ISE-TR-92-99

A Parallel Programming Model and its Application by Integrating Imperative and Functional Evaluation Schemes

by

Myuhng Joo Kim, Chu Shik Jhon and Tetsuo Ida

August 3, 1992

A Parallel Programming Model and its application by integrating Imperative and Functional Evaluation Schemes

Myuhng Joo Kim*, Chu Shik Jhon* and Tetsuo Ida**

*Department of Computer Engineering
Seoul National University
**Institute of Information Sciences and Electronics,
University of Tsukuba

abstract

We propose a new parallel programming model by integrating imperative evaluation scheme and functional evaluation scheme, which are suited to sequential and parallel processing respectively. In this model, a program is defined as a set of grains. Each grain is considered as a unit of sequential processing and is evaluated under the imperative(von Neumann) scheme. A functional evaluation scheme, a kind of pure parallel model, is used as the inter-grain evaluation scheme. Since several functional evaluation schemes can be specified on each grain according to its evaluation characteristics, more efficient parallel execution solution can be specified for a given problem. The practical application of the model is explained by Functional C language, which is derived from the conventional imperative C language.

Key words: communication system model, grain, evaluation scheme, demand-driven, data-driven

1 Introduction

Since a parallel processing system has many computing resources which deal with the partitioned segments of a program independently under some evaluation scheme, it can be considered as a kind of communication system. That is, processing elements in a parallel processing system communicate some information each other (or one another) in order to solve a given problem. Consequently, it is necessary to construct a more elaborate investigation of communication system models both from theoretical and practical viewpoint if one hopes to develop more efficient parallel processing system.

Several communication system models have been developed until now: Petri Nets[1], CSP[2], ACP[3], Γ-calculus[4], π-calculus[5], and so on. These models can be classified into three groups by mobility, a property of changing the topology (i.e., communication structure) while some communication is going on. While Petri Nets, CSP, and ACP do not support this property directly. π-calculus can support it directly. Γ-calculus lies between these two groups. Since all the above-mentioned models are developed with the conventional von Neumann evaluation scheme mind, they treat each communication subject as a process. Because of this background, they take account of the side effect such as scope extrusion[6].

In this paper, we propose *G-system* (Grain-manipulating system), which is a new communication system model developed under a non-von Neumann evaluation scheme, especially functional evaluation scheme. The functional evaluation scheme, which emerged as a new candidate for future parallel processing around the end of the 1970's, exhibits several attractive merits: asynchronism of parallel processing, implicit parallelism, referential transparency, and so on[7,8]. In G-system, a parallel program is considered as a set of grains which possess mathematical functionality. The functional evaluation scheme is used to specify the inter-grain communication. The conventional imperative(von Neumann) evaluation scheme, more efficient to sequential processing than any other scheme, is used within a grain. We begin by giving an intuitive view of two major factors in G-system as a communication system model.

2 Basic concepts of a communication system model

2.1 Communication subject: Grain

Two basic concepts that constitute a communication system are the communication subject and the communication protocol. In G-system, we call the communication subject *grain*. A grain is a sequential evaluation unit. This sequentiality means that conventional control-driven evaluation scheme is used within a grain. So a grain is thought as a sequence of commands. A grain can be divided into atomic units. Each atomic unit corresponds to a command, which can be a grain itself. Thus a grain can be seen as a molecule consisting of atomic units. In the following example, grain C contains all the atomic units of both grain A and grain B:

```
A (x,y) ≡  x + y
B (x,y) ≡  x * y
C (x,y) ≡  if x > y then x + y else x * y
        ≡  if x > y then A (x,y) else B (x,y)
```

A grain can be considered as a mathematical function. So multiple calls of a grain with the same argument values always produce the same result value. This means that, in G-system, the functional evaluation scheme is used as a communication protocol.

While executing a program, several grains can have the same sequence of commands with different argument values. Both the calls of the same grain and the usage of a standard operator(e.g., +, -, <, ···) correspond to this situation. This observation leads to some distinct notions: (*grain*) *definition* and (*grain*) *instance*.[1] All the communication subjects to solve a given problem are described by definitions. During the execution of this program, each definition can be called and made into an instance. So a definition is viewed as the seed of several instances. That is, an instance is a process of executing a definition under its own environment. We call the conversion of a definition into an instance *generation*.

1) Hereafter, we will simply omit the first word *grain* from these notions. And we will frequently use the word *grain* instead of *instance* except when some confusion may occur. Most explanations on *grain* are closely related with (*grain*) *instance*.

2.2 Communication protocol: Functional evaluation scheme

The other basic concept as a communication system is the communication protocol. In G-system, this can be specified by two complementary ways: *equation* and *argument specification.*

2.2.1 Equation for connectivity

An equation specifies connectivity which is data dependency between(or among) several instances. A unique variable, called *a grain variable,* is assigned to each instance. This grain variable appears in the left-hand side of an equation. By referring to other grain variables in the right-hand side, an equation specifies connectivity. A grain variable used in several equations represents *grain sharing.* To maintain functionality, all the grain variables must appear only once in the left-hand side of an equation. Thus this rule is called *a single assignment rule.*

2.2.2 Argument specification for driving condition

Driving conditions state the requirements for evaluation of instances. Depending on the driving force, the functional evaluation scheme are classified into two categories: *a demand-driven scheme* and *a data-driven scheme.* The former is superior to the latter in the utilization of computing resources, while the latter is superior in the degree of exploitation of parallelism. To increase the probability of obtaining more efficient parallel execution solutions, both of them are provided in G-system. In order to evaluate grains of large size more efficiently, however, we have refined these two schemes into four primitive evaluation schemes and their hybrid evaluation schemes. Four primitive evaluation schemes are *lazy evaluation*, *predemand evaluation*, *total data-driven evaluation*, and *partial data-driven evaluation.*

Under the lazy evaluation scheme, a grain propagates demands to its arguments only when they are needed. This is the usual demand-driven evaluation scheme. Under the predemand evaluation scheme, a grain propagates demands in anticipation of its arguments which are expected to be evaluated. Since this increases parallelism, we can reduce the computing time. Under the total data-driven evaluation scheme, a grain can be evaluated only when all the arguments are ready. This is the usual data-driven evaluation scheme. Under the partial data-driven evaluation scheme, a grain can be evaluated when only the requisite arguments are ready. If a grain can be evaluated without arguments that are not requisite, we can also reduce the computing time. These driving conditions are described in the argument specification of a definition.

We will give more detailed and formal explanation of the communication subject and protocol with the full syntax of G-system in the following section.

3 G-system

G-system deals with four objects: *definition*, *equation*, *instance*, and *message*. Thus G-system is defined as a quadruple set $\langle \mathcal{D}, \mathcal{E}, \mathcal{I}, \mathcal{M} \rangle$, where $\mathcal{D}$, $\mathcal{E}$, $\mathcal{I}$, and $\mathcal{M}$ denote the set of definitions, the set of equations, the set of instances, and the set of messages respectively. For all objects of G-system, we will explain their basic property and syntax with some examples.

3.1 Definition: $\mathcal{D}$

A definition is composed of three parts: definition name, argument specification, and grain body.

<Definition> ::= <DefName> ≡ <ArgSpec> <GrainBody>

The definition name is an identifier given to a definition. Thus each grain is identified by the definition name. We can treat all the standard operators as definition names. When a standard operator is used as a definition name, we call it *grain operator* and distinguish it from the standard operator by putting a prime (′). The operator prefixed (or postfixed) means that it is evaluated under the data-driven (or demand-driven) scheme.

<DefName> ::= <Identifier> | <GrainOpr>
<GrainOpr> ::= ′ <StandardOpr> | <StandardOpr> ′

Infix standard operators are converted into prefix grain operators for syntactic uniformity. Two examples for a grain operator '$*$' are given in example 2 with their evaluation scheme. An argument conceptually serves as a channel to receive some value from another instance.

To support four primitive functional evaluation schemes, an argument can be one of three types: *data-driven*, *demand-driven*, and *capped*. A data-driven argument waits for some value to arrive. A demand-driven argument propagates a demand signal when it is needed. A capped argument is a list of demand-driven arguments which are capped with a special symbol γ. This γ is a kind of data-driven argument which receives a demand signal generated from some other grain. When a demand signal arrives at this grain (i.e., γ of this grain receives a data of demand signal), this capped argument is uncapped and behaves like a list of demand-driven arguments. This supports the demand-driven evaluation scheme in that all the demand-driven arguments can be demanded only after their corresponding grains receive demand signals from other grains. Note that a capped data-driven argument is useless. Syntactically, arguments proceeded by '?' are demand-driven type and arguments without '?' are data-driven type. Demand-driven arguments capped by γ are capped type. The behavior of each argument type is summarized by transition rules in section 3.5.

Depening on the time when the arguments are needed, they can be classified into two classes: ***requisite*** and *optional*. All the requisite arguments must be satisfied before the corresponding grain starts to be evaluated, but the optional arguments need not be satisfied. The meaning of ***satisfying an argument*** is dependent on the argument type. For a data-driven argument, some value for this argument must be available. For a demand-driven argument, a demand signal must be propagated to other instance whose name is that argument. For a capped argument, after γ receives a demand signal, all the demand-driven arguments must generate their demand signals. This is the semantics of the functional evaluation scheme. Syntactically, all the requisite arguments are put in the left part of argument specification, and all the optional arguments are in the right part. Thus the grain body can be evaluated only after all the arguments in the left part are satisfied. Arguments in the right part can be moved into the left part if they are needed during the evaluation of instance body and their values are not ready. The syntax of the argument specification is summarized as follows:

```
<ArgSpec>          ::=  [ <ArgList> | <ArgList> ]
<ArgList>          ::=  ε | <ArgList> <Arg>
<Arg>              ::=  <DataArg> | <DemandArg> | <CappedArg>
<DataArg>          ::=  <Var>
<DemandArg>        ::=  ? <Var>
<CappedArg>        ::=  γ | γ(<DemandArgList>)
<DemandArgList>    ::=  <DemandArg> | <DemandArg> <DemandArgList>
```

Example 1. In G-system, a grain can be specified by several functional evaluation schemes, which are classified into four basic functional evaluation schemes. Furthermore, many hybrid evaluation schemes are possible. For a grain f(x,y,z), these are specified as follows:

(a) *Lazy evaluation*: f ≡ [γ | ?x ?y ?z] 《 ······ 》
When a demand signal is propagated to this grain f, its body starts to be evaluated. If an argument is needed during evaluation of the grain body, a demand is propagated to a grain which is designated by the argument.

(b) *Predemand evaluation*: f ≡ [γ (?x) | ?y ?z] 《 ······ 》
When a demand signal is propagated to this grain f, the demand for argument x is pre-propagated and the grain body starts to be evaluated under lazy evaluation on y and z and under partial data-driven evaluation on x.

(c) *Total data-driven evaluation*: f ≡ [x y z |] 《 ······ 》
When all the values of x, y, and z arrive at the grain f, its body starts to be evaluated.

(d) *Partial data-driven evaluation*: f ≡ [y z | x] 《 ······ 》
When the values of y and z arrive at the grain f, its body starts to be evaluated. If the value of x does not arrive when x is needed in the grain body, f will be suspended until the value of x is ready.

(e) *Hybrid evaluation 1*: f ≡ [γ x y | ?z] 《 ······ 》
If the values of x and y are ready when a demand signal is propagated to the grain f, the grain body starts to be evaluated under lazy evaluation on z.

(f) *Hybrid evaluation 2*: f ≡ [x y | γ(?z)] 《 ······ 》
When the values of x and y arrives at the grain f, its body starts to be evaluated. Only after a demand to f is propagated, argument z can propagate a demand when it is needed.

The grain body is a list of commands. Since these commands are evaluated sequentially, the order of the commands in a sequence is important.

```
<GrainBody>    ::=  《 <CmdList> 》
<CmdList>      ::=  <Cmd> | <CmdList>; <Cmd>
```

At this moment, we have the following symple syntactic structures for the commands in a grain body. This is because we want to avoid complex reasoning with programs in the grain body. Of course, the grain body should be specified in a more complexed language if we wished.

```
<Cmd>    ::=  <Var> := <Exp>
            | if (<ConditionalExp>) (<CmdList>) (<CmdList>)
            | goto <Label>
            | <Label> : <Cmd>
            | skip
            | invoke { <EqnSet> }
```

The first command is to assign the value of an expression <Exp> to a variable. Since <Exp> is rather standard, its explanation is omitted here. Full syntax for <Exp> is summarized in the appendix.

```
<Exp> ::= <SimpleExp> | <ArithmeticExp> | <BooleanExp> | <ComparisonExp>
```

The second command is the conditional statement. The third and the fourth commands are used for unconditional branch. The fifth command is to pass over the current control. The last command is to invoke a set[2] of equations.

```
<EqnSet>  ::=  <Equation> | <EqnSet> , <Equation>
```

These invoked equations will generate instances by referring to definitions. The explanation of equations is given in section 3.2.

Example 2. Two grain operators for a standard operator '$*$' can be defined as follows:

(a) $'*$ ≡ [x y |] 《 $'*$:= x $*$ y 》 (data-driven evaluation)

(b) $*'$ ≡ [γ | ?x ?y] 《 $*'$:= x $*$ y 》 (demand-driven evaluation)

Example 3. The following definition SequentialFact is a grain for computing the factorial of a number. Note that this grain is evaluated under total the data-driven evaluation scheme and invokes no equation in its body.

```
SequentialFact ≡ [ low high | ] 《 index := low;
                                   SequentialFact := 1;
                                   loop: SequentialFact := SequentialFact * index;
                                         index := index + 1;
                                         if (index > high) (skip) (goto loop) 》
```

2) Since the single assignment rule is kept by all the equations, we use *set* concept instead of *list*.

Example 4. **LazySeqFact**, the lazy evaluation version of **SequentialFact**, is as follows:

```
LazySeqFact ≡ [ γ | ?low ?high ] 《index := low;
                                   LazySeqfact := 1;
                                   loop: LazySeqFact := LazySeqFact * index;
                                         index := index + 1;
                                         if (index > high) (skip) (goto loop) 》
```

Example 5. We give another example of computing factorial. The value of **ParallelFact(1,n)** is the factorial of **n**. To parallelize this grain, we adopt *divide & conquer* method. If the given number is larger than some threshold value (say, 100), then its factorial is computed by multiplying factorials of its two subintervals, which is supported by *invoke* command (i.e., parallelism is provided by *invoke* command). Otherwise, sequential method (e.g., **SequentialFact** in example 3) is used to compute the factorial.

```
ParallelFact ≡ [ low high . | ]
               《 if (low == high)
                    (ParallelFact := low)
                    (high1 := high - 1;
                     if (low == high1)
                        (ParallelFact := low * high)
                        (diff := high - low;
                         if (diff <= 100)
                            (invoke (ParallelFact := SequentialFact low high))
                            (sum := low + high;
                             mid := sum / 2;
                             mid1 := mid + 1;
                             invoke (v1 = ParallelFact low mid,
                                     v2 = ParallelFact mid1 high);
                             ParallelFact := v1 * v2 ))) 》
```

Example 6. We will treat the following definitions as some special cases. The definition **buffer** is a kind of buffer under the data-driven evaluation scheme. The definition **lazybuffer** is a kind of buffer under the demand-driven evaluation scheme. The definition **cond** is a nonstrict version of conditional buffer under the partial data-driven evaluation scheme. The definition **lazycond** is a nonstrict version of conditional buffer under the lazy evaluation scheme.

```
(a) buffer     ≡ [ anything |  ] 《 buffer := anything 》
(b) lazybuffer ≡ [ γ | ?anything ] 《 lazybuffer := anything 》
(c) cond       ≡ [ condvar | truepart falsepart ]
                 《 if (condvar == true) (cond := truepart) (cond := falsepart) 》
(d) lazycond   ≡ [ γ | ?condvar ?truepart ?falsepart ]
                 《 if (condvar == true) (lazycond := truepart) (lazycond := falsepart) 》
```

3.2 Equation: ε

An equation is syntactically similar to the assignment command. An equation, however, must observe the single assignment rule, which is manifested by '=' instead of ':=' in syntax. A special variable **$answer** should appear only once in the left-hand side of an equation as the grain variable. An equation starting with '?' in the right-hand side indicates that its corresponding instance will be generated with the data of demand signal.

```
<Equation>        ::=  <GrainVar>  =    <DefName> <ActualArgList>
                    |  <GrainVar>  =  ? <DefName> <ActualArgList>
<GrainVar>        ::=  <Var> | $answer
<ActualArgList>   ::=  ε | <ActualArgList> <ActualArg>
<ActualArg>       ::=  <Var> | <Value>
```

Example 7. The following grain **LazyParFact** is the lazy version of **ParallelFact**. Two parallel instances are generated by *invoke* command. Note that symbol '?' is specified in front of two grain names, which means that their corresponding instances will be generatred with demand signals from the beginning.

```
LazyParFact ≡ [  γ  |  ?low  ?high  ]
                  《 if  (low  ==  high)
                         (LazyParFact :=   low)
                         (high1 :=  high  -   1;
                          if  (low  ==  high1)
                              (LazyParFact :=  low  *  high)
                              (diff :=    high  -  low;
                               if  (diff <=  100)
                                   (invoke  (LazyParFact :=  LazySeqFact  low  high))
                                   (sum :=   low   +  high;
                                    mid :=   sum   /   2;
                                    mid1 :=   mid   +    1;
                                    invoke  (v1   =  ?  LazyParFact  low  mid,
                                             v2   =  ?  LazyParFact  mid1  high);
                                    LazyParFact :=   v1   *  v2 ))) 》
```

Example 8. To compute the factorial of a number (for example, 250) by using **SequentialFact** of example 3, **LazySeqFact** of example 4, **ParallelFact** of example 5, or **LazyParFact** of example 6, we must specify equations respectively as follows:

```
(a) leftval   =  buffer  1,                          (for SequentialFact)
    rightval  =  buffer  250,
    $answer   =  SequentialFact  leftval  rightval
(b) leftval   =  lazybuffer  1,                        (for LazySeqFact)
    rightval  =  lazybuffer  250,
    $answer   =  ? LazySeqFact  leftval  rightval
(c) leftval   =  buffer  1,                             (for ParallelFact)
    rightval  =  buffer  250,
    $answer   =  ParallelFact  leftval  rightval
(d) leftval   =  lazybuffer  1,                         (for LazyParFact)
    rightval  =  lazybuffer  250,
    $answer   =  ? LazyParFact  leftval  rightval
```

3.3 Instance: $\mathcal{I}$

Instances represent the dynamic behavior of a program. An instance is generated from an equation by referring to some adequate definition. It is defined as follows:

```
<Instance> ::= <GrainVar> ← <ArgSpec> <InsBody> <EnvSpec> <OutSpec>
```

Generation of an istance from an equation is a kind of replacement procedure. The definition name is replaced by the variable in the left hand side of the equation i.e., the grain variable. *Argument specification* <ArgSpec> is formed through replacing the arguments of definition by the corresponding variables in the right-hand side of equation. *Instance body* <InsBody> has the same command sequence that the definition body has, except that a *notifier* '#' is placed in front of the first command and that the replacement as above-mentioned has occurred. This notifier # indicates where the instance must start to be evaluated after the current driving condition is satisfied. Thus if an instance is just generated, # is placed on the first command. Otherwise, # is placed on some other command in the instance body.

```
<InsBody>       ::=  《 <InsCmdList> 》
<InsCmdList>    ::=  #<Cmd> | <Cmd> | <InsCmdList> ; <Cmd>
```

Environment <EnvSpec> is used to maintain the binding information of arguments, local variables, and the grain variable. If a value is transmitted from another instance to an argument of this instance, this binding information is kept in environment and the bound argument is removed from argument specification. If either a local variable or the grain variable is bound to a value during the evaluation of the instance body, this information is recorded in the environment, too.

```
<EnvSpec>       ::=  { <EnvList> }
<EnvList>       ::=  <Env> | <EnvList> <Env>
<Env>           ::=  (<Var> : <Value>)
```

Output specification <OutSpec> reserves all the addresses of other instances where the result value is sent. To support grain sharing under several functional evaluation schemes, these addresses are classified into two groups: *requisite* and *optional*. When the result value is generated, it should be sent to all the requisite addresses. For the optional addresses, it is sent only when a demand is propagated from them. Note that a special grain variable $system is used to send the final result to G-system.

```
<OutSpec>       ::=  [ <AddrList> | <AddrList> ]
<AddrList>      ::=  ε | <AddrList> <Addr>
<Addr>          ::=  <GrainVar> | $system
```

Example 9. From the grain equations of example 8(a) and the definitions of example 3 and 6(a), instances are generated as follows:

```
leftval    ←  [ | ] 《 #leftval  :=  anything 》 { anything:1 } [ $answer | ]
rightval   ←  [ | ] 《 #rightval :=  anything 》 { anything:250 } [ $answer | ]
$answer    ←  [ leftval rightval | ]   《 #index :=  leftval;
                                          $answer :=  1;
                                          loop: $answer :=  $answer * index;
                                                index :=  index  +  1;
                                                if (index > rightval) (skip) (goto  loop)
                                        》 { } [ $system | ]
```

3.4 Message: $\mathcal{M}$

In G-system, an instance sends some binding information to another instance by generating a message. A message is syntactically described by x ! (y : z), where x is the destination grain variable and y : z is the contents of message. Depending on the binding information, messages are classified into two types: *argument binding* and *address binding*. An argument binding message contains information of what value is bound to the designated argument, and an address binding message contains information where the result value must be sent. The argument binding message supports the data-driven evaluation scheme, and the address binding message supports the demand-driven evaluation scheme.

```
<Message>       ::=   <ArgBindMsg> | <AddrBindMsg>
<ArgBindMsg>    ::=   <DestGrainVar> ! <Env>
<AddrBindMsg>   ::=   <DestGrainVar> ! <AddrInform>
<AddrInform>    ::=   ( γ : <ReturnGrainVar> )
```

Example 10. Since all the instances of example 9 are under the data-driven evaluation scheme, we can have only three argument binding messages as follows:

```
$answer ! ( leftval : 1 )                   (a message from leftval to $answer)
$answer ! ( rightval : 250 )                (a message from rightval to $answer)
$system ! ( $answer : factorial(250) )      (a message from $answer to $system)
```

3.5 Theoretical development

In this section, the whole behavior of G-system will be briefly and formally explained by transition rules. First of all, we will introduce some preliminary abbreviations and definitions that are used in transition rules.

3.5.1 Preliminary abbreviations and definitions

(1) Instance

By referring to section 3.3, we can abbreviate the syntax of an instance as follows:

$$x \leftarrow A\, B_t\, E\, O$$

x is a grain variable, which lies in the left-hand side of the corresponding equation. A is the argument specification, which can be defined by the juxtaposition of left (requisite) part A_l and right (optional) part A_r, i.e., $(A_l \mid A_r)$. B_t is the grain instance body at the execution time t. It can be followed by a window <a_command>, which shows the currently executed command. The command part of the window can be empty (syntactically, $B_t < \varepsilon >$). This means that instance body has been executed completely. Two special command notations can appear within the window: Cmd[?x] and Cmd[!x]. Cmd[?x] means a command reading the value of a variable x. Cmd[!x] means a command writing the value of a variable x.

(2) Generation from an equation to an instance by referring to a definition

An instance (i_k) is generated from an equation(e_k) by referring to some adequate definition($\mathcal{D}$). This is described by the following notation.

$$i_k \leftarrow e_k \,\|\, \mathcal{D}, \quad k \in \{\, 1,\ 2,\ \cdots,\ n \ \text{ where } n \text{ is the number of equations} \,\}$$

This generation procedure is composed of following several actions.

○ All the identifiers in a definition are replaced by the corresponding identifiers in an equation. The definition name is replaced by the grain variable.

○ If an argument in a definition is replaced by a value, then this binding information is added into the environment specification of this instance.

○ The notifier '#' is specified in front of the first command of the instance body.

○ For the equation having '?' in the right-hand side, γ is uncapped in its instance.

○ Connectivity information is given within the output specification through the graph analysis of all the equations.

(3) Miscellaneous notations

GV(e_j) is the grain variable generated by an equation e_j. *RLT*(Cmd) is the result value generated by executing the right-hand side of a command 'Cmd'. $V_{out}(\{e_1,e_2,\cdots,e_n\})$ is the set of external output arguments for equations $e_1,e_2,\cdots,e_n$. $V_{in}(\{e_1,e_2,\cdots,e_n\})$ is the set of external input arguments for equations $e_1,e_2,\cdots,e_n$. $\perp$ is an instance within erroneous state.

3.5.2 Transition rules

The transition rules of G-system are grouped into four classes: *generation*, *propagation*, *arrival*, and *evaluation*. Generation class contains one rule: (1). Rule 1 is for the generation of equations into instances by referring to some adequate definitions. Propagation class contains three rules: (2) ~ (4). Rules 2 and 3 are for data propagation and rule 4 is for predemand propagation. If the result of an instance has been propagated into all the grain instances listed in its output specification, then this instance must be destroyed i.e., garbage-collected (rule 2). Otherwise, it cannot be destroyed (rule 3). Arrival class contains four rules: (5) ~ (8). Rules 5, 6, and 7 are for demand arrival and rule 8 is for data arrival. Evaluation class contains five rule: (9) ~ (14). Rule 9 is for the invoke command. Rule 10 is for the partial data-driven scheme. Rules 11 and 12 are for the demand-driven scheme. Rules 13 and 14 are for reading and writing operations respectively. Note that many internal actions effected by several commands in the grain body have not been described by the transition rules. The titles of these fourteen rules are listed as follows:

(1) generation of instances
(2) propagation of data with destructiveness
(3) propagation of data with nondestructiveness
(4) propagation of predemand
(5) arrival of demand for demand-driven argument
(6) arrival of demand for requisite/capped argument
(7) arrival of demand for optional/capped argument
(8) arrival of data for data-driven argument
(9) evaluation of 'invoke' command
(10) evaluation for partial data-driven argument
(11) evaluation for demand-driven argument
(12) evaluation for capped argument
(13) evaluation with reading error
(14) evaluation with environment addition

Now transition rules are specified as follows:

$$\frac{\langle\ \{e_1,e_2,\cdots,e_n\},\ \mathcal{D},\ \mathcal{I},\ \mathcal{M}\ \rangle}{\langle\ \emptyset,\ \mathcal{D},\ \mathcal{I}\cup\{\ i_1,i_2,\cdots,i_n\ \},\ \mathcal{M}\ \rangle} \tag{1}$$
$$\text{where } i_k \leftarrow e_k \| \mathcal{D},\quad k \in \{\ 1,\ 2,\ \cdots\ ,\ n\ \}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\emptyset|A_r)(B_t<\varepsilon>)E(O_1|\emptyset)\},\ \mathcal{M}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I},\ \mathcal{M}\cup\{y_i!(x:a)|\forall i,y_i\in O_1,(x:a)\in E\}\ \rangle} \tag{2}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\emptyset|A_r)(B_t<\varepsilon>)E(O_1|O_r)\},\ \mathcal{M}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\{\gamma\}|A_r)(B_t<\varepsilon>)E(\emptyset|O_r)\},\ \mathcal{M}\cup\{y_i!(x:a)|\forall i,y_i\in O_1,(x:a)\in E\}\ \rangle} \tag{3}$$
$$\text{where } O_r \neq \emptyset$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (A_1\cup\{?y\}|A_r)B_tEO\},\ \mathcal{M}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (A_1|A_r\cup\{y\})B_tEO\},\ \mathcal{M}\cup\{y_i!(\gamma:x)\}\ \rangle} \tag{4}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (A\cup\{\gamma\})B_tE(O_1|O_r)\},\ \mathcal{M}\cup\{x!(\gamma:y)\}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow AB_tE((O_1\cup\{y\})|(O_r-\{y\}))\},\ \mathcal{M}\ \rangle} \tag{5}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (A_1\cup\{\gamma(a_1\ a_2\ \cdots\ a_n)\}|A_r)B_tE(O_1|O_r)\},\ \mathcal{M}\cup\{x!(\gamma:y)\}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (A_1\cup\{a_1,a_2,\cdots,a_n\}|A_r)B_tE((O_1\cup\{y\})|(O_r-\{y\}))\},\ \mathcal{M}\ \rangle} \tag{6}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (A_1|A_r\cup\{\gamma(a_1\ a_2\ \cdots\ a_n)\})B_tE(O_1|O_r)\},\ \mathcal{M}\cup\{x!(\gamma:y)\}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (A_1|A_r\cup\{a_1,a_2,\cdots,a_n\})B_tE((O_1\cup\{y\})|(O_r-\{y\}))\},\ \mathcal{M}\ \rangle} \tag{7}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow AB_tEO\},\ \mathcal{M}\cup\{x!(y:a)\}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (A-\{y\})B_t(E\cup\{(y:a)\})O,\ \mathcal{M}\ \rangle} \tag{8}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\emptyset|A_r)(B_t<\mathrm{invoke}\{e_1,e_2,\cdots,e_n\}>)EO\},\ \mathcal{M}\ \rangle}{\langle\ \mathcal{E}\cup\{e_1,e_2,\cdots,e_n\},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (V_{out}(\{e_1,e_2,\cdots,e_n\})|A_r)B_{t+1}EO\},\ \mathcal{M}\cup\mathcal{M}'\ \rangle} \tag{9}$$
$$\text{where } \mathcal{M}' = \{z_{ij}!(z_i:a_i)|\forall(i,j),\ z_i\in V_{in}(\{e_1,e_2,\cdots,e_n\}),(z_i:a_i)\in E,z_{ij}\in \mathit{GV}(e_j)\}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\emptyset|A_r\cup\{y\})(B_t<\mathrm{Cmd}[?y]>)EO\},\ \mathcal{M}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\{y\}|A_r)(B_t<\mathrm{Cmd}[?y]>)EO\},\ \mathcal{M}\ \rangle} \tag{10}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\emptyset|A_r\cup\{?y\})(B_t<\mathrm{Cmd}[?y]>)EO\},\ \mathcal{M}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\{y\}|A_r)(B_t<\mathrm{Cmd}[?y]>)EO\},\ \mathcal{M}\cup\{y!(\gamma:x)\}\ \rangle} \tag{11}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\emptyset|A_r\cup\{\gamma(a_1\ a_2\ \cdots\ a_m\ ?y\ a_{m+1}\ \cdots\ a_n)\})(B_t<\mathrm{Cmd}[?y]>)EO\},\ \mathcal{M}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\{\gamma(?y)\}|(A_r\cup\{a_1,a_2,\cdots,a_n\}))(B_t<\mathrm{Cmd}[?y]>)EO\},\ \mathcal{M}\ \rangle} \tag{12}$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\emptyset|A_r)(B_t<\mathrm{Cmd}[?y]>)EO\},\ \mathcal{M}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\bot,\ \mathcal{M}\ \rangle} \tag{13}$$
$$\text{where } y \notin A_r$$

$$\frac{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\emptyset|A_r)(B_t<\mathrm{Cmd}[!y]>)EO\},\ \mathcal{M}\ \rangle}{\langle\ \mathcal{E},\ \mathcal{D},\ \mathcal{I}\cup\{x \leftarrow (\emptyset|A_r)B_{t+1}(E\cup\{y:\mathit{RLT}(\mathrm{Cmd})\})O\},\ \mathcal{M}\ \rangle} \tag{14}$$
$$\text{where } y = x \text{ or } y \notin A_r$$

4 Application example: Functional C

As a communication system model, G-system can be applied to two areas: *compiler target language* and *parallel programming language.* Since G-system supports both functional evaluation schemes and control of granularity of varying size, it can be used as a compiler target language of existing parallel programming languages, especially functional programming languages. Thus we can think of G-system as a compiler target language like DACTL[9].

Since the exploitation of parallelism leads to the significant advances in the future generation computer systems, many researchers strive to incorporate parallelism in their research areas. Central to the exploitation of parallelism is the design of the architectures for parallel processing and the new programming languages. Though each of them has independently surpassed the embryonic research stages, the effects of combining them into a system is by no means matured. One of the crucial reasons for the disharmony can be found in the trade-off relationship between parallelism and communication overhead with respect to granularity. For a given problem with fixed size, fine granularity increases parallelism but incurs heavy communication overhead. Therefore this issue of controlling parallelism can be converted into the issue of the optimization of the granularity[10]. Although the future parallel compilers may be designed to satisfy this issue and to generate optimal parallel programs, the current status of parallel compilers are not so satisfying. If a programmer can write a program by both binding sequentially executable parts into grains so as to reduce the communication overhead and specifying several evaluation schemes on grains so as to increase the degree of parallel processing, we can fill gaps between the optimal parallel programs of future and the inefficient parallel programs automatically generated by current compilers. Although this approach makes it difficult to design a program, programmers' knowledge in the problem domain can be reflected on parallel programming fully. From this point of view, we have suggested a new parallel programming language, **Functional C**[11], which is designed by applying G-system directly to conventional C programming language.

Example 11. We can describe all the versions of grain f(x, y, z) of example 1 in Functional C. Keyword 'grain' indicates that this function is a grain. Two keywords 'demand' and 'data' are used to specify the driving force of grain. A special symbol '#' is used to make an argument deviate from canonical meaning, i.e., # makes a demand-driven argument deviate from lazy into predemand and makes a data-driven argument deviate from total into partial. Assuming that x, y and z receive integer values and function f(x, y ,z) produces an integer value, all the versions of grain f(x, y, z) of example 1 can be programmed in Functional C as follows:

(a) *Lazy evaluation*

```
demand grain int f(x, y, z)
int x, y, z;
{ /* Conventional C programming is used */ }
```

(b) *Predemand evaluation*

```
demand grain int f(x, y, z)
int #x, y, z;
{ /* Conventional C programming is used */ }
```

(c) *Total data-driven evaluation*

```
data grain int f(x, y, z)
int x, y, z;
{ /* Conventional C programming is used */ }
```

(d) *Partial data-driven evaluation*

```
data grain int f(x, y, z)
int #x, y, z;
{ /* Conventional C programming is used */ }
```

(e) *Hybrid evaluation 1*

```
demand grain int f(x, y, z)
data int x, y;
int z;
{ /* Conventional C programming is used */ }
```

(f) *Hybrid evaluation 2*

```
data grain int f(x, y, z)
int x, y;
demand int #z;
{ /* Conventional C programming is used */ }
```

Example 12. The program of factorial has been explained in G-system by examples 3, 4, 5, 6, 7, and 8. In Functional C, the factorial can be programmed under the lazy evaluation scheme as follows:

```
main()
/* example 8(d) */
{    int leftval, rightval;
     leftval   =  LazyBuffer(1);
     rightval  =  LazyBuffer(250);
     return ( demand LazyParFact(leftval, rightval) );
}

demand grain int LazyBuffer( anything )
int anything;
/* example 6(b) */
{    return ( anything );   }

demand grain int LazyParFact( low, high )
int low, high;
/* example 7 */
{    if (low == high) return low;
     else if ( low == (high - 1) ) return (low  * high);
     else if ( (high - low) <= 100 ) return ( LazySeqFact(low, high) );
     else {  int mid, v1, v2;
             mid = (low + high) / 2;
             invoke ( v1 = demand LazyParFact(low, mid),
                      v2 = demand LazyParFact(mid+1, high) );
             return ( v1 * v2 );
}

demand grain int LazySeqFact( low, high)
int low, high;
/* example 4 */
{    int  index, sum;
     sum = 1;
     for (index = low; index <= high; index++)
         sum *= index;
     return ( sum );
}
```

5 Conclusion

In this paper, the skeleton of a new communication system model, G-system, is described together with its application example, Functional C. In G-system, a parallel program is treated as a set of grains which are sequentially executable parts. Thus conventional imperative evaluation scheme can be used within a grain. To utilize the merits of functional evaluation schemes in parallel programming, we adopt them as inter-grain evaluation schemes. The operating mechanism of G-system is explained by transition rules. Our target machine of G-system and compilation issues for Funcational C are explained in [11]. G-calculus, a calculus for G-system, is currently being studied together with versificationsystems. And we are investigating further both incorporating data parallelism in view of granularity of G-system and parallelizing conventional sequential programming languages with the features of G-system. We hope that G-system contributes to the manipulation of grains as parallel processing units in the future parallel computing systems.

References

[1] W. Reisig, Petri Nets, EATCS Monographs on Theoretical Computer Science, (eds. W. Brauer, G. Rozenberg, A. Salomaa), Springer Verlag, 1983.

[2] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[3] J. A. Bergstra, J. W. Klop, Algebra of Communication Process with Abstraction, Journal of Theor. Comp. Science, Vol.33, pp.77-121, 1985.

[4] G. Berry, G. Boudol, The Chemical Abstract Machine, ACM 089791-343-4/90/0001/0081, pp.81-94, 1990.

[5] R. Milner, Functions as Processes, from Automata, Language and Programming, Springer Verlag, pp.167-180, 1990.

[6] R. Milner, J. Parrow, D. Walker, A Calculus of Mobile Processes I & II, ECS-LFCS-89-85/86, University of Edinburgh, 1989.

[7] J. Backus, "Can programming be liberated from the von Neumann style? A Functional Style and its Algebra of Program", Communication of ACM, Vol.21, No.8, pp.613-641, Aug. 1978.

[8] P. Hudak, "Concepts, Evolution and Application of Functional Programming Languages", ACM Computing Surveys, Vol.21, No.3, pp.359-411, Sep. 1989.

[9] J. R. W. Glauert, J. R. Kennaway, M. R. Sleep, "Dactl: A Computational Model and Compiler Target Language based on Graph Reduction", ICL Technical Journal, pp.509-537, May 1987.

[10] C. McCreary, H. Gill, "Automatic Determination of Grain Size for Efficient Parallel Processing", Communications of ACM, Vol.32, No.9, pp.1073-1078, Sep. 1989.

[11] M. J. Kim, C. S. Jhon, "Functional C: An Extented Functional Programming Language", The 5th Jerusalem Conference on Information Technology, IEEE Computer Society Press, pp.298-302, Oct. 1990.

Appendix: the Syntax of G-system

1 Definition: $\mathcal{D}$

```
<Definition>       ::=  <DefName> ≡ <ArgSpec> <GrainBody>
<DefName>          ::=  <Identifier> | <GrainOpr>
<GrainOpr>         ::=  ´ <StandardOpr> | <StandardOpr> ´
<ArgSpec>          ::=  [ <ArgList> | <ArgList> ]
<ArgList>          ::=  ε | <ArgList> <Arg>
<Arg>              ::=  <DataArg> | <DemandArg> | <CappedArg>
<DataArg>          ::=  <Var>
<DemandArg>        ::=  ? <Var>
<CappedArg>        ::=  γ | γ(<DemandArgList>)
<DemandArgList>    ::=  <DemandArg> | <DemandArg> <DemandArgList>
<GrainBody>        ::=  《 <CmdList> 》
<CmdList>          ::=  <Cmd> | <CmdList>; <Cmd>
<Cmd>              ::=  <Var> := <Exp>
                     |  if (<ConditionalExp>) (<CmdList>) (<CmdList>)
                     |  goto <Label>
                     |  <Label> : <Cmd>
                     |  skip
                     |  invoke { <EqnSet> }
<Exp>              ::=  <SimpleExp>
                     |  <ArithmeticExp>
                     |  <BooleanExp>
                     |  <ComparisonExp>
<SimpleExp>        ::=  <Var> | <Value>
<ArithmeticExp>    ::=  <SimpleExp> <ArithOpr> <SimpleExp>
<BooleanExp>       ::=  <BoolOpr1> <SimpleExp>
                     |  <SimpleExp> <BoolOpr2> <SimpleExp>
<ComparisonExp>    ::=  <SimpleExp> <CompOpr> <SimpleExp>
<ConditionalExp>   ::=  <BooleanExp> | <ComparisonExp>
<StandardOpr>      ::=  <ArithOpr> | <BoolOpr1> | <BoolOpr2> | <CompOpr>
<ArithOpr>         ::=  + | - | * | /
<BoolOpr1>         ::=  not
<BoolOpr2>         ::=  and | or
<CompOpr>          ::=  == | <> | > | <
<EqnSet>           ::=  <Equation> | <EqnSet> , <Equation>
```

2 Equation: $\mathcal{E}$

```
<Equation> ::=  <GrainVar>  =    <DefName> <ActualArgList>
                  | <GrainVar>  = ? <DefName> <ActualArgList>
<GrainVar>      ::=  <Var> | $answer
<ActualArgList> ::=  ε | <ActualArgList> <ActualArg>
<ActualArg>     ::=  <Var> | <Value>
```

3 Instance: $\mathcal{I}$

```
<Instance>      ::=  <GrainVar> ← <ArgSpec> <InsBody> <EnvSpec> <OutSpec>
<InsBody>       ::=  《 <InsCmdList> 》
<InsCmdList>    ::=  #<Cmd> | <Cmd> | <InsCmdList> ; <Cmd>
<EnvSpec>       ::=  { <EnvList> }
<EnvList>       ::=  <Env> | <EnvList> <Env>
<Env>           ::=  (<Var> : <Value>)
<OutSpec>       ::=  [ <AddrList> | <AddrList> ]
<AddrList>      ::=  ε | <AddrList> <Addr>
<Addr>          ::=  <GrainVar> | $system
```

4 Message: $\mathcal{M}$

```
<Message>       ::=  <ArgBindMsg> | <AddrBindMsg>
<ArgBindMsg>    ::=  <DestGrainVar> ! <Env>
<AddrBindMsg>   ::=  <DestGrainVar> ! <AddrInform>
<AddrInform>    ::=  ( γ : <ReturnGrainVar> )
```

Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba 305, Japan

REPORT DOCUMENTATION PAGE	REPORT NUMBER ISE-TR-92-99

TITLE

A Parallel Programming Model and its Application by Integrating Imperative and Functional Evaluation Schemes

AUTHOR(S)

Myuhng Joo Kim[†], Chu Shik Jhon[†] and Tetsuo Ida[‡]
[†]Department of Computer Engineering, Seoul National University
[‡]Institute of Information Sciences and Electronics, University of Tsukuba

REPORT DATE August 3, 1992	NUMBER OF PAGES 20
MAIN CATEGORY Software	CR CATEGORIES D.3.2, D.1.1

KEYWORDS
communicating system model, grain, evaluation scheme, demand-driven computation, data-driven computation

ABSTRACT

We propose a new parallel programming model by integrating imperative evaluation scheme and functional evaluation scheme, whcih are suited to sequential and parallel processing respectively. In this model, a program is defined as a set of grains. Each grain is considered as a unit of sequential processing and is evaluated under the imperative (von Neumann) scheme. A functional evaluation scheme, a kind of pure parallel model, is used as the inter-grain evaluation scheme. Since several functional evaluation schemes can be specified on each grain according to its evaluation characteristics, more efficient parallel execution solution can be specified for a given problem. The practical application of the model is explained by Functional C language, which is derived from the conventional imperative C language.

SUPPLEMENTARY NOTES

This paper is an extended version of the paper "G-system: A functionality-based communication system model for parallel processing, IFIP '92"