes in a box, and assigns the remaining processors to the rest of the nodes (Algorithm 2). In this particular example, processor "0" is assigned to nodes "2a", "2b" and "3a", and processors "1", "2" and "3" are assigned to nodes "4a", "4b" and "3c", respectively. Those processors assigned to leaf nodes are traced to the upper-level parent nodes by the APA routine. The code generation routine will use this information to generate the necessary C++ code and system calls on the target machine. The next section outlines the concept of object migration and the associated translation scheme.

3.2 Object Migration

In our distributed object-oriented system, an object can move from one object to another. This is called object migration. There are two kinds of objects in Parallel-C++: (1) dynamic objects - objects that migrate, and (2) static objects - stationary objects (or processes) that do not migrate. Static objects reside at a certain process at a certain time, and dynamic objects float from a static object (or process) to another. This concept can be used for object migration. Parallel-C++ provides facilities to import and to export dynamic objects between objects in the system. Using "export / import" constructs, dynamic objects can travel around among static objects (or processes). Such a static object is called the owner of the corresponding dynamic object.
Communication models provide interactions between independent objects while preserving encapsulation and information hiding in a computational object. Export/import constructs enable synchronous communication between the objects (or processes) by using dynamic objects. Dynamic objects can be moved from one static object to another. Once a dynamic object is exported, then the importing object can access this incoming object, and the exporting object (or process) cannot access this object anymore. Not only has the logical address of this exported dynamic object been changed, but the address space of this object has also been changed in the distributed computing system. This imported object responds to the messages with the new environment provided by the enclosing object (or process). This basic conceptual model is illustrated in Figure 10, and the corresponding language construct is shown in Figure 11.

When an object moves to a new owner (importing object), its state is moved at the same time. The information kept in an object is retained and the integrity of that object is preserved as it moves. This model represents situations that occur in many real-world situations. These "export /import" constructs support object migration which can represent practical real-world problems such as air-traffic control systems, load balancing problems, and networking, etc. in a natural way. Export /import constructs in Parallel-C++ are translated using the scheme shown in Figure 12. Actual C++ codes with iPSC/2 C routines are shown in Figure 13. An importing object (or process) instantiates a new object of the same type to import a dynamic object from an exporting object (or process). The state of the new object is replaced by the state of the imported object. The exporting object (or process) sends the current state of the object by transferring the contents of this memory region from the current object (or process) to the object region in the importing object (or process). There may be a possible waiting time for the importing object. Once the message containing the exported object arrives at the buffer of the importing object (or process), the exporting object (or process) continues execution of the statements following the export statement.
those. When an airplane object is exported, the airport object importing such an airplane object takes over the state and information of the imported airplane. The translation is shown in Figure 15.

```
#include <stream.h>
class plane {
  int Name;
  int Destine;
  public:
    plane(int n, int d) {
      Name = n; Destine = d;
      cout << "Airplane " << obj(Name) << " instantiated.\n";
    }
    void take-off(int Port) {
      cout << "An airplane arrived.\n";
    }
    void landing(int Port) {
      cout << "is landing at\n";
    }
};
class airport {
  int Port; int Wing; int Destine;
  public: airport(int a, int p, int d);
  char* obj(int i);
};

class message-class {
  ...
};

airport::airport(int a, int p, int d) {
  Port = a; Wing = p; Destine = d;
  cout << "<obj>(Port)" << " instantiated.\n";
  plane Flight(Wing, Destine);
  printf("%s controls %s\n", obj(Port), obj(Wing));
  Flight.take-off(Port);
  export(Flight, process(Destine));
  x = import(plane);
  x.landing(Port);
}

char* obj(int i) {
  switch(i) {
    case 0: return("OKLAHOMA");
    ...
  }
}

main() {
  int OKLAHOMA = 0; int ARIZONA = 1; int TEXAS = 2;
  int PHOENIX = 4; int DALLAS = 5;
  int AMERICAN = 6; int UNITED = 7; int WESTERN = 8;
  parallel_commands(3) {
    process(OKLAHOMA) {
      airport OK-City(OK-CITY, AMERICAN, ARIZONA);
    }
    process(ARIZONA) {
      airport Phoenix(PHOENIX, UNITED, TEXAS);
    }
    process(TEXAS) {
      airport Dallas(DALLAS, WESTERN, OKLAHOMA);
    }
  }
}
```

Figure 14. An Example using Object Migration between Objects

```
#include "pcpp.h"
main() {
  getcube("Parallel-C++", "4", "default", 0);
  setpid(PC_HOST_PID);
  load("node", PC_ALL_NODES, PC_APPL_PID);
  message_class msg;
  csend(PC_MSG_TYPE_I, &msg, sizeof(msg), PC_ALL_NODES, PC_APPL_PID);
  csend(PC_MSG_TYPE_I, &msg, sizeof(msg), PC_ALL_NODES, PC_APPL_PID);
}
```

Figure 15. Translated Programs using C++ and iPSC/2 C routines
IV. CONCLUSION

We have presented an implementation scheme for the Parallel-C++ translator which has been investigated as a high-level language for object-oriented parallel and distributed programming. The translation scheme for the two key features among the several language constructs in Parallel-C++ were discussed. Those language constructs are the "parallelCommands" construct for explicit parallelism and the "export / import" constructs for object migration. The translation techniques described in this paper have been used in the implementation of the Parallel-C++ translator, which is currently running on the Intel iPSC/2.

Current implementation relies heavily on static information, such as the number of available processors in the system, given at translation time. The translation using dynamic information given at run-time is more useful in real programming. Another remaining problem of translation for explicit process allocation is load balancing, which improves node utilization. These refinements to the translator are considered as future work. Furthermore, based on the experience gained from implementing the translator, work on the development of a compiler is in progress.

REFERENCES