TITLE:

ADINA Computer I and II : II. Data Structure

AUTHOR(S):

NOGI, Tatsuo

ADINA Computer I and II

II. Data Structure

By

Tatsuo Nogi*

(Received March 30, 1981)

Abstract

The ADINA Computer treats data of a characteristic structure which is very successful for parallel computation, and also shows well the structure of the machine itself. The data structure, as shown in this paper, is easily introduced into a high level programming language PASCAL. For illustration, some programs are shown by using the language.

Introduction

The previous papers [1] and [2] proposed new parallel machines, ADINA-I, -II, for high speed computing in science and engineering. The ADINA-1 is especially useful for computer simulation in a one or two dimensional physical space, and the ADINA-II is useful for that in a three dimensional space.

This paper aims to show some characteristics of the data structure of the ADINA Computer, and some commands and actions in parallel computation. It will be found that the data structure is very simple and natural, and that it is easily introduced into a high level programming languge, for example, PASCAL. Hence, it is also found how well the machine itself is structured.

§1 is for the ADINA-I and §2 is for the ADINA-II.

§1 ADINA-I and its data structure

1.1 The main part of the architecture of the ADINA-I is conceptionally shown in Fig. 1. Here, every circle means an arithmetic processor unit (AU) having an equivalent structure and a private memory block. Those processors are numbered from 1 to N, as shown in the figure. In order to always allow a data transfer between any pair of processors, a two dimensional array of buffer memories (BM) is equipped, and each BM is expressed by a square in the figure. This array is a

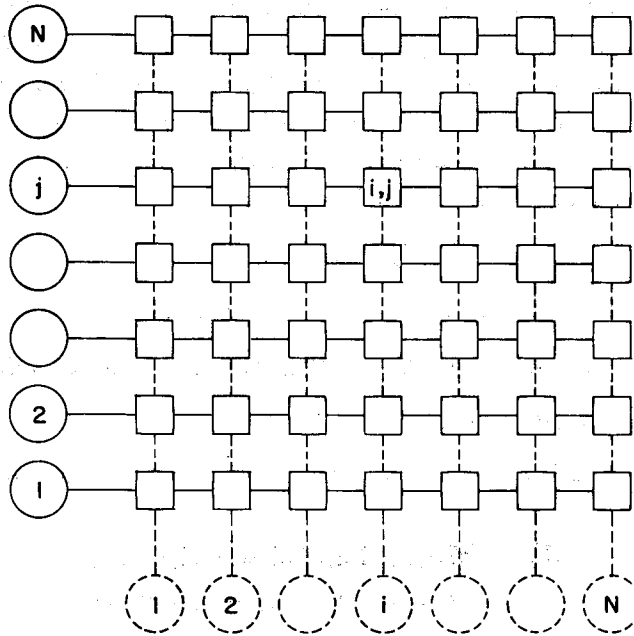* Department of Applied Mathematics and Physics

Fig. 1.   Buffer memories and processors of the ADINA-I.

main characteristic of the ADINA-I.

The set of N processors can take two positions in the way of time sharing. These positions are shown by real line circles and broken line circles, and are called the row position and the column position respectively.   A data transfer from the AU-i to the AU-j, whichever AU-i (alternatively the AU-j) is at the row (column) position or the column (row) position, is mediated by the buffer memory BM-(i,j) or BM-(j,i).   In order that such a data transfer be allowed for every pair of processors, every AU-j must be connected to the buffer memories BM-(i,j) $(i=1, 2, \cdots, N)$ and BM-(j,k) $(k=1, 2, \cdots, N)$ by busses.   In order to reduce the quantity of the hardware as much as possible, and to make each processor take an asynchronous action easily, it is more convenient to use a FIFO (First-In-First-Out) memory as a buffer memory, as stated in [1].

1.2   For a computer simulation of a two dimensional physical phenomenon, an unknown variable u is usually expressed by a set of data on a two dimensional array of grid points, such as

$$\textbf{var } u: \textbf{ array } [1..K, 1..L] \textbf{ of } real; .$$

The value of u itself is expressed as $u[i,j]$.   The letters K and L may, of course, be any integers, but for simplicity, only the case where $K=L=N$ is considered in the

following. As a rule, every processor then plays a role of computing on the corresponding row and column of grid points. For example, the AU-j is concerned to determine a one dimensional array of $u[i,j]$ $(i=1, 2, \cdots, N)$ or $u[k,j]$ $(k=1, 2, \cdots, N)$ (at the row or column position, respectively). In order to clear the distinction between the two kinds of one dimesnional arrays, we call one kind a data row and the other kind a data column, denoted by

$$u[i, (j)] \quad \text{and} \quad u[(i), k]$$

respectively. The idea is to deal with data declared like

var u: **array** $[1 .. N, (1 .. N)], [(1 .. N), 1 .. N]$ **of** *real*;.

This gives a typical data structure of the ADINA-I.

1.3 For an illustration, we first consider the following example of an iterative method:

$$(1) \qquad u_{i,j}^{n+1} = (u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n})/4 ,$$
$$n = 0, 1, 2, \cdots, \quad i, j = 1, 2, \cdots, N ,$$

where n is the number of iterations, and i and j are the numbers of the grid coordinates. To realize a one time computation from n to $n+1$, we take the following procedure:

   (i) $v_{i,j} = u_{i,j+1} + u_{i,j-1}$ $(i, j = 1, 2, \cdots, N)$,

   (ii) $v_{i,j} = (u_{i+1,j} + u_{i-1,j} + v_{i,j})/4$ $(i, j = 1, 2, \cdots, N)$,

   (iii) $u_{i,j} = v_{i,j}$ $(i, j = 1, 2, \cdots, N)$.

It is noticed in the first step that the right hand side of (i) contains only the variables with the same first suffix i and the different values of the second suffix $j+1$ and $j-1$. This allows for the computation (i) to proceed in parallel and independently for all i's. It is also noticed in the second step that the right hand side of (ii) contains only the variables having the same second suffix j and the different values of the first suffix $i+1$, i, $i-1$. It allows for the computation (ii) to be done in parallel and independently for all j's. For formulating such a procedure, the array type with the double bracket is very convenient, as seen in the following program. (It is supposed that at the starting point $u[i, (j)]$ $(i, j = 1, 2, \cdots, N)$ are known.)

procedure *one setp of the Jacobi's iteration method for the Laplace operator*;
const $N=16$;
var u, v: **array** $[1 .. N, (1 .. N)], [(1 .. N), 1 .. N]$ **of** *real*;
   $i, j: 1 .. N$;

begin

  for $i, j =: 1$ **to** N **ptransfer** $u[(i), j]: = u[i, (j)]$;

    {*For the parallel transfer to reread the data row as the the data column, and for the detail of the action of* **ptransfer**, *see the next procedure.*}

    for $i: = 1$ **to** N **pdo**

    begin

      for $j: = 1$ **to** N **do** $v[(i), j]: = u[(i), j+1] + u[(i), j-1]$

    end;

    {*parallel doing the computation enclosed by* **begin** *and* **end** *over all* i*'s*}

  for $i, j: = 1$ *to* N **ptransfer** $v[i, (j)]: = v[(i), j]$;

    {*to reread the data column as the data row*}

  for $j: = 1$ **to** N **pdo**

    begin

      for $i: = 1$ **to** N **do** $v[i, (j)]: = (u[i+1, (j)] + u[i-1, (j)] + v[i, (j)])/4$

    end;

  for $j: = 1$ **to** N **pdo**

    begin

      for $i: = 1$ **to** N **do** $u[i, (j)]: = v[i, (j)]$

    end

end;.

In the above procedure, the parallel command **ptransfer** and **pdo** are introduced. Their means and the methods of their application are easily read from the above procedure. Only one point to notice here is that all variables in a loop of **pdo** must be data of the same type, row or column.

1.4 In order to **ptransfer**, an array of buffer memories is essential. As mentioned above, we use a FIFO memory as each buffer, and here introduce the **type fifo** for its expression. That is,

  type $S =$ **fifo of** real;

  var *buffer*: **array** $[1 .. N, (1 .. N)] = [(1 .. N), 1 .. N]$ **of** S; .

The **type fifo** is the same data structure as that of the **type file**, but we have introduced the former in order to make a distinction from the latter used in the usual way. Now, we use both expressions

$$\textit{buffer } [i, (j)] \quad \text{and} \quad \textit{buffer } [(i), j]$$

for the same buffer. The sign $=$ in the definition of **var** *buffer* means the idendity as the hardware. The double brackets are, however, introduced to distinguish between the i-th element of the j-th row of buffer $[i, (j)]$ and the j-th element of

the i-th column of *buffer* $[(i),j]$.

The detail of **ptransfer** is as follows:

```
procedure ptransfer from row to column;
const N=16;
type S= fifo of real;
var buffer array [1..N, (1..N)]=[(1..N), 1..N] of S;
    u: array [1..N, (1..N)], [(1..N), 1..N] of real;
    i,j: 1..N;
begin
  for j: =1 to N pdo
    begin for i: =1 to N do
          begin rewrite (buffer [i, (j)]); buffer [i, (j)] ↑ : =u[i, (j)];
                put(buffer [i, (j)])
          end
    end;
  for i: =1 to N pdo
    begin for j: =1 to N do
          begin reset (buffer [(i),j]); u[(i),j]:=buffer [(i),j] ↑ ;
                get(buffer [(i),j])
          end
    end
end;
```

§2 ADINA-II and its data structure

2.1 The ADINA-II has a two dimensional array of N^2 arithmetic processor units and N two dimensional arrays of N^2 buffer memories. Fig. 2 shows the way of the bus connection between the k-th array of buffer memories (figured by the squares) and the related processors (figured by the circles) $(k=1, 2, \cdots, N)$. The array of all processors is made to take two positions, as in the ADINA-I. They are again shown partly by the real line circles and the broken line circles, and are again called the row position and the column position respectively.

The way of data transfer through the k-th array of buffer memories is the same as that in the ADINA-I, but differs only in the point that one side of data transfer is the k-th row of processors, and its counterpart is the k-th column of processors. (In the ADINA-I, both sides are the same one dimensional array.) In this case, it is not always allowed to transfer data directly between any pair of processors, but only between every pair, both parts of which are in a row and a
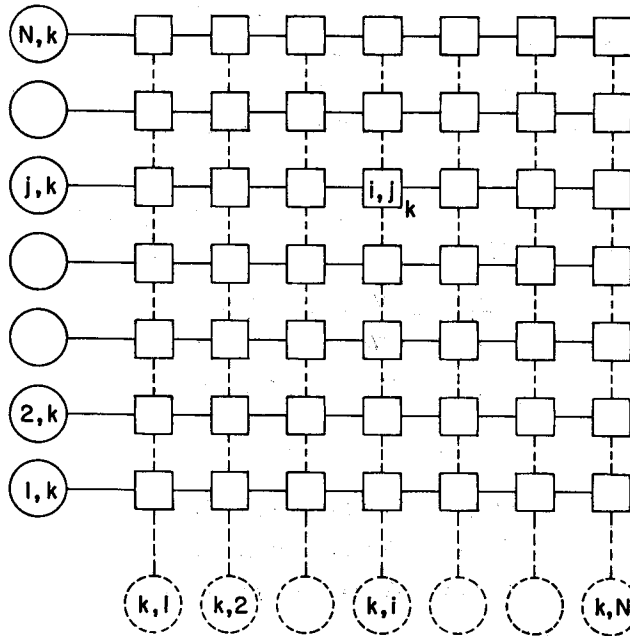
Fig. 2. The k-th array of buffer memories and related processors of the ADINA-II.

column with the same order number respectively. It is, however, easily seen that every data transfer is always allowed indirectly through only one processor as a mediator, such as

$$((j, k)) \rightarrow (l, j)_k \rightarrow ((k, l)) \rightarrow (m, k)_l \rightarrow ((l, m)) ,$$

where $((k, l))$ is the mediator for transferring from $((j, k))$ to $((l, m))$, and $((,))$ is the number of processors and $(,)_k$ is the number of buffer memories on the k-th array. (See [2] for details.) Such a way of data transfer is a main characteristic of the ADINA-II.

2.2 The other characteristic point for use is that the complex of all the processors and buffer memories may take three positions with regard to a fixed cube of lattice points for computation. These positions are called the a-position, the b-position and the c-position respectively. (See [2], especially Fig. 4-a, -b, -c.) Corresponding to such a situation, we here introduce three kinds of one dimensional array instead of the usual three dimensional array, for example,

$$u[i, (j, k)] , \quad u[i), j, (k] \quad \text{and} \quad u[(i, j), k]$$

instead of $u[i, j, k]$. Here, it is assumed that the first variable belongs to the (j, k)-

th processor, the second to the (k, i)-th and the third to the (i, j)-th when these processors are at the row positions of the a-, b- and c-position of the complex respectively. We call such arrays the a-array, the b-array and the c-array respectively, which assume a main aspect of the data structure.

2.3 Such data structure may be well understood by reading the following application program, which solves the Poisson equation $\Delta u = f$ in a three dimensional cube by the well-known ADI method of Douglas and Rachford:

(1) $\qquad (-\Delta_1 + r)u^{n+1/3} = (\Delta_2 + \Delta_3 + r)u^n$,

(2) $\qquad (-\Delta_2 + r)u^{n+2/3} = -\Delta_2 u^n + r u^{n+1/3}$,

(3) $\qquad (-\Delta_3 + r)u^{n+1} = -\Delta_3 u^n + r u^{n+2/3}$

where r is the parameter of iteration, n is the number of iteration and

$$\Delta_1 u_{i,j,k} = u_{i+1,j,k} - 2u_{i,j,k} + u_{i-1,j,k} ,$$
$$\Delta_2 u_{i,j,k} = u_{i,j+1,k} - 2u_{i,j,k} + u_{i,j-1,k} ,$$
$$\Delta_3 u_{i,j,k} = u_{i,k,k+1} - 2u_{i,j,k} + u_{i,j,k-1} .$$

It is supposed in the following program that the values of u at the n-th stage and the boundary are given, and that the **procedure** *double sweep* is defined beforehand. The procedure solves tri-diagonal systems arising from the equations (1), (2), (3).

procedure *one step of ADI method*;
const $N = 16$;
var i, j, k: $1..N$;
  f: **array**$[1..N, (1..N, 1..N)]$ **of** *real*;
  $u, \Delta_2 u$: **array**$[1..N, (1..N, 1..N)], [1..N), 1..N, (1..N], [(1..N, 1..N),$
        $1..N]$ **of** *real*;
  $\Delta_3 u$: **array**$[1..N, (1..N, 1..N)], [(1..N, 1..N), 1..N]$ **of** *real*;
  {*The symbol Δ_2 or Δ_3 is, of course, not allowed in PASCAL, but we here use it for simplicity of illustration. Other convenient symbols will also appear in later program without notice.*}
begin
  for j, k: $=1$ **to** N **pdo** *double sweep*$(u[\cdot, (j, k)])$;
  {*to solve the equation (1) for every (j,k) in parallel at the a-position and get the a-array of the solution $u[\cdot, (j, k)]$*}
  for i, j, k: $=1$ **to** N **ptransfer**
    begin $u[i), j, (k]$: $=u[i, (j, k)], \Delta_2 u[i), j, (k]$: $=\Delta_2 u[i, (j, k)]$
    end; {*to reread the a-arrays as the b-arrays*}
  for k, i: $=1$ **to** N **pdo** *double sweep* $(u[i), \cdot, (k])$;

{*to solve the equation (2) for every (k,i) in parallel at the b-position and get the b-array of the solution u[i], ·, (k)*}

for $i, j, k: =1$ **to** N **ptransfer** $u[(i,j), k]: =u[i), j, (k]$;

{*to reread the b-array as the c-array*}

for $i, j: =1$ **to** N **pdo** *double sweep* $(u[(i,j), ·])$;

{*to solve the equation (3) for every (i,j) in parallel at the c-position and get the c-array of the solution u[(i,j), ·]*}

for $i, j, k: =1$ **to** N **ptransfer** $u[i), j, (k]: =u[(i,j), k]$

{*to reread the c-array as the b-array*}

for $k, i: =1$ **to** N **pdo**

  begin for $j: =1$ **to** N **do**

    $\Delta_2 u[i), j, (k]: =u[i), j+1, (k] -2*u[i), j, (k] +u[i), j-1, (k]$

  end;

for $i, j, k: =1$ **to** N **ptransfer** $\Delta_2 u[(i,j), k]: =\Delta_2 u[i), j, (k]$;

for $i, j: =1$ **to** N **pdo**

  begin for $k: =1$ **to** N **do**

    $\Delta_3 u[(i,j), k]: =u[(i,j), k+1] -2*u[(i,j), k] +u[(i,j), k-1]$

  end;

for $i, j, k: =1$ **to** N **ptransfer**

  begin $u[i, (j, k)]: =u[(i,j), k]$, $\Delta_2 u[i, (j, k)]: =\Delta_2 u[(i,j), k]$, $\Delta_3 u[i, (j, k)]: =$
    $\Delta_3 u[(i,j), k]$

  end

end;

It is seen in this program that the main parts of the computation generally proceed in a fully parallel way, and hence the efficiency of parallelism is almost 100%. It is found that the ADINA Computers are very well suited for the ADI methods, the Fractional Step methods etc..

2.4 Now, we comment about the procedure **ptransfer** in the ADINA-II. When the buffer memories are RAM's as in [2], the procedure is simple. In fact, it is only a change of view point, from the row (column) position to the column (row) position of the processors, for example,

  "**for** $i, j, k: = 1$ **to** N **ptransfer** $u[i), j, (k]: = u[i, (j, k)]$"

means to consider the a-array stored from the row position at the a-position as the b-array by viewing from the column position and also the a-array at the b-position simultaneously. Thus there is no time loss with regard to the **ptransfer**.

On the other hand, where the buffer memories are the FIFO's, there is some

time loss for the **ptransfer.** We have, then, a special data structure of buffer memories to notice here, for example,

> **type** S=**fifo of** *real*;
> **var** *buffer*: **array** $[1..N, (1..N, (1..N))] = [1..N), 1..N, ((1..N)],$
> $[(1..N)), 1..N, (1..N] = [((1..N), 1..N), 1..N],$
> $[(1..N, (1..N)), 1..N] = [1..N, ((1..N), 1..N)]$
> **of** S;.

Then, we use both expressions

$$buffer[i, (j, (k))] \quad \text{and} \quad buffer[i), j, ((k)]$$

for the (i, j)-th buffer of the k-th array as the complex of processors and buffer memories is at the a-position. The former is used as the i-th buffer viewed from the j-th processor on the k-th row, the array of processors being at the row position. The latter is used as the j-th buffer viewed from the i-th processor on the k-th column, the array of processors being at the column position. The mentioned processors are really connected to the k-th array of buffer memories. Similarly, both

$$buffer[(i)), j, (k] \quad \text{and} \quad buffer[((i), j), k]$$

are expressions for the (j, k)-th buffer of the i-th array as the complex is at the b-position. The former is the j-th buffer viewed from the k-th processor on the i-th row, the array of processors being at the row position. The latter is the k-th buffer viewed from the j-th processor on the i-th column, the array of processors being at the column position. Also, both

$$buffer[(i, (j)), k] \quad \text{and} \quad buffer[i, ((j), k)]$$

are expressions for the (k, i)-th buffer of the j-th array as the complex is at the c-position. The former is the k-th buffer viewed from the i-th processor on the j-th row, the array of processors being at the row position. The latter is the i-th buffer viewed from the k-th processor on the j-th column, the array of processors being at the column position.

As seen above, the correspondence between the pattern of the double brackets and the real positions of the complex and the array of processors is ruled as follows:

The inner bracket ( ) being at		The complex being at
the third position	$\leftrightarrow$	the a-position
the first position	$\leftrightarrow$	the b-position
the second position	$\leftrightarrow$	the c-position

In the outer bracket (,), the		The array of processors
inner bracket being at		being at
the latter	$\leftrightarrow$	the row position
the former	$\leftrightarrow$	the column position

We here omit the detail of **ptransfer** by using such FIFO buffers, since it is the same as that in the ADINA-I mentioned in §1. The following program however, gives an example of their usage.

2.5 It is an electrostatic particle code of the plasma simulation, which shows that the ADINA Computer is very useful not only for fluid simulation but also for particle simulation. In the particle code, it is characteristic to have a procedure of 'particle push', which moves every particle to each new position. The procedure is very complicated for other parallel machines, but it is comparatevely easy for the ADINA Computer, as seen in the following program.

The electrostatic model is as follows:

$$\rho(\boldsymbol{r}) = \sum_{\alpha} q_{\alpha} S(\boldsymbol{r}-\boldsymbol{r}_{\alpha}) ,$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{E}(\boldsymbol{r}) = 4\pi\rho(\boldsymbol{r}) ,$$

$$\boldsymbol{F}(\boldsymbol{r}_{\alpha}) = \int S(\boldsymbol{r}'-\boldsymbol{r}_{\alpha})\boldsymbol{E}(\boldsymbol{r}')d\boldsymbol{r}' ,$$

$$m_{\alpha} \frac{dv_{\alpha}}{dt} = q_{\alpha}(\boldsymbol{F}(\boldsymbol{r}_{\alpha})+\frac{1}{c} v_{\alpha} \times \boldsymbol{B}_0) ,$$

where $\boldsymbol{r}$ is the three dimensional Euclidian vector, $\boldsymbol{r}_{\alpha}$, v_{α}, q_{α} and m_{α} are the position, the velocity, the charge and the mass of the α-particle, respectively. $S(\boldsymbol{r})$ is the shape factor of the super particle with a finite size, $\boldsymbol{E}(\boldsymbol{r})$ the electric field, $\boldsymbol{F}(\boldsymbol{r}_{\alpha})$ the force by the electric field, $\boldsymbol{B}_0$ the given static magnetic field and c the light velocity.

It is assumed that the computation is realized on a cubic lattice, with the accuracy of the first order as regards the width of the lattice points (grid points). For that, the following approximate methods are applied. The first equation is approximated by the formula

$$\rho(\boldsymbol{r}) = \sum_{g} [S(\boldsymbol{r}-\boldsymbol{r}_g)Q(\boldsymbol{r}_g)+\boldsymbol{D}(\boldsymbol{r}_g)\boldsymbol{\nabla}_g \cdot S(\boldsymbol{r}-\boldsymbol{r}_g)]$$

where $\boldsymbol{r}_g$ is the nearest grid point of $\boldsymbol{r}_{\alpha}$, $Q(\boldsymbol{r}_g)$ and $\boldsymbol{D}(\boldsymbol{r}_g)$ are the monopole and the dipole at the grid point $\boldsymbol{r}_g$ respectively, and are given by

$$Q(\boldsymbol{r}_g) = \sum_{\alpha \in g} q_{\alpha}, \quad \boldsymbol{D}(\boldsymbol{r}_g) = \sum_{\alpha \in g} q_{\alpha}\varDelta\boldsymbol{r}_{\alpha} \quad (\varDelta\boldsymbol{r}_{\alpha} = \boldsymbol{r}_{\alpha}-\boldsymbol{r}_g) .$$

In order to get force $\boldsymbol{F}$ from the monopole and the dipole, the method of Fast Fourier Transform (FFT) is very useful. It is, in fact, as follows:

i) to get the discrete Fourier Transform $\tilde{Q}(\boldsymbol{k})$ and $\tilde{\boldsymbol{D}}(\boldsymbol{k})$ by the FFT method,

ii) to get $\tilde{F}(k)$ from the equations

$$\tilde{\rho}(k) = \tilde{S}(k)[\tilde{Q}(k) - \sqrt{-1}k \cdot \tilde{D}(k)] \,,$$
$$\tilde{E}(k) = -\sqrt{-1}4\pi k\tilde{\rho}(k)/|k|^2 \,,$$
$$\tilde{F}(k) = \tilde{S}(-k)\tilde{E}(k) \,,$$

iii) to get $F(r_g)$ from $F(k)$ (FFT^{-1}).

The next step is to determine the destination of every particle by solving the difference equations

$$F^{n-1/2}(r_a) = F(r_g) + \Delta r_a^{n-1/2} \nabla_g \cdot F(r_g) \,,$$
$$\frac{v_a^n - v_a^{n-1}}{\Delta t} = \frac{q_a}{m_a}\left(F^{n-1/2}(r_a) + \frac{v_a^n + v_a^{n-1}}{2c} \times B_0 \right) \,,$$
$$r_a^{n+1/2} = r_a^{n-1/2} + \Delta t v_a^n \,.$$

The last step is the 'particle push'.

```
procedure one step of the electrostatic code;
const N = 16;
type vector = array[1..3] of real; complex = record real part, imaginary part: real end;
    complex vector = record real part, imaginary part: vector end;
    T = file of record destination: record first, second, third: 1..N end;
                        r_a, Δ_a r, v_a: vector;
                        m_a, q_a: real
            end;
var particle: array[1..N, (1..N, 1..N)], [1..N), 1..N, (1..N] of T;
    i, j, k: 1..N;
procedure Q and D;
begin
  for j, k: = 1 to N pdo
    begin for i: = 1 to N do
          begin Q[i, (j, k)]: = 0; D[i, (j, k)]: = 0;
                reset particle [i, (j, k)];
                while not eof particle [i, (j, k)]) do
                    begin Q[i, (j, k)] := Q[i, (j, k)] + particle[i, (j, k)].q_a;
                        D[i, (j, k)] := D[i, (j, k)] + particle[i, (j, k)].q_a * Δr_a
                    end
            end
        end
    end
end;
procedure Q̃ and D̃
```

var $\tilde{Q}_1$:   **array**$[1 \ldots N, (1 \ldots N, 1 \ldots N)], [1 \ldots N), 1 \ldots N, (1 \ldots N]$ **of** *complex;*

    $\tilde{D}_1$:   **array**$[1 \ldots N, (1 \ldots N, 1 \ldots N)], [1 \ldots N), 1 \ldots N, (1 \ldots N]$ **of** *complexe vector;*

    $\tilde{Q}_2$:   **array**$[1 \ldots N), 1 \ldots N, (1 \ldots N], [(1 \ldots N, 1 \ldots N), 1 \ldots N]$ **of** *complex;*

    $\tilde{D}_2$:   **array**$[1 \ldots N), 1 \ldots N, (1 \ldots N], [(1 \ldots N, 1 \ldots N), 1 \ldots N]$ **of** *complex vector;*

    $\tilde{Q}_3$:   **array**$[(1 \ldots N, 1 \ldots N), 1 \ldots N], [1 \ldots N, (1 \ldots N, 1 \ldots N)]$ **of** *complex;*

    $\tilde{D}_3$:   **array**$[(1 \ldots N, 1 \ldots N), 1 \ldots N], [1 \ldots N, (1 \ldots N, 1 \ldots N)]$ **of** *complex vector;*

begin for $j, k := 1$ **to** N **pdo** $FFT(\tilde{Q}_1[\cdot, (j, k)], \tilde{D}_1[\cdot, (j, k)]);$

    {*to take the one dimensional discrete Fourier Transformation in the direction of the first coordinate, the procedure FFT being assumed to be defined beforehand*}

  for $i, j, k := 1$ **to** N **ptransfer**

    begin $\tilde{Q}_1[i), j, (k] := \tilde{Q}_1[i, (j, k)], \tilde{D}_1[i), j, (k] := \tilde{D}_1[i, (j, k)]$

    end

end;

begin for $k, i := 1$ **to pdo** $FFT(\tilde{Q}_2[i), \cdot, (k], \tilde{D}_2[i), \cdot, (k]);$

  for $i, j, k := 1$ **to** N **ptransfer**

    begin $\tilde{Q}_2[(i, j), k] := \tilde{Q}_2[i), j, (k], \tilde{D}_2[(i, j), k] := \tilde{D}_2[i), j, (k]$

    end

end;

begin for $i, j := 1$ **to** N **pdo** $FFT(\tilde{Q}_3[(i, j), \cdot], \tilde{D}_3[(i, j), \cdot]);$

  for $i, j, k := 1$ **to** N **ptransfer**

    begin $\tilde{Q}_3[i, (j, k)] := \tilde{Q}_3[(i, j), k], \tilde{D}_3[i, (j, k)] := \tilde{D}_3[(i, j), k]$

    end

end; {**procedure** $\tilde{Q}$ and $\tilde{D}$}

procedure $\tilde{F}$;

var $\tilde{\rho}, \tilde{S}, \tilde{S}^-$: **array**$[1 \ldots N, (1 \ldots N, 1 \ldots N)]$ **of** *complex;*

    {*It is supposed that $\tilde{S}(k)$ and $\tilde{S}^-(k) = \tilde{S}(-k)$ are defined beforehand*}

    $\tilde{E}$: **array**$[1 \ldots N, (1 \ldots N, 1 \ldots N)]$ **of** *complex vector;*

    $\tilde{F}$: **array**$[1 \ldots N, (1 \ldots N, 1 \ldots N)], [1 \ldots N), 1 \ldots N, (1 \ldots N]$ **of** *complex vector;*

begin

  for $j, k := 1$ **to** N **pdo**

    begin for $i := 1$ **to** N **do**

        begin $\tilde{\rho}[i, (j, k)] := \tilde{S}[j, (j, k)] * (\tilde{Q}_3[i, (j, k)] - \sqrt{-1} k \cdot \tilde{D}[i, (j, k)]);$

            $\tilde{E}[i, (i, k)] := -4\pi\sqrt{-1} k * \rho[i, (j, k)]/|k|^2;$

            $\tilde{F}[i, (j, k)] := \tilde{S}^-[i, (j, k)] * \tilde{E}[i, (j, k)]$

      end

    end

end;

procedure F *and* $\nabla \cdot F$;

const $h = 1/N$;

var F: **array**$[(1 \ldots N, 1 \ldots N), 1 \ldots N], [1 \ldots N, (1 \ldots N, 1 \ldots N)], [1 \ldots N), 1 \ldots N, (1 \ldots N]$
       of *vector*;

  F_x: **array**$[1 \ldots N, (1 \ldots N, 1 \ldots N)]$ **of** *vector*;

  F_y: **array**$[1 \ldots N, (1 \ldots N, 1 \ldots N)], [1 \ldots N), 1 \ldots N, (1 \ldots N]$ **of** *vector*;

  F_z: **array**$[1 \ldots N, (1 \ldots N, 1 \ldots N)], [(1 \ldots N, 1 \ldots N), 1 \ldots N]$ **of** *vector*;

begin FFT^{-1} $(F[(i,j), k])$

    $\{$*It is hree assumed that the procedure* FFT^{-1} *is defined beforehand.   The procedure*
    is very similar to **procedure** $\tilde{Q}$ *and* $\tilde{D}\cdot\}$.

end;

begin

  for $i, j := 1$ **to** N **pdo**

    begin for $k := 1$ **to** N **do** $F_z[(i,j), k] := (F[(i,j), k+1] - F[(i,j), k-1])/2h$

    end;

  for $i, j, k := 1$ **to** N **ptransfer**

    begin $F[i, (j, k)] := F[(i,j), k]$, $F_z[i, (j, k)] := F_z[(i,j), k]$

    end;

  for $j, k := 1$ **to** N **pdo**

    begin for $i := 1$ **to** N **do** $F_x[i, (j, k)] := (F[i+1, (j, k)] - F[i-1, (j, k)])/2h$

    end;

  for $i, j, k := 1$ **to** N **ptransfer** $F[i), j, (k] := F[i, (j, k)]$;

  for $k, i := 1$ **to** N **pdo**

    begin for $j := 1$ **to** N **do** $F_x[i), j, (k] := (F[i), j+1, (k] - F[i), j-1, (k])/2h$

    end;

  for $i, j, k := 1$ **to** N **ptransfer** $F_y[i, (j, k)] := F_y[i), j, (k]$

end;

procedure *particle destination*;

begin

  for $j, k := 1$ **to** N **pdo**

    begin for $i := 1$ **to** N **do**

        begin *reset* $(particle[i, (j, k)])$;

          while not *eof* $(particle\ [i, (j, k)])$ **do**

           begin

            $\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$

            $particle[i, (j, k)] \cdot v_a \uparrow := \cdots\cdots\cdots\cdots\cdots$;

$$particle[i, (j, k)].\ r_\alpha \uparrow := particle[i, (j, k)].r_\alpha \uparrow +$$
$$\Delta t * particle[i, (i, k)].r_\alpha \uparrow \ ;$$

$$particle[i, (j,k)].destination \uparrow := \cdots\cdots\cdots;$$
$$particle[i(j,k)].\Delta r_\alpha \uparrow := \cdots\cdots\cdots;$$
$$put(particle[i, (j, k)])$$

                              end

                       end

              end

end;

procedure *particle push*;

type $S=$**fifo of** *real*

var *buffer*:   **array**$[1..N, (1..N, (1..N))] = [1..N), 1..N, ((1..N)],$
                       $[(1..N)), 1..N, (1..N] = [((1..N), 1..N), 1..N]$ **of** S;

  l, m, n: $1..N$;

  {*When a particle contained in the particle*$[l, (j, k)]$ *is carried into the particle* $[n, (l, m)]$,
  with **for** $j, k := 1$ **to** N **pdo**', *it is once transferred into the*
                    $buffer[l, (j, (k))] = buffer[l), j, ((k)]$,
  with '**for** $k, l := 1$ **to** N **pdo**', *it is then transferred into the*
                    $buffer[(l)), m, (k] = buffer[((l), m), k]$,
  and with '**for** $l, m := 1$ **to** N **pdo**', *it is written into the particle*$[n, (l, m)]$.}

begin

  for $j, k := 1$ **to** N **pdo**
    begin for $i := 1$ **to** N **do** *rewrite* $(buffer[i, (j, (k))])$
    end;

  for $j, k := 1$ **to** N **pdo**
    begin for $i := 1$ **to** N **do**
        begin *reset* $(particle[i, (j, k)])$;
          while not *eof* $(particle[i, (j, k)])$ **do**
            begin $l := particle[i, (j, k)].destination.second \uparrow$;
               $buffer[l, (j, k))] \uparrow := particle[i, (j, k)] \uparrow$;
               put $(buffer[l, (j, (k))])$; *get* $(particle[i, (j, k)])$
            end
        end
    end;

  for $k, l := 1$ **to** N **pdo**
    begin for $j := 1$ **to** N **do** *rewrite* $(particle[l), j, (k])$;

```
for j:=1 to N do
    begin reset (buffer[l], j, ((k)]) ;
        while not eof (buffer[l], j, ((k)]) do
            begin particle[l], j, (k) ↑ :=buffer[l], j, ((k)] ↑ ;
                put (particle[l], j ,(k)); get (buffer[l],  j,((k)])
            end
        end
    end
for k, l:=1 to N pdo
    begin for m:=1 to N do rewrite (buffer[(l)), m, (k)]
    end;
for k, l:=1 to N pdo
    begin for j:=1 to N do
        begin reset (particle[l], j,( k)) ;
            while not eof (particle[l], j, (k)) do
                begin m:=particle[l], j, (k).destination. third ↑ ;
                    buffer[(l)), m, (k) ↑ :=particle[l], j, (k) ↑ ;
                    put (buffer[(l)), m, (k)); get (particle[l], j, (k))
                end
            end
        end;
for l, m:=1 to N pdo
    begin for n:=1 to N do rewrite (particle[n, (l, m)]);
        for k:=1 to N do
            begin reset (buffer[((l), m), k]);
                while not eof (buffer[((l), m), k]) do
                    begin n:=buffer[((l), m), k].destination.first ↑ ;
                        particle[n, (l, m)] ↑ :=buffer[((l), m), k] ↑ ;
                        put (particle[n, (l, m)]); get (buffer[((l), m), k])
                    end
                end
            end
    end; {particle push}
begin
    Q and D; Q̃ and D̃; F̃; F and ∇·F; particle destination; particle push
end; {one step of the electrostatic code}
```

2.6 The final example is to show how to realize the command **ptransfer** of the

ADINA-I by the ADINA-II.   It is for the simple data transfer:

procedure *ptransfer of the ADINA-I*
const $N^2=256$;
var u: **array**$[1 .. N^2, (1 .. N^2)], [(1 .. N^2), 1 .. N^2]$ **of** *real*;
   p,q: $1 .. N^2$;
begin for $p, q := 1$ **to** N^2 **ptransfer** $u[(p), q] := u[p,(q)]$
end;

It is convenient to assign four dimensional arrays for the two dimensional arrays of u, namely

$$u[p, (q)] \leftrightarrow v[p \ mod \ N, (q \ mod \ N, q \ div \ N)] \ [p \ div \ N],$$
$$u[(p),q] \leftrightarrow v[(p \ mod \ N, q \ mod \ N), p \ div \ N] \ [q \ div \ N].$$

Then, the last procedure is equivalent to

procedure *realization by the ADINA-II*;
const $N=16$;
var v: **array**$[1 .. N, (1 .. N, 1 .. N)][1 .. N], [(1 .. N, 1 .. N), 1 .. N][1 .. N]$ **of** *real*;
  w: **array**$[(1 .. N, 1 .. N), 1 .. N][1 .. N]$ **of** *real*;
  i, j, k, m:$1 .. N$;
begin
  for $i, j, k, m := 1$ **to** N **ptransfer** $v[(i, j), k][m] := v[i, (j, k)][m]$;
  for $i, j := 1$ **to** N **pdo**
    begin for $k := 1$ **to** N **do**
        begin for $m := 1$ **to** N **do** $w[(i, j), k][m] := v[(i, j), k][m]$
        end;
      for $m := 1$ **to** N **do**
        begin for $k := 1$ **to** N **do** $v[(i, j), m][k] := w[(i, j), k][m]$
        end
    end
end;

Conclusion

We have shown the data structure of the ADINA Computer with some examples of its program for illustration.   The characteristic feature has been seen in the expression of double brackets of variable array.   It is not as cumbersome as it looks because it reflects the architecture of the machine so sufficiently that it is very easy to compile the sources with a very high efficiency.   Also, it covers the details of hardware so well that the user may not know its realization.

Thus, we can say that the data structure is very natural and useful for parallel

computation, and for that reasons we can say that the ADINA Comupter itself is very valuable.

References

1) Nogi, T. and Kubo, M.: ADINA Computer I, I. Architecture and Theoretical estimates, Memoirs of the Faculty of Engineering, Kyoto University, Vol. XLII, Part 4, Oct. 1980.
2) Nogi, T.: ADINA Computer II, I. Architecture and Theoretical estimates, ibid., Vol. XLIII, Part 1, Apr. 1981.
3) Jensen, K. and Wirth, N.: PASCAL User Manual and Report, Lecture Notes in Computer Scinece 18, Springer-Verlag, 1974.