Parallel Processing Letters, Vol. 11, Nos. 2 & 3 (2001) 363–374

DENOTATIONAL SEMANTICS OF AN HPF-LIKE DATA-PARALLEL LANGUAGE MODEL

MINYI GUO
Department of Computer Software, The University of Aizu
Aizu-Wakamatsu City, Fukushima 965-8580, Japan

Received May 2000
Revised January 2001
Accepted by Y. Muraoka

ABSTRACT

It is important for programmers to understand the semantics of a programming language. However, little work has been done about the semantic descriptions of HPF-like data-parallel languages. In this paper, we first define a simple language $\mathcal{D}$, which includes the principal facilities of a data-parallel language such as HPF. Then we present a denotational semantic model of $\mathcal{D}$. It is useful for understanding the components of an HPF-like language, such as data *alignment* and *distribution* directives, *forall* data-parallel statements.

1. Introduction

The data-parallel programming model, which is simple and portable across a large variety of parallel architectures, is very often used in parallel programming. There is a growing need for optimized compilers, or programming environments including parallelizing, data distributing and debugging tools. The data-parallel languages, such as HPF[4] and Fortran D[3], provide the parallel mechanism to execute the parallel statement (*forall* in HPF), and describe how data are distributed onto each local processor (in HPF, it is called *directive*). In the past recent years, much work have been accomplished about the formal semantics of sequential languages. Also, related to semantics of task-parallel (or control-parallel) programming model, some models such as Hoare's CSP, and Milner's CCS languages are proposed. In contrast with data parallelism, however, the semantics of data-parallel programming model are also competing to take the lead. To correctly understand the meanings of parallel components of a language such as HPF directives, is important for parallel programming or automatic generation of parallelizing compilers. Therefore, it is considered a significant work if we can present a formal semantic definition at "high level" for a data-parallel programming model.

In this paper, we intend to give a denotational semantic model for an HPF-like data-parallel language. Domain-theoretic classical denotational semantics provides efficient and well-understood techniques to reason about collections of programs. We

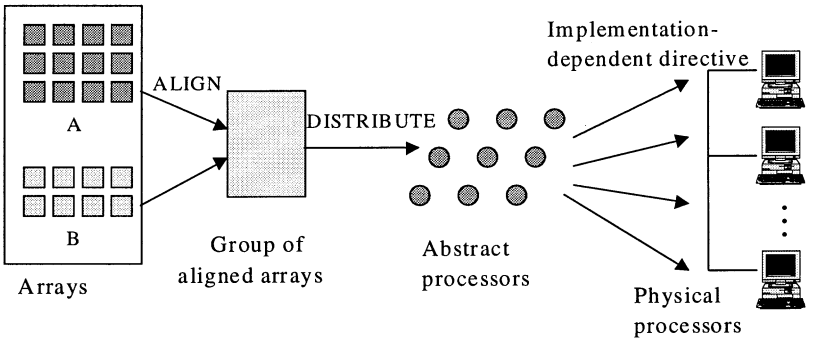

Figure 1: HPF data mapping pattern.

focus main attention on describing the semantics of those different with traditional sequential language components. we are interested in how to specify the semantics of declarations of virtual processors, templates, alignment and distribution directives and data-parallel control constructs. The foundations of denotational semantics are not concerned in the paper due to the lack of space but we explain our idea as detail as possible.

The rest of the paper is organized as follows: the abstract syntax of the data-parallel language $\mathcal{D}$, some general functions used throughout and concepts of denotational semantics are given in Section 2. Section 3 defines the details about the denotational semantics of $\mathcal{D}$, including the semantic domains and the semantic functions. In this section, we also present some examples for well-understanding the semantics of the data-parallel languages defined in this paper. In Section 4, an implementation framework is presented. Some related work and conclusions are given in Section 5 and Section 6 respectively.

2. Preliminaries

HPF[4] or Fortran D[3] supports data-parallel programming paradigm (The compiler translates the source code to the SPMD code) through two aspects of the new features including the data parallel execution features — FORALL construct and the data distribution features — PROCRSSORS directive, TEMPLATE directive, ALIGN directive, and DISTRIBUTE directive (We do not concern the realign and redistribute directives because they have similar semantic model with align and distribute directives).

To express parallel computation explicitly, the FORALL construct is to provide a convenient syntax for simultaneous assignments to large groups of array elements. Such assignments lie at the heart of the data parallel computations that HPF is designed to express. The multiple assignment functionality it provides is very similar to that provided by the array assignment statement in Fortran 90.

HPF data alignment and distribution directives allow the programmer to advise

the compiler how to assign array elements to processor memories. The model is that there is a two-level mapping of data objects to memory regions, referred to as "abstract processors". Data objects (typically array elements) are first aligned relative to one another; this group of arrays is then distributed onto a rectilinear arrangement of abstract processors specified by PROCESSORS directive (see Fig.1).

For the sake of simplicity, a common kernel of data-parallel languages, named language $\mathcal{D}$ is designed in this paper. We only consider the above parallel features and omit the scalar part of the sequential languages. Furthermore, we simplify the construct of ALIGN directive and DISTRIBUTE directive in HPF replaced by a united form "mapping statement". That is, an HPF align directive

$$\text{!HPF\$ ALIGN } array - expression \text{ WITH } template - expression$$

is replaced by a mapping statement

$$template - expression \ :- \ array - expression.$$

and an HPF distribute directive

$$\text{!HPF\$ DISTRIBUTE } template - expression \text{ ONTO } processor - name$$

is replaced by a mapping statement

$$processor - expression \ :- \ template - expression.$$

The advantage of this notation is that two level mapping features have a united notation, and BLOCK(n)/ CYCLIC(n) distribution can be expressed by subscript computation in the mapping statement.

Clearly, it only changes the representations but does not change the semantics of the directives. Formally the abstract syntax of the parallel mechanism part of $\mathcal{D}$ is defined as in Fig.2. For example, the data-parallel program of matrix multiplication written in $\mathcal{D}$ can be as follow. The mapping statements give the row-major order block distribution for the arrays a and c and column-major order block distribution for b.

```
 /* matrix.d */
#define n  64
#define pn 4

main()
{
  vp procs(pn);
  tmp s(n);
  array a(n,n), bt(n,n), c(n,n);
  int i,j,k;
```

parallel-mechanism	→	parallel-directive | parallel statement
parallel-directive	→	parallel-object-directive | mapping-statement
parallel-object-directive	→	processors-directive | template-directive | array-declaration
processors-directive	→	**vp** processors-name shape-spec-list
template-directive	→	**tmp** template-name shape-spec-list
array-declaration	→	**array** array-name shape-spec-list
mapping-statement	→	(processors-name | template-name) index-expression-list : – (template-name | array-name) index-expression-list
shape-spec-list	→	({ constant-expression }$^+$)
index-expression-list	→	({ index-expression |–}$^+$)
parallel-statement	→	**forpar** (forpar-header) assignment
forpar-header	→	index-name : '{' subscript .. subscript '}'

Figure 2: Abstract syntax of the parallel mechanism of $\mathcal{D}$

```
  s(i) :- a(i,-);
  s(i) :- bt(-,i);
  s(i) :- c(i,-);
  procs(i/16) :- s(i);

  forpar (j:{0..n-1}, k:{0..n-1}) {
      c(i,-) += a(j,-) * bt(-,k);
    }
}
```

The semantics of a statement is a transformation from one state to another. The main idea of the denotational semantics is that each phrase of the language is given a denotation: a mathematical object (functions, numbers, tuples, etc.) that represents the contribution of the phrase to the meaning of any complete program in which it occurs. A denotational semantics of a programming language gives the mapping from programs in the language to the functions denoted. There are three essential parts that we must present clearly. That is abstract syntax, semantic domains and semantic functions, where the semantic functions generally have the form $syntax - domain \rightarrow semantic - domain_1 \rightarrow \cdots \rightarrow semantic - domain_n$. The chief syntax domains of $\mathcal{D}$ which will be assigned a semantics involve *Exp*: (index) expression, *IeList*: index expression list, *Dec*: parallel object directive, *Mp*: mapping statement, and *Cmd*: statement.

The followings are some notations and functions will applied in the following sections thoroughly.

• A shape specification list (shape-spec-list, an n-tuple $(u_1, u_2, \ldots, u_n)$) declared in a parallel object directive $id(u_1, u_2, \ldots, u_n)$, where u_i is a constant (or a constant

$\sigma : Pro = Env \times S$	/* program state */
$s : S = Loc \rightarrow D$	/* statement state */
$\rho : Env = E_s \times E_m$	/* environment */
$\rho_s : E_s = Id \rightarrow D$	/* static environment */
$\rho_m : E_m = [Id \times Id \rightarrow Inx \times Inx]+$ $[Id \times Id \rightarrow (Inx \rightarrow Loc)]$	/* mapping environment */
$l : Loc = Prc \times Mem$	/* location */
$g : Iv = Iid \rightarrow N$	/* index value */
$p : Prc = Pid \times BS$	/* processor domain */
$\beta : Tmp = Tid \times BS$	/* template domain */
$\alpha : Arr = Aid \times BS$	/* array object domain */
$\delta : D = BS + Loc + Prc + Arr + Tmp + Id + \cdots$	/* denotative value */
$i : Inx = BS$	/* index */
$id : Id = Pid + Tid + Aid + Iid + \cdots$	/* identifier */

Figure 3: The semantic domains of $\mathcal{D}$

expression), defines a *bound set*
$BS = \{(a_1, a_2, \ldots, a_n) | (a_1, a_2, \ldots, a_n) \in \{1, \ldots, u_1\} \times \ldots \times \{1, \ldots, u_n\}\}$, which specifies the shape (rank and the extent in each dimension) of a virtual processor, a template, or an array.

The function $\mathbf{all}(id, k) = (0, \ldots, u_k - 1)$ represents the all elements of the $k - th$ dimension of the object id and the function $\mathbf{rank}(id, k) = u_k$ represents the rank of the $k - th$ dimension of the object id.

• Let f:X $\rightarrow$ Y and g:X $\rightarrow$ Z are both total, one-to-one functions, the joint function •: Y $\rightarrow$ Z is defined by

$$f \bullet g = \{(y, z) | f^{-1}(y) = f^{-1}(z)\}.$$

Let f:Y $\rightarrow$ Z and g:X $\rightarrow$ Y are both total, one-to-one functions, the composition function $\circ$: X $\rightarrow$ Z is defined by

$$(f \circ g)(x) = f(g(x)).$$

Let $F = \{f_1, \ldots, f_n\}$, where $F : X \rightarrow Y$, is a function with $f_i = (x_i, y_i) \in X \times Y$, and let also $f' = (x', y') \in X \times Y$, the update function $\lhd$ is defined as

$$F \lhd f' = \begin{cases} \{f_1, \ldots, f_{i-1}, f', f_{i+1}, \ldots, f_n\}, & \text{for } x' = x_i, \\ \{f_1, \ldots, f_n, f'\}, & \textit{otherwise.} \end{cases}$$

Let $F, G : X \rightarrow Y$ be the functions with $G = \{g_1, \ldots, g_n\}$, we define

$$F \lhd G = F \lhd g_1 \lhd \ldots \lhd g_n.$$

3. Denotational Semantics of $\mathcal{D}$

3.1. Semantic Domains

The semantic domains are some mathematical objects which correspond to the denoted components of a language. Figure 3 shows the semantic domains of $\mathcal{D}$ beginning from program state. We give the explanations for the meanings of these domains. The semantic domains are defined by using the notation "domain variable: domain name = the construct of domain".

The semantics $[\![P]\!]$ of a program P is a function from *program state* to *program state*. The program state Pro is made of an environment and a statement state. The statement state S is a mapping from a location to a designation value domain. This indicates a fact that is, after having executed a statement, the stored values in the location are changed. The environment Env is made of a static environment E_s and a mapping environment E_m, where E_m is a mapping from two identifier to a pair of index or to a location, and E_s is a static environment corresponding to the various declarations of a program. The store domain Loc indicates the concept of the local memory of a processor. The basic domains Prc, Tmp, and Arr are the domains of the processors directives, template directives and array declarations. The domains Mem and N are (virtual) memory and a set of natural number respectively. The denotative value domain involves all values used in programs, and the Inx domain is the scope of index which may be used in mapping statements or array subscript expressions.

3.2. Semantic Functions

As we mentioned in the above, the denotational semantics of a component of a language can be understood as a function of mapping from one domain to another. In our framework, the semantic functions involve three parts: semantic functions of the parallel object directives, the mapping statements and the parallel statement. Before the definitions of the semantic function for processors directive, template directive, and array declaration, we first define a semantic function for expression value evaluation which is used in the all of the following sections.

A semantic funtion $\mathbf{E}[\![exp]\!](\mathbf{E} : Exp \rightarrow S \rightarrow S)$ denotes the value obtained by evaluating expression exp in the static environment E_s and program state S. Furthermore, if $id(u_1, u_2, \ldots, u_n)$ is specified in a parallel object directive (**vp, tmp**, or **array**) in $\mathcal{D}$ and $id(ie_1, \ldots ie_n)$ is its index expression used in mapping statements or program statements, then $\mathbf{E}[\![(ie_1, \ldots ie_n)]\!] = (\mathbf{E}[\![ie_1]\!], \ldots, \mathbf{E}[\![ie_n]\!])$. Let $ie_k = -$ represent the all elements of the k-th dimension, then $\mathbf{E}[\![ie_k]\!] = \mathbf{all}(id, k)$.

In the following discussions, the notation pid, tid and aid represent processor name, template name and array name respectively, iel and jel are index expression list.

3.2.1. Semantic Functions of Parallel Directives

A parallel object directive (processors, template or array) is a part of declarations

which modifies the current static environment. We might give the semantic function for parallel object directive as: $\mathbf{D} : Dec \rightarrow E_s \rightarrow E_s$.

$\mathbf{D}[\![\mathbf{vp}\ pid(u_1 \ldots u_n)]\!]_{\rho_s} \equiv \rho_s \lhd (pid, p)$, where $p \in Prc$.

$\mathbf{D}[\![\mathbf{tmp}\ tid(u_1 \ldots u_n)]\!]_{\rho_s} \equiv \rho_s \lhd (tid, \beta)$, where $\beta \in Tmp$.

$\mathbf{D}[\![\mathbf{array}\ aid(u_1 \ldots u_n)]\!]_{\rho_s} \equiv \rho_s \lhd (aid, (is, mem))$, where $is \in BS$ and $mem \in Mem$.

A processors directive specifies a virtual processor grid, and a template directive specifies an abstract space of indexed positions; it can be considered as an "array of nothings". An array directive declares a virtual memory space because in a parallel program, the global array will be distributed into local memory of processors through mapping statement. Therefore, all of these parallel object objectives change the static environment.

3.2.2. Semantic Functions of Mapping Statements

The mapping statements include two kinds:

- $am = tid\ iel : - aid\ jel$, which is used to specify that certain data objects are to be mapped in the same way as certain other data objects (An array aid is aligned with a template tid). Operations between aligned data objects are likely to be more efficient than operations between data objects that are not known to be aligned.

- $dm = pid\ iel : - tid\ jel$ or $pid\ iel : - aid\ jel$, which specifies a mapping of data objects to abstract processors in a processor arrangement.

The semantic function for a mapping statement is $\mathbf{M} : Mp \rightarrow E_m \rightarrow E_m$.

$\mathbf{M}[\![am]\!]_{\rho_m} \equiv \rho_m \lhd ((tid, aid), IP)$,

$IP : IeList \times IeList \rightarrow Inx \times Inx$,

$IP(iel, jel) = \{(\mathbf{E}[\![(ie_1, \ldots, ie_n)]\!], \mathbf{E}[\![(je_1, \ldots, je_m)]\!]) | 0 \leq \mathbf{E}[\![ie_k]\!] < t_k \wedge 0 \leq \mathbf{E}[\![je_{k'}]\!] < a_{k'}, 1 \leq k \leq n, 1 \leq k' \leq m\}$,

where $t_k = \mathbf{rank}(tid, k)$ and $a_{k'} = \mathbf{rank}(aid, k')$.

$\mathbf{M}[\![dm]\!]_{\rho_m} \equiv \rho_m \lhd ((pid, tid), SP)$,

$SP : IeList \times IeList \rightarrow Inx \times Loc$,

$SP(iel, jel) = \{\mathbf{E}[\![iel]\!], (\mathbf{E}[\![jel]\!], mem) | 0 \leq \mathbf{E}[\![ie_k]\!] < p_k \wedge 0 \leq \mathbf{E}[\![je_{k'}]\!] < t_{k'}, 1 \leq k \leq n, 1 \leq k' \leq m\}$,

where $t_{k'} = \mathbf{rank}(tid, k')$ and $p_k = \mathbf{rank}(pid, k)$.

$\mathbf{M}[\![am; am']\!]_{\rho_m} \equiv \mathbf{M}[\![am']\!]_{\rho_m} \circ \mathbf{M}[\![am]\!]_{\rho_m}$.

$\mathbf{M}[\![am; dm]\!]_{\rho_m} \equiv \rho_m \lhd AP$,

$AP = \{(aid, pid), IP \bullet SP | IP \in \rho_m(tid, aid) \wedge SP = \rho_m(tid, pid)\}$.

$\mathbf{M}[\![dm; dm']\!]_{\rho_m} \equiv \mathbf{M}[\![dm']\!]_{\rho_m} \circ \mathbf{M}[\![dm]\!]_{\rho_m}$.

3.2.3. Semantic Functions of Parallel Statement

The followings are some semantic functions that will be used in definition of the semantic functions for a parallel statement.

Let g be an index identifier evaluation function at one time of iterations(a state transition), for example, $g = (i, 1), g' = (j, 10)$, etc.. $G = \{g_0, \ldots, g_n\}$ represents the range of value assigned to an index identifier during whole iteration. The function **L**: $Aid \times iel \rightarrow Iv \rightarrow Env \rightarrow Loc$, which is used to determine the locations of index variables appeared in the left hand of assignment statements, is defined as $\mathbf{L}[\![aid\ iel]\!]_{g\rho} \equiv \lambda p.AP(aid, p)(\mathbf{E}[\![iel]\!])$.

The semantic function for an assignment is $\mathbf{C} : Cmd \rightarrow Iv \rightarrow S \rightarrow S$, $\mathbf{C}[\![c]\!]_{g\sigma} = \mathbf{C}[\![aid(ie_1, \ldots, ie_n) = v]\!]_{g\sigma} \equiv \sigma \lhd (\mathbf{L}[\![aid(ie_1, \ldots, ie_n)]\!]_{g\rho}, \mathbf{E}[\![v]\!])$.

The semantic function for sequential loop **LS**: $Cmd \rightarrow pow(Iv) \rightarrow S \rightarrow S$ is recursively defined as $\mathbf{LS}[\![c]\!]_{\phi\sigma} \equiv \sigma$, $\mathbf{LS}[\![c]\!]_{G\sigma} \equiv \mathbf{LS}[\![c]\!]_{(G\setminus\{g\})\sigma'}$, where $g = min(G)$ and $\sigma' = \mathbf{C}[\![c]\!]_{g\sigma}$.

The index range evaluation function **R**:$ir \rightarrow pow(Iv)$ is defined as $\mathbf{R}[\![iid : \{l..u\}]\!] \equiv \{g_0, \ldots, g_{u-l}\}$, where each g_i is an index identifier evaluation function such that $g_i = (iid, i + l)$.

Let $G = \{g_0, \ldots, g_n\}$ be a set of the index identifier evaluation functions and let $(j_0, \ldots, j_n)$ be an arbitrary permutation of $(0, \ldots, n)$. Now we can define the semantic functions for sequential loop and parallel loop as follows:

$\mathbf{C}[\![\textbf{forseq}\ (i : \{l..u\})\ c]\!]_{g\sigma} \equiv \mathbf{LS}[\![c(g)]\!]_{\mathbf{R}[\![ir]\!],\sigma}$. The semaitic function for parallel loop **LP**: $Cmd \rightarrow pow(Iv) \rightarrow S \rightarrow S$ is recursively defined as $\mathbf{LP}[\![c]\!]_{G\sigma} \equiv \sigma'$, where $\forall i \in \{0, \ldots, n\}.\sigma_i = \mathbf{C}[\![c]\!]_{g_i\sigma}$, $\sigma' = \sigma \lhd \sigma_{j_0} \lhd \ldots \lhd \sigma_{j_n}$.

$\mathbf{C}[\![\textbf{forpar}\ (i : \{l..u\})\ c]\!]_{g\sigma} \equiv \mathbf{LP}[\![c(g)]\!]_{\mathbf{R}[\![ir]\!],\sigma}$, where $ir = i : \{l..u\}$.

The instances of a **forpar** loop body are supposed to have no dependencies on each other. For this reason the semantics of the assignment in the body can be determined for each index vector independently. However, if dependencies do exist between the loop instances the order in which the assignments are executed determines the semantics of such a loop. Since the semantics require that the loop instances may occur in any order, the semantic function **LP** is non-deterministic in case of existing loop dependencies. This is a principal different from semantics of **forseq** loop. The latter is rigorously executed in lexicographic order.

3.3. Examples

Example 1 *For the following segment of a $\mathcal{D}$ program including parallel object directives and mapping statements.*

$$
\begin{array}{lcl}
p_1 & : & \textbf{vp}\ p(4); \\
t_1 & : & \textbf{tmp}\ s(10); \\
t_2 & : & \textbf{tmp}\ t(10, 10); \\
a_1 & : & \textbf{array}\ a(10); \\
a_2 & : & \textbf{array}\ b(10, 100, 10);
\end{array}
$$

$$\begin{aligned} m_1 &: s(i) :- a(i*2+1); \\ m_2 &: t(i,j) :- b(j-2,-,i+2); \\ m_3 &: p(i/4) :- s(i); \end{aligned}$$

The denotational semantics of this segment are
$\mathbf{D}[\![p_1]\!]_{\rho_s} \equiv \rho_s \lhd (p, \{0,1,2,3\})$,
$\mathbf{D}[\![t_1]\!]_{\rho_s} \equiv \rho_s \lhd (s, \{0,1,\ldots,9\})$,
$\mathbf{D}[\![t_2]\!]_{\rho_s} \equiv \rho_s \lhd (t, \{0,\ldots,9\} \times \{0,\ldots,9\})$,
$\mathbf{D}[\![a_1]\!]_{\rho_s} \equiv \rho_s \lhd (a, (\{0,1,\ldots,9\}, \{\mu_0,\ldots,\mu_9\})$,
$\mathbf{D}[\![a_2]\!]_{\rho_s} \equiv \rho_s \lhd (b, (\{0,\ldots,9\} \times \{0,\ldots,99\} \times \{0,\ldots,9\}, \{\mu_0,\ldots,\mu_{9999}\}))$, *where* μ_i *represent the (virtual) memory locations.*

$\mathbf{M}[\![m_1]\!]_{\rho_m} \equiv \rho_m \lhd ((s,a), IP_{m_1})$,
$IP_{m_1} = IP((i),(i*2+1)) = \{(\mathbf{E}[\![i]\!], \mathbf{E}[\![i*2+1]\!]) | 0 \le \mathbf{E}[\![i]\!] < s' \wedge 0 \le \mathbf{E}[\![(i*2+1)]\!] < a'\}$, *because* $s' = 10$ *and* $a' = 10$, *then*
$IP((i),(i*2+1)) = \{(i, i*2+1) | 0 \le i < 10 \wedge 0 \le i*2+1 < 10\} = \{(0,1),(1,3),\ldots,(4,9)\}$.

The following mapping statement illustrates the combination from an array to a template.
$\mathbf{M}[\![m_2]\!]_{\rho_m} \equiv \rho_m \lhd ((t,b), IP_{m_2})$,

$$\begin{aligned} IP_{m_2} &= IP((i,j),(j-2,-,i+2)) \\ &= \{((i,j), \mathbf{E}[\![(j-2,-,i+2)]\!] | 0 \le \mathbf{E}[\![(i,j)]\!] < s'_k \wedge 0 \le \\ &\quad \mathbf{E}[\![(j-2,-,i+2)]\!] < a'_k, 0 \le k < 2\} \\ &= \{(i,j),(j-2,(0,\ldots,99),i+2) | 0, 0 \le i,j < 10, 10 \wedge \\ &\quad 0 \le j-2 < 10 \wedge 0 \le i+2 < 10\} \\ &= \{((i,j),(j-2,(0,\ldots,99),i+2) | 0 \le i < 8 \wedge 2 \le j < 10\} \\ &= \{((0,2),((0),(0,\ldots,99),(2))),((0,3),((1),(0,\ldots,99),(2))),\ldots,((7,9), \\ &\quad ((7),(0,\ldots,99),(9)))\}. \end{aligned}$$

The following mapping statement illustrates how a template is distributed onto processors.
$\mathbf{M}[\![m_3]\!]_{\rho_m} \equiv \rho_m \lhd ((s,p), SP_{m_3})$,
$SP_{m_3} = SP((i),(i/4)) = \{\mathbf{E}[\![(i)]\!], (\mathbf{E}[\![(i/4)]\!], l) | 0 \le i < 10 \wedge 0 \le i/4 < 4\}$
$= \{((0),(0,l_0)),((1),(0,l_1)),(2,(0,l_2)),(3,(0,l_3)),\ldots,((8,(2,l_0)),((9,(2,l_1))\}$.

The following mapping statements thus determine the relationship of an array to the processors by the two-level mapping.
$\mathbf{M}[\![m_1; m_3]\!]_{\rho_m} \equiv \rho_m \lhd AP_{m_1 p_1}$,
$AP_{m_1 p_1} = \{(a,p), IP_{m_1} \bullet SP_{m_3} | IP_{m_1} \in \rho_m(s,a) \wedge SP_{m_3} = \rho_m(s,p)\}$
$= \{(a,p), \{(1,(0,l_0)),(3,(0,l_1)),(5,(0,l_2)),(7,(0,l_3)),(9,(1,l_0))\}\}$.

Example 2 *Consider the following sequential loop statement:*

$$\textbf{forseq}\ (i : \{1..2\})\ a(i) = a(i-1) * 2$$

Suppose a is an array of integers with integer a_i which is stored at location L_i at position i and if this statement is executed with program state $\sigma = ((L_0, 2), (L_1, 3), \ldots)$ (that is, array a has the initial value $a = (2, 3, 4, \ldots)$). Then the semantics of the statement when executed:

$$\begin{aligned}
&\mathbf{C}[\![\textbf{forseq}\ (i : \{1..2\})\ a(i) = a(i-1) * 2]\!]_{g\sigma} \\
&\quad \equiv\ \mathbf{LS}[\![a(i) = a(i-1) * 2]\!]_{\mathbf{R}[\![i:\{1..2\}]\!],\sigma} \\
&\quad =\ \mathbf{LS}[\![a(i) = a(i-1) * 2]\!]_{\{g_0,g_1\}\sigma},\quad (g_0(i) = 1, g_1(i) = 2) \\
&\quad =\ \mathbf{LS}[\![a(1) = a(0) * 2]\!]_{\{g_1\}\sigma'}\mathbf{LS}[\![a(2) = a(1) * 2]\!]_{\{\}\sigma''} = \sigma'',
\end{aligned}$$

where $\sigma' = \sigma \lhd (L_1, a_0 * 2)$, and $\sigma'' = \sigma' \lhd (L_2, a_1 * 2)$. Now, the program state $\sigma = ((L_0, 2), (L_1, 4), (L_2, 8), \ldots)$.

Example 3 *Consider the following parallel loop statement:*

$$\textbf{forpar}\ (i : \{0..9\}\ j : \{0..9\})\ a(i,j) = b(j,i)$$

Let $k_{0,0}, \ldots, k_{9,9}$ be an arbitrary permutation of $(0,0), \ldots, (9,9)$. Then

$$\begin{aligned}
&\mathbf{C}[\![\textbf{forpar}\ (i : \{0..9\}\ j : \{0..9\})\ a(i,j) = b(j,i)]\!]_{g\sigma} \\
&\quad \equiv \mathbf{LP}[\![a(i,j) = b(j,i)]\!]_{\{g_0,\ldots,g_9\}\{g'_0,\ldots,g'_9\}\sigma} = \sigma',
\end{aligned}$$

where $\sigma' = \sigma \lhd \sigma_{k_{0,0}} \ldots \lhd \sigma_{k_{9,9}}$, and $\sigma_{k_{i,j}} = \mathbf{C}[\![a(i,j) = b(j,i)(g_i)]\!]_{g'_j\sigma} = \sigma \lhd (L_{a_{ij}}, b_{ji})$.

4. Implementation

We have implemented a kernel of $\mathcal{D}$ compiler. It consists of a front end that parses the input $\mathcal{D}$ program, and a back end that generates the SPMD target code (which is MPI code currently). The front end parses the input $\mathcal{D}$ program and outputs an intermediate language. Then we define a class of rewrite rule to generate target code. The left hand side specifies a program pattern while the right hand side defines its replacement. We say that a rule matches a program construct if the rule is defined for this program construct. If a rule matches a program construct it will replace this program construct by that of the right hand side.

We aim at achieving an engine that reads a program and a specification of a transformation rule, and will result in a modified program. For example, the implementation of mapping statements in $\mathcal{D}$ are performed by using following transformation rules:

$$\begin{aligned}
&\mathsf{MP}(proc, tmp) \rightarrow \mathsf{MP}(proc, array) \backslash \mathsf{MP}(tmp, array), \\
&\mathsf{MP}(proc, array) \rightarrow \mathsf{distribute}(local - array, proc, index - pair), \\
&\mathsf{MP}(tmp, array) \rightarrow \mathsf{align}(global - array, temp, index - pair).
\end{aligned}$$

The function distribute and align implement to distribute global array onto local processors and align the index between two global arrays.

According to our semantic model, we interpret **forpar** as a normalized parallelizing loop statement — a loop statement without communication in MPI — if there is no data dependence in loop body. The implementation is based on the rule below:

$$\mathsf{FP}(header, body) \rightarrow \mathsf{loop}(header, body)\ if\ not\ \mathsf{DataDepend}(header, body)$$

Here the function DataDepend detects for the given loop body if (flow, anti, and output) data dependences exist.

5. Related work

Several researchers have attempted to give formal semantics to data-parallel languages. Stewart [7] provided a axiomatic semantics and inference rules for three representations of data-parallel array assignment — generalized array assignment, FORTRAN 90 array assignment and HPF array assignment.

Bouge *et. al.* [1,2] defined both axiomatic and denotational semantics for a "low level" data-parallel language $\mathcal{L}$, which gives instructions and control constructs including Skip, Assignment, Communication, Sequencing and iteration. They also define an intermediate data-parallel language where asynchronism becomes explicit in the syntax. Its semantics is based on a twin memory management. Based on these semantic models, they researched some extent work such as implementation of conditioning constructs and formal definitions of structural clocks for a loosely synchronized language, compilation and program equivalence.

Breehaart, Paalvast, and Sips *et.al.* [9] proposed an experimental data parallel language called *Booster*. Some parallel features such as View concept and the separation of algorithm description and machine mapping were introduced. Then they developed an array based calculus named *V*-cal which could be used in translating Booster. A semantic system based on λ-calculus approach was introduced to be able to translate and optimize programs into *V*-cal expressions.

6. Conclusions

A denotational semantics has been presented for a small (kernel) data parallel language. This semantics presented differs from that of a simple sequential language in that it provides meaning for:

(i) some data distribution features, and

(ii) a data parallel statement similar as FORALL in HPF

in place of the standard semantics for simple assignment. The semantics for localized data-parallel statement is order independent, and the semantics for the data distribution directives are defined by using mapping statement to bind an array element to a local processor.

A number of research directions remain to be explored. It should be interesting to extend the semantic explanations of some other directives such as INDEPENDENT and INHERIT directives in HPF. Another direction would be to develop a "low level" (instruction level) data-parallel language with its semantic model, and give the transformation rule from $\mathcal{D}$ to this language, in order to enable compilers writers to explore the effect of heuristics used in parallelizing compilers systematically.

[1] L. Bouge, Y. LeGuyadec, G. Utard, and B. Virot: On the Expressivity of a Weakest Precondition Calculus for a Simple Data-Parallel Programming Language (extended version). Technical report, LIFO: RR94-07, Universite d'Orleans, April 1994.

[2] L Bouge, and P. Garda: Towards a Semantic Approach to SIMD Architectures and Their Languages. *Proc. 18th Spring School of the LITP*, LNCS 469, Springer-Verlag , 1990. pp. 142–175.

[3] S. Hiranandani, K. Kennedy, and C. Tseng: Compiling Fortran D for MIMD Distributed Memory Machines, *Communications of the ACM*, Vol. 35, No. 8(1992), pp. 66–80.

[4] HPF Forum: *High Performance Fortran Language Specification*, Rice University, Houston, Texas, version 2.0 edition, Nov. 1996.

[5] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel: *The High Performance Fortran Handbook*, The MIT Press, 1994.

[6] E. Robinson, Logical Aspects of Denotational Semantics. LNCS 283, 1987. pp. 238–253.

[7] A. Stewart: Reasoning about Data-Parallel Array Assignment, *Journal of Parallel and Distributed Computing*, Vol. 27(1995), pp. 79–85.

[8] Thinking Machines Corporation: *C* Programming Guide.* Technical Report, Cambridge MA, 1990.

[9] J.A. Trescher, L.C. Breebaart, P.F.G. Dechering, A.B. Poelman, J.P.M. de Vreught, and H.J. Sips: A Formal Approach to the Compilation of Data-Parallel Languages. *Proc. 7th Annual Workshop on Languages and Compilers for Parallel Computing*, LNCS 892, Springer-Verlag, 1995. pp. 155–169.

[10] G. Winskel: *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing Series. The MIT Press, 1993.

[11] M. Wolfe: *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1995.