APPLICATIONS OF PARALLEL COMPUTATION IN MICRO-MECHANICS AND FINITE ELEMENT METHOD

Final Report to be submitted to NASA Lewis Center

by

Hui-Qian Tan
Department of Mathematical Sciences
The University of Akron
Akron, Ohio 44325

Time Period: **October 1, 1993 to May 31, 1996**

APPLICATIONS OF PARALLEL COMPUTATION IN MICRO-MECHANICS AND FINITE ELEMENT METHOD

Hui-Qian Tan

Department of Mathematical Science
The University of Akron
Akron, Ohio 44325

1. INTRODUCTION

Numerical solutions of large-scale engineering or scientific problems require extensive computations on computers with conventional architectures. Even with today's fast-advancing computer technology, it appears that the speed of sequential machines is gradually approaching its limit. Perhaps parallel computation is the ultimate solution for the next generation computers. One unique feature of these machines is that they do not rely on exceptionally fast processors, but on a large number of relatively simple processing units. Thus, the increase in speed is achieved by increasing the number of processors in conjunction with appropriate parallel algorithms. Obviously, the speed of a parallel computer is achieved by performing multitask operations in a collective form, rather than sequentially.

Problems of obvious interest for parallel processing because of their computational intensity include Matrix Computation, Computer Vision, Image Processing, Finite Element Analyses, Signal Processing, Simulation, Optimization, etc. As an example, Finite Element method widely is used in material analyses in civil and mechanical engineering. Usually the computation is not trivial due to the complexity and large number of elements.

Parallel processors fall mainly into two general classes. One class is of Single Instruction Multiple Data(SIMD) architecture. Another is of Multiple Instruction Multiple Data architecture. SIMD has many elementary processors(>1000). It is data level parallelism, that is, assigning a processor to a data unit. Typically its memory is distributed and a potential problem is communication speed limit. On the other hand, MIMD has a few powerful processors(<1000). It is control level parallelism, that is, assigning a processor to a unit of code. Typically its memory is shared and a potential problem is memory speed limit. In summary, all processors of SIMD are given the same instruction and each processor operates on different data. On the other hand,

each processor of MIMD runs its own instruction sequence and each processor works on a different part of problem.

The choice between SIMD and MIMD depends on the application[2]. For well-structured problems with regular patterns of control, SIMD machines have the edge, because more of the hardware is devoted to operations on the data. For the applications in which the control flow required of each processing element is complex and data dependent, MIMD architecture has the advantage.

This project is for application of parallel computations related with respect to material analyses. Briefly speaking, we analyze some kind of material by elements computations. We call an element a cell here. A cell is divided into a number of subelements called subcells and all subcells in a cell have the identical structure. The detailed structure will be given later in this paper. It is obvious that the problem is "well-structured". SIMD machine would be a better choice. In this paper we try to look into the potentials of SIMD machine in dealing with finite element computation by developing appropriate algorithms on MasPar, a SIMD parallel machine. In section 2, the architecture of MasPar will be discussed. A brief review of the parallel programming language MPL also is given in that section. In section 3, some general parallel algorithms which might be useful to the project will be proposed. And, combining with the algorithms, some features of MPL will be discussed in more detail. In section 4, the computational structure of cell/subcell model will be given. The idea of designing the parallel algorithm for the model will be demonstrated. Finally in section 5, a summary will be given.

2. MASPAR MACHINE ARCHITECTURE AND MASPAR PROGRAMMING LANGUAGE (MPL)

MasPar parallel processing system is of the SIMD architecture. It consists of a front end(FE) and a Data Parallel Unit(DPU).

The front end includes a workstation that runs an implementation of the UNIX operating system and standard I/O. At present, the front end is a DECstation 5000 with Ultrix operating system.

The DPU consists of an Array Control Unit(ACU), an array of Processor Elements(PEs), and PE communication mechanisms. Figure 2-1 is the system diagram.

The ACU is a processor with its own registers and data and instruction memory. It has up to 22 32-bit registers available for user-declared register variables, 128 KBytes of data memory and 1 MByte of RAM of virtual instruction memory. It controls the PE array and performs operations on singular data. The ACU sends data and instructions to each PE simultaneously.

Each PE is a 4-bit load/store arithmetic processing element with dedicated registers and RAM. Each PE has 40 32-bit registers, up to 33 of which are available for user-declared register variables. Each PE has 16 KBytes(64 KBytes as optional) of RAM. Each PE receives the same instruction on variable that reside on the PEs. In MPL, any variable that is declared as a PE variable is replicated exactly, except for its value, in every PE.

The array of PEs is divided into non-overlapping square matrix of 16 PEs. These square matrices are called PE clusters. PE clusters are used by the MPL language constructs and library routines that are involved with Global Router PE communications(for example, router and connected()). The system can communicate with all PE clusters simultaneously, but it can communicate with only one PE per cluster at one time. At most one outgoing communication connection and one incoming communication connection can be made per cluster at one time.

There are three kind of communications, that is, communication between the front end and the DPU, communication between the ACU and the PE, and communication among the PEs.

Communication between the front end and the DPU is needed sometimes. One reason is that the DPU memory is very limited compared with the front end. Therefore it can not accommodate very big programs. Another reason is that the DPU is less efficient than the front end in dealing with certain operations because the processors of DPU are 4-bits only. A good example is that the DPU is much slower than the front end in doing power operation(0.007s vs. 0.000003s). The third reason is that the front end has a very abundant library, some of which DPU can not support. So appropriate programming model would be generating both front end code and DPU code and Executing them interactively. The communication between FE and DPU is done through the queues or DMA, where the DMA manner is a new feature supplied with the System Software V3.2. The DMA is faster than the queue. For example, in a experiment of transfer 4096 double precision numbers from FE to PEs, the time with queue is 0.054 seconds, while the time with DMA is 0.0068 seconds.

The communication between ACU and PEs is achieved by broadcasting network and OR-Reduction network. The broadcasting network allows data to be broadcast from the ACU to all PEs. The OR-Reduction network provides a boolean logical OR reduction network that enables global PE data conditions to be easily detected.

MasPar provides two mechanisms to implement communication between PEs. They are X-Net and global router network. X-Net allow selected PEs to move data to a neighboring PE, one bit per clock cycle. X-Net communication between any PE and any other PE lies on a straight line from the original PE in one of the eight directions, as seen in Figure 2-2. It should be aware that during one communication, the direction and the distance is unique for any active PEs. Global Router Communication makes it possible to communicate between any particular PE and the members of an arbitrary subset of PEs simultaneously. It transmits data in bit-serial format, one bit per clock cycle.

MasPar provide parallel processing resources. How to use it, however, depends on how users are interacting with it. Users can use high-level languages such as MasPar Fortran, a modified Fortran 90. In this case, one source drives all parts and the compiler manages interactions. These languages automatically generate two cooperating pieces of code: one for the FE and one for the DPU. Therefore, the PEs and communication networks are transparent to users. Alternatively, Users can use lower-level languages. In this case, there is one source for the DPU and one source for the FE. The programmer explicitly manages the DPU-FE interactions by means of subroutine calls that communicate between the two source programs. One of such lower-level languages is MasPar Programming Language(MPL).

The MPL is the lowest level(most efficient and most flexible) programming language that MasPar supports. The purpose of MPL is to program the DPU. Use MPL to recode the appropriate parts of existing applications to execute in a data parallel way. And the front end program calls these MPL subroutines.

MPL is based on ANSI C. In addition, statements, keyword and library functions have been added to support data parallel programming. MPL extensions include:

* A new keyword, plural, distinguishes between two independent address spaces. Variables defined using the keyword plural are allocated identically on each PE in the PE array. Variables defined without using the keyword plural are singular variables and are located on the ACU.

* Plural expressions are supported. All arithmetic and addressing operations defined in ANSI C are supported for plural data types.
* SIMD control statement semantics are implemented.
* Dynamic auto arrays are implemented. These allow you to develop functions that can run on any size of DPU.

An important concept in SIMD programming is the concept of the active set. The active set is the set of PEs that is enabled at any given time during execution. This set is defined by conditional tests in user's program. Plural data operations in an MPL program apply uniformly to all active PEs.

With MPL, a programmer can access any PEs and directly use communication mechanics mentioned earlier in this section. For example, you can access a PE by using the construct proc, you can implement communication between any PEs by using the construct router, you can copy a variable in one PE to the PEs along a certain direction by using the construct xnetc. All these greatly facilitate programmers in parallel programming and make it possible to develop most efficient application programs.

On the other hand, high-level languages such as MasPar Fortran use the parallel resources automatically. And, the compilers have been highly developed. Nevertheless, These compilers are general-purpose ones. They may not be smart enough to optimize all code compiled by them. Thus, it is logical to assert that with the help of MPL, a programmer may develop more efficient code for some application if the application is appropriate to parallel processing and the programmer can reasonably design the program and greatly utilize MPL. More details will be given in the following sections.

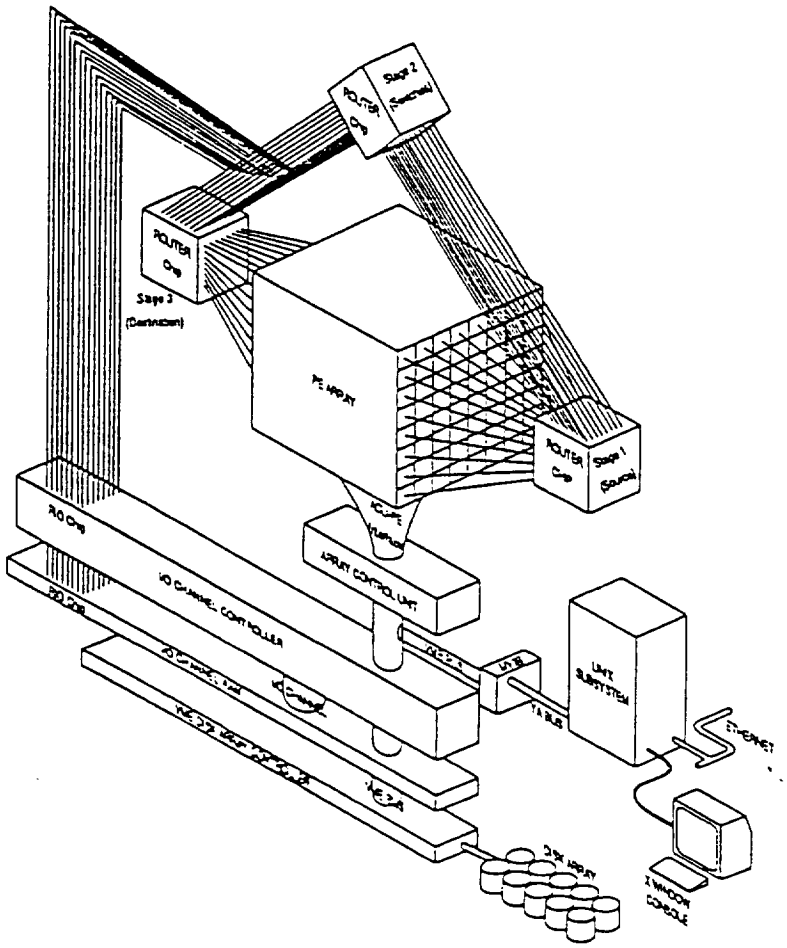


Figure 2-1 MasPar system block diagram
from "MasPar Overview and MPPE Manuals"

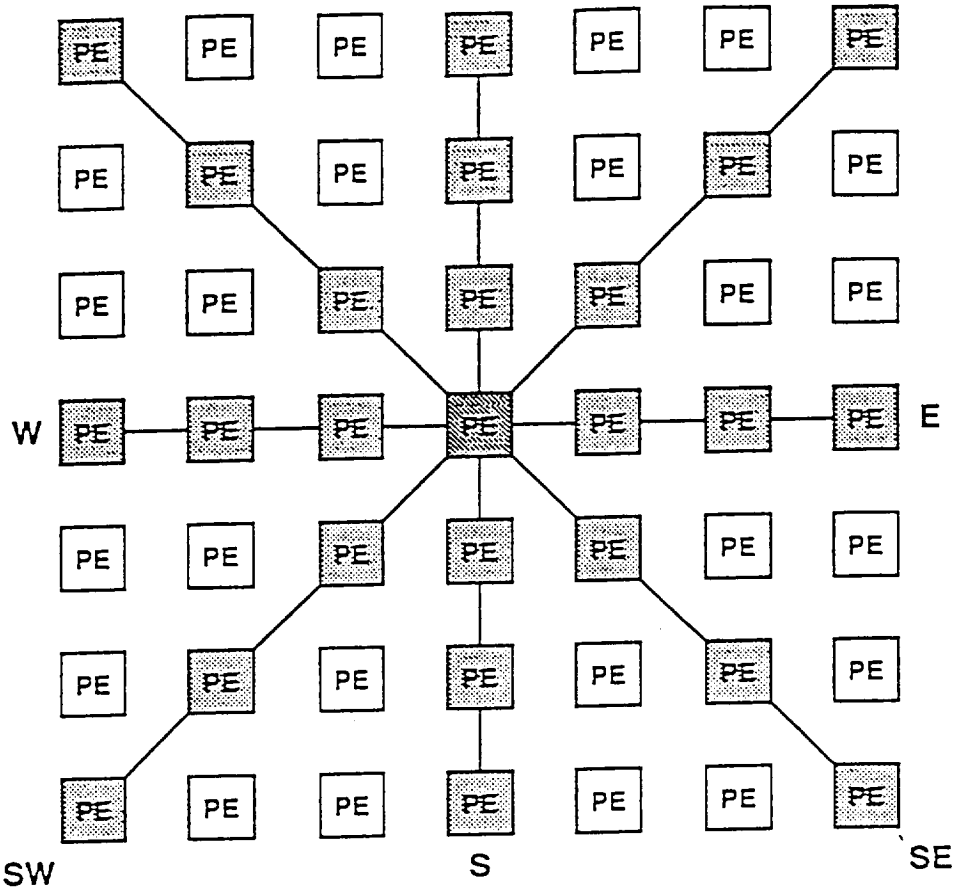


Figure 2-2 Eight x-net directions
from "MasPar MPL Manuals"

3. IMPLEMENTATION OF GENERAL PARALLEL ALGORITHMS ON MASPAR

Some algorithms are commonly used in parallel processing. In this section, some of these algorithms are introduced and implemented on MasPar. They are estimated in terms of time efficiency. We also try to compare communication mechanisms supplied by MasPar.

One of the algorithms is Computing All Sums[3]. Assume that each processor P_i holds in its local memory a number a_i, $1 \le i \le N$. It is often useful to compute, for each Pi, the sum $a_1+a_2+...+a_i$. It can be shown that using the following algorithm the sum of N numbers is computed in O(log N) time instead of O(N) time. The idea is to keep as many processors busy as long as possible and exploit the associativity of the addition operation.

Algorithm 1: AllSums

for i=0 to logN-1 do
  for i=2^j+1 to N do in parallel
    Processor P_i
      (i) obtains a_{i-2^j} from P_{i-2^j} and
      (2) replaces a_i with $a_{i-2^j}+a_i$.
  end for
end for.

The working of AllSums is illustrated in Figure 3-1 for N=8 with Aij referring to the sum $a_i+a_{i+1}+...+a_j$. When the procedure terminates, a_i has been replaced by $a_1+a_2+...+a_i$ in the local memory of P_i, for $1 \le i \le N$.

It is not difficult to extend Algorithm 1 so that it can sum up NxN numbers located in PE array in O(2logN) instead of O(N^2).

In Algorithm AllSums, each PE repeatly takes two actions, getting a number from other PE and adding the fetched number to its local number. The first action deals with communication between PEs. It is obvious a good communication scheme will make the algorithm efficient. In List 3-1, several communication approaches are used and are compared in terms of execution time. This program performs sum of integer numbers located in a portion of the PE array with different distances. The distance between two PEs refers to the minimum number of steps from one PE to the other.

List 3-1 Implementations of AllSums

```
/* Using different communication schemes to perform AllSums */
#include <mpl.h>
#include <stdio.h>
#include <math.h>

extern double dpuTimerElapsed();
extern void dpuTimerStart();
void p_printf();
main()
{
  int x_end, y_end,x_dist, y_dist, x_num, y_num, log_x_num, log_y_num;
  int x_start, y_start;
  plural x_ndx, y_ndx;
  plural int src, dest;
  int i,j,k;
  plural int target,activeSet;
  double et;

  printf("x_dist ? "), scanf("%d",&x_dist);
  printf("y_dist ? "), scanf("%d",&y_dist);
  printf("x_start ? "), scanf("%d",&x_start);
  printf("y_start ? "), scanf("%d",&y_start);
  printf("x_end ?"), scanf("%d",&x_end);
  printf("y_end ?"), scanf("%d",&y_end);

  if(x_start<0 || y_start<0 ||
     x_start>=nxproc || y_start>=nyproc){
    return;
  }
  x_num=(x_end-x_start+1)/x_dist;
  if((x_end-x_start+1)%x_dist)
    x_num++;
  y_num=(y_end-y_start+1)/y_dist;
  if((y_end-y_start+1)%y_dist)
```

```
 y_num++;
log_x_num=log(x_num)/log(2);
log_y_num=log(y_num)/log(2);

x_ndx=-1;
y_ndx=-1;
if(ixproc>=x_start && iyproc>=y_start &&
   ixproc<=x_end && iyproc<=y_end &&!
   ((ixproc-x_start)%x_dist) && !((iyproc-y_start)%y_dist)){
  x_ndx=(ixproc-x_start)/x_dist;
  y_ndx=(iyproc-y_start)/y_dist;
}

/* xnet */
src=1;
/* x-direction */
  dpuTimerStart();
k=1;
for(j=0;j<log_x_num;j++){
  if(x_ndx>=k && x_ndx<x_num){
    src+=xnetW[k*x_dist].src;
  }
  k*=2;
}

/* y-direction */
k=1;
for(j=0;j<log_y_num;j++){
  if(y_ndx>=k && y_ndx<y_num){
    src+=xnetN[k*y_dist].src;
  }
  k*=2;
}
et=dpuTimerElapsed();

/* router */
```

```
src=1;
dest=0;
all activeSet=0;
if(
   ixproc>=x_start && ixproc<=x_end && !((ixproc-x_start)%x_dist) &&
   iyproc>=y_start && iyproc<=y_end && !((iyproc-y_start)%y_dist))
  activeSet=1;
target=nxproc*y_start+x_start;
__routerCount=0;
dpuTimerStart();
while(activeSet){
  if(connected(target)){
    router[target].dest+=src;
    activeSet=0;
  }
}
et=dpuTimerElapsed();

/* sendwith */
src=1;
dest=0;
all activeSet=0;
if(
   ixproc>=x_start && ixproc<=x_end && !((ixproc-x_start)%x_dist) &&
   iyproc>=y_start && iyproc<=y_end && !((iyproc-y_start)%y_dist))
  activeSet=1;
target=nxproc*y_start+x_start;
dpuTimerStart();
if(activeSet)
  dest=sendwithAdd32(src, target);
et=dpuTimerElapsed();

/* sequential */
src=1;
dest=0;
dpuTimerStart();
```

```
 for(j=y_start;j<=y_end;j+=y_dist)
   for(i=x_start;i<=x_end;i+=x_dist)
     proc[y_start][x_start].dest+=proc[j][i].src;
 et=dpuTimerElapsed();
}
```

In this program xnet construct family is used to implement AllSums. The xnet constructs allow each active PE to communicate with a PE that is a uniform distance and direction from the active PE. The syntax of an xnet construct is:

```
xnet[singular_expression].plural_expression
```

where xnet is one of the keywords listed in Table 3-1. One member of xnet family is plain access, which communicate between the active set and connected to set without affecting any other PEs. The pipe constructs attempt to communicate between the active set and the connected to set, but if there is any active set along the pipeline between the two PEs that are attempting to communicate, then the communication is broken. The pipe constructs generally are faster than the plain access constructs. For more information about the timing features see MasPar programming Language(MPL) User Guide. The copy constructs work the same way the pipe constructs work except that the value being communicated is deposited in each PE along the pipeline. Singular_expression is an integer expression that gives the distance between the active and the connected to PEs. Plural_expression is a variable or expression that is accessed in each connected to PE. The plural_expression must be a basic type; it can not be an aggregate such as a structure or union or a whole array. To compensate this limit, MPL supplies xfetch and xsend library routines which are capable of transferring any bytes of data between the active and the connected to PEs. Another important feature of these library routines that the xnet constructs are not of is that the direction of communication is not limited to the eight directions. However this gain is in the cost of execution time. List 3-2 is an example of using the two communication mechanics. In this example, an integer is transferred from the active PEs and the connected to PEs. the execution time are 0.000037 seconds and 0.015142 seconds with the xnet construct and the xsend routine, respectively.

Table 3-1 xnet Keywords

Plain Access	Copy	Pipe

xnetN	xntecN	xnetpN
xnetNE	xnetcNE	xnetpNE
xnetE	xnetcE	xnetpE
xnetSE	xnetcSE	xnetpSE
xnetS	xnetcS	xnetpS
xnetSW	xnetcSW	xnetpSW
xnetW	xnetcW	xnetpW
xnetNW	xnetcNW	xnetpNW

List 3-2 Comparison between xnet construct and xsend routines

```
/* compare xnet and xsend */
#include <mpl.h>
#include <stdio.h>

extern double dpuTimerElapsed();
extern void dpuTimerStart();

main()
{
  int step;
  plural int i, j;
  int dx, dy, nbytes;
  double et;

  /* xnet */
  step=3;
  dpuTimerStart();
  xnetSE[step].j=i;
  et=dpuTimerElapsed();
  printf("time(xnet)= %f0,et);

  /* xsend */
  dy=-3;
  dx=3;
  nbytes=sizeof(i);
```

```
  dpuTimerStart();
  ss_xsend(dy,dx,&i,&j,nbytes);
  et=dpuTimerElapsed();
  printf("time(xsend)= %f0,et);
}
```

Router is another construct that MPL supplies. As mentioned earlier in this section, router construct allows each active PE to communicate with any other PE in an arbitrary way. The syntax of a router construct is:

```
router[plural_index_expression].plural_expression
```

where plural_index_expression specifies which PE wants to connect to. Plural_expression is a variable or expression that is evaluated in each connected to PE. Like the plural_expression in the xnet constructs, it must be basic type. MPL suplies the rsend and rsend library routines which are capable of transferring any bytes of data. The router construct is much more flexible than the xnet construct. However, The performance of the router construct greatly depends on the number of collisions. A collision occurs when more than one active PEs attempt to access the PEs in one cluster. In this case, the active PEs will be waiting in line for access. A random communication pattern with all PEs participating takes an average of 5000 clocks for 32-bit operands, while a left-hand side(LHS) xnet[] operation on 32-bit operands takes 34*dist+6 where dist>1 is the distance between communicating PEs. For more information, please see MasPar Programming Language(MPL) User Guide. By comparing between the router construct and the xnet constructs we conclude that if an application programming needs much flexibility, we would prefer the router construct; if the emphasis is on timing and we can reasonably arrange the PE arrays, the xnet constructs is advantageous.

The program in List 3-1 also applies a parallel primitive routines sendwith. The syntax of a sendwith routine is:

```
sendwith(plural src, plural dest_pe);
```

where src is a plural variable whose value is to be sent and dest_pe is the destination PE for src for each active PE. For each PE in the active set, the sendwith routines return the sum, AND, maximum, minimum, product, or OR of all values of src that

were sent to that PE. The difference between the sendwith routines and the router construct is that with the router construct subsequent sends to the same destination PE overwrite each other; with the sendwith routines all sends to the same destination are combined.

For the sake of comparison between parallel processing and sequential processing the proc construct was used. The proc construct allows user to access a single variable on a single PE, or it allows you to evaluate an expression on a single PE. The syntax of the proc construct is:

```
proc[singular_expression].plural_expression
```

or

```
proc[singular_expression][singular_expression].plural_expression
```

where singular_expression is an integer expression used to index the PE array. In the one-dimensional case, the singular_expression is the linear PE index. In the two-dimensional case, the first singular_expression is the PE row index and the second singular_expression is the PE column index. Plural_expression is a variable or expression that is accessed in the indexed PE.

Table 3-2 shows the execution time to perform Algorithm AllSums with different communication approaches given different distances and numbers of PEs. Figure 3-2 is the plot of time vs number of PEs with the distance of one. Observing Table 3-2 and Figure 3-2, we can find that when the number of PEs is small, the router construct, even the sequential approach with the proc construct is superior over the xnet construct. This is because, on one hand, fewer collision occurred to router construct. On the other hand, for the xnet approach, the computing steps are not economical for the small number of PEs. However, as the number of PEs increases, the increase of execution time with the router construct is much faster than that of the xnet approach. Same thing is true for the proc construct.

Table 3-2 Execution Time with Different Communication Schemes

distances		start		end		time(second)			
x	y	x	y	x	y	xnet	router	sendwith	sequential
1	1	0	0	1	1	.005770	.000391	.014771	.000182
1	1	0	0	3	3	.005892	.001505	.014928	.000675
1	1	0	0	7	7	.005849	.005961	.014931	.002607
1	1	0	0	15	15	.005992	.023782	.015341	.010336
1	1	0	0	31	31	.006280	.095068	.016108	.041022
1	1	0	0	63	63	.006449	.380211	.015414	.163509
2	2	0	0	1	1	.005630	.000113	.014560	.000057
2	2	0	0	3	3	.006137	.000393	.014840	.000184
2	2	0	0	7	7	.005826	.001505	.014878	.000672
2	2	0	0	15	15	.005955	.005961	.014724	.002607
2	2	0	0	31	31	.006148	.023782	.015051	.010336
2	2	0	0	63	63	.006505	.095067	.015073	.041026
4	4	0	0	1	1	.009420	.000112	.014828	.000058
4	4	0	0	3	3	.005627	.000112	.014579	.000058
4	4	0	0	7	7	.005796	.000391	.014661	.000182

It is important to note that Algorithm AllSums can be modified to solve any problem where the addition operation is replaced by any other associative binary operation. Furthermore, the idea of this algorithm can be applied to general parallel programming design other than merely binary numerical accumulations.

Finally in this section, some concepts regarding MPL language would be clarified. The control statements in ANSI C, such as *if*, *while* and *switch*, are expanded to handle plural variables and expressions. They are very helpful in parallel programming. But don't consider the branches in these control statements are performed parallelly. For example, in the conditional statement

```
if (expression) statement block 1
else statement block 2
```

the block 1 and the block 2 are performed in serial. Another feature that MPL provides is the plural pointers. There are four kind of pointers in MPL: singular pointer pointing

to singular data, singular pointer pointing to plural data, plural pointer pointing to singular data and plural pointer pointing to plural data. These pointers greatly facilitate memory accessing. but this gain is in the cost of time efficiency due to the complicated pointer manipulation.

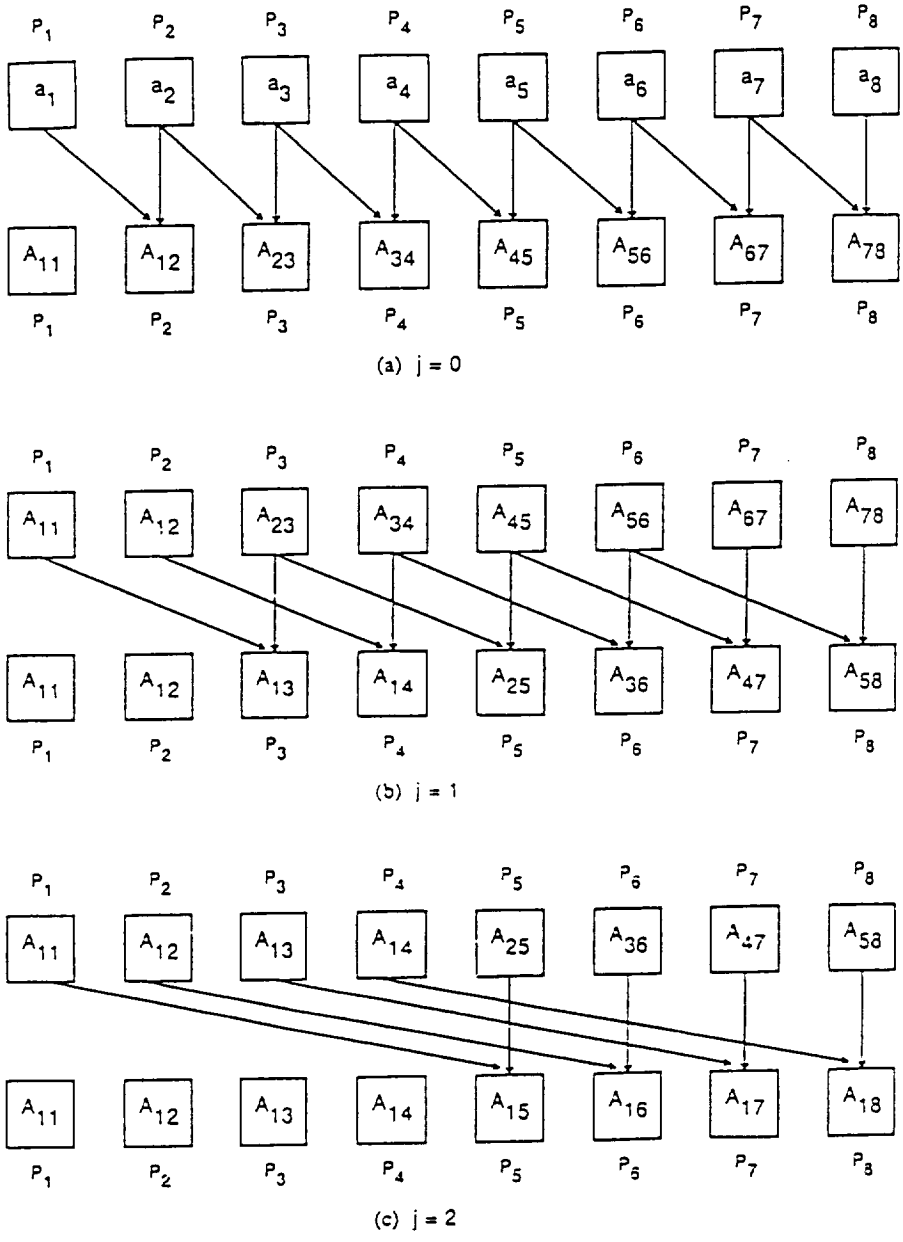


Figure 3-1 Computing the sums of eight numbers using Algorithm AllSums

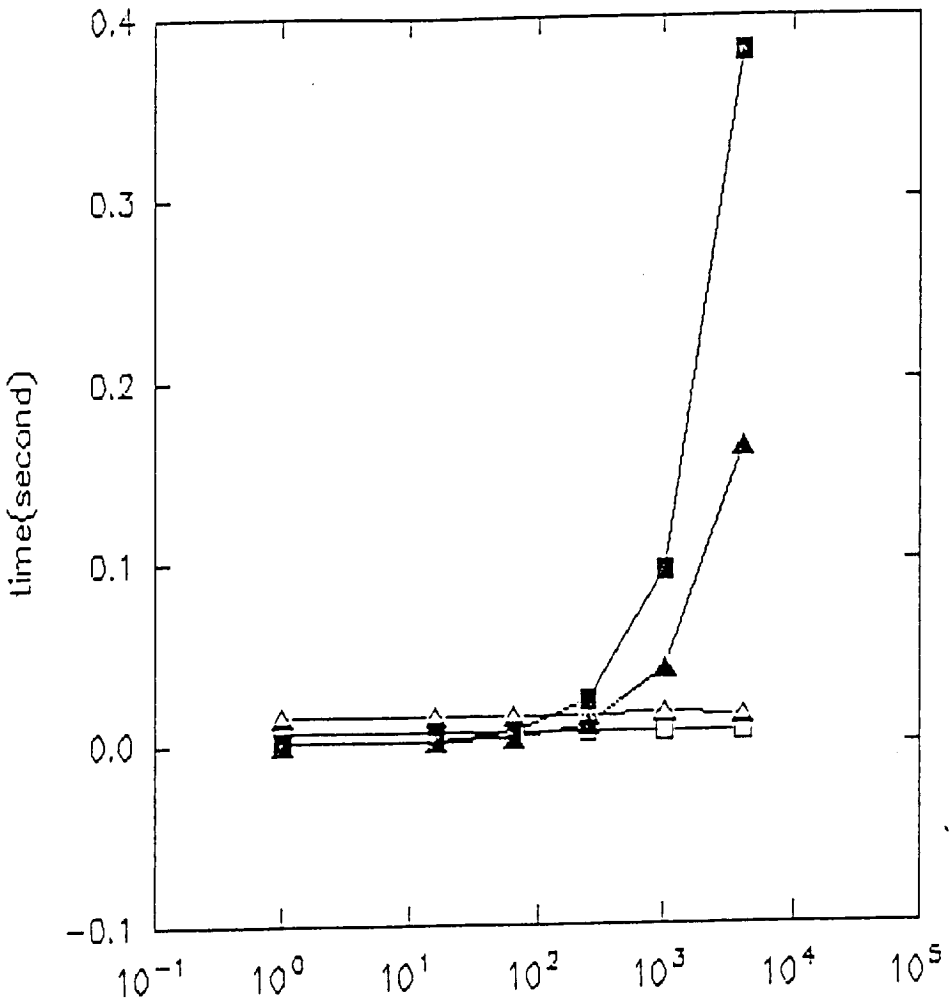


Figure 3-2 Execution time of computing a bunch of integers using different communication mechanics
Black square: router
Black triangle: proc
white triangle: sendwith
white square: x-net

4. PARALLEL PROGRAMMING FOR MATERIAL ANALYSIS

There are some common themes to any data parallel programming. Let us see a simple example of parallel programming first. Suppose we want to perform the addition of two matrices and compute the trace of the resulting matrix. First, we should choose appropriate representation on parallel machine. On serial machine, a matrix is represented in a two-dimensional array. On parallel machine, a natural representation for the matrix is to assign each element of the matrix to a uniform variable in a PE. After determining the data structure, we should figure out how to do the addition and trace computation. The addition easily is performed by adding the two variables representing the elements of two matrices in all involved PEs parallelly and assign the result to a new variable. Then, to perform trace computation, sum up the new variable on some PEs that represent the diagonal elements of the new matrix by using some sum algorithm(of course you can use Algorithm AllSums). The addition is done locally in each PE, but in computing the trace, communication between PEs is needed. This algorithm is partitioned into two tasks, ADDITION and TRACE. They are performed with some subalgorithms. TRACE is dependent of ADDITION. With machines of MIMD architecture, this two tasks may run concurrently, but some synchronization mechanism may be needed. With SIMD machines, on the other hand, these two tasks are done in order and there is no need for synchronization. Of course, we should not forget to initialize PEs at the beginning.

Generally speaking, in performing a parallel processing, we need:

Establishing Parallel and/or scalar Data structures. Data parallel programs can be expressed in terms of the same data structures used in serial programs. The difference is that the individual elements of a composite data structure, such as an array, are spread across PEs, so that each data element has an associated processor. Since each PE has its own dedicated memory, the task of associating data elements with PEs is simply the task of assigning memory locations across processors. In some cases, data are scalar, thus they are declared and stored in the ACU or in the front end.

Establishing Linkages Among Data Elements. During the execution of a program, data from different problem elements are used together. The linkages among them are mainly via communications. Sometimes the pointers and subscripts are subsidiary.

Synchronizing tasks. For SIMD machine, All active PEs follow an unique instruction at any time. Thus synchronization of tasks does not pose any problems. Sometimes, asynchronous computation within an algorithm is desired. For example, an variable in one PE is dependent of an variable of another PE. In this case the first PE attempting to evaluate the local variable has to wait until the variable in the second PE has been

evaluated. This delay may be done by disabling the first PE.

The execution time of a parallel code mainly is spent on PE initialization, parallel computing and communications. PE initialization is necessary, but it usually takes a small portion of the execution time. Therefore designing a reasonable parallel computing schedule and using an appropriate communication schemes is critical for the time efficiency of the program. It should be mentioned that the choices are very dependent on applications and machines.

The program package we are working on is the one that models the mechanical performance of materials, such as concrete constitution, with the aid of finite elements method. Generally speaking, a material is characterized by a sequence of elements. By computing some mechanical quantities, such as strain and stress on these elements, we get the performances of this kind of materials. Here we call an element a cell. During one execution, a sequence of cells, say 2000 cells, are computed. Each cell is divided into nxn (n=4 at present) subcells. These subcells have a uniform structure as the following:

entries 1 - 6: total strain;

entries 7 - 12: micro quantities (for Micromechanics Model);

entries 13 - 18: inelastic strain;

entries 19 - 30: state variables;

The original package was coded and run on machines of Von Neumann architecture. A list of simplified execution steps are shown below:

1. user inputs parameters for the material to be analized
2. system initialization
3. history control adjustment for computing next cell
4. compute a cell, including computing subcells within the cell
5. if the number of computed cells has not reached the limit, go to 3
6. end.

In sequential programming, a subcell is expressed by an one dimensional array of size 30. A cell is expressed in the form of a subcell sequence. The computation is done with two-layer iteration. In inner layer, the subcells of a cell are computed. In outer layer, the cell sequence is computed.

A. Developing Parallel Algorithm on DPU

Here, we will concentrate on finding appropriate parallel algorithms for computing subcells. As stated previously, subcells have the identical structure and are independent of each other. This implies that the operations on each subcells are the same. It is natural to assign one PE to a subcell and compute these subcells in parallel. In order to show the power of parallel processing in computing the subcells, an experiment was done. One PE was assigned to each subcell and the original sequential algorithm was implemented without major modification. Each time a different number of subcells were processed simultaneously. Table 4-1 shows the execution time with different number of PEs. Figure 4-1 is the corresponding plot. By this experiment, we can find that the more the PE in parallel computation, the faster the speed.

Table 4-1 comparison of execution time with different numbers of PEs in parallel

number of PEs	time(second)
1	0.32
2	0.25
3	0.18
4	0.13
5	0.12
6	0.11
7	0.11
8	0.077
9	0.077
10	0.077
11	0.077
12	0.076
13	0.076
14	0.076
15	0.066
16	0.047

In the above experiment, the parallel processing between subcells was achieved. the processing within a subcell, however, remained sequential. In fact, It is possible that some quantities and intermediate variables within a subcell can be computed in parallel. In order to demonstrate this possibility, a part of the original FORTRAN 77

code, which deals with subcell computation, was shown in List 4-1. In the code the one-dimensional array SA of size 30 is the input and the one-dimensional array DSA of size 30 in the output. There are some intermediate arrays of size 6 each, such as SS, S and R. Rationally, we attempt to assign the elements of these arrays across the PEs. The total of PEs needed is 78. Looking into the code, however, we find it is possible to use fewer PEs. The array SS actually is the duplication of a trunk of the input SA. Therefore it can be deleted without any problem. SA itself also is redundant, since only SA(7) through SA(12) and SA(21) through SA(26) are used. And, for the output DSA, only the elements DSA(13) through DSA(26) are computed, where DSA(13) through DSA(18), DSA(21) through DSA(26) have uniform or almost uniform expressions for each group. The elements of the array S have almost uniform expression and are independent of each other. The elements of the array R have exactly uniform expression and are independent. So parallel processing can be achieved within each of these groups. Usually parallelism can not be achieved between groups, since they have different expression and/or are data dependent.

List 4-1 FORTRAN Code(Sequential) for Computing a Subcell

```
C######################################################################
      SUBROUTINE BODNER(DSA,SA)
C
C     PURPOSE:  BODNER-PARTOM VISCOPLASTIC MODEL
C
C     NOTE:  1) IN THIS SUBROUTINE, [SA] AND [DSA] CONTAIN THE
C               "MICRO" QUANTITIES FOR ABOUDI'S MICROMECHANICS MODEL
C
C            2) ARRANGEMENT OF [DSA] & [SA]  ARRAYS
C
C            VARIABLE            LOCATION
C          ____________________________________
C         |
C         | STRAIN RATE          (1-6)
C         |-----------------------------
C         | STRESS RATE          (7-12)
C         |-----------------------------
C         | INELASTIC
```

```
C        | STRAIN RATE          (13-18)
C        |------------------------------
C        | 12 "SLOTS"           (18-30)
C        | FOR STATE VARIABLES
C        |_____________________________
C
C
C     CALLED FROM:  NPARM
C---------------------------------------------------------------------
C
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      LOGICAL NDEBUG
C
      COMMON /DEBUG/ NDEBUG
      COMMON /IOUN/ DEBUG
      include 'micro.inc'
      include 'vpv.inc'
c      COMMON /VPT/ NVP(6,2),MATT(16)
      include 'vpt.inc'
C
      DIMENSION SS(6),S(6),R(6),DSA(30),SA(30)
C
C
C---------------------------------------
C  SET-UP VISCOPLASTIC MATERIAL CONSTANTS
C---------------------------------------
C
        IT=MATT(NM)
        JS=NVP(IT,1)
        D0=VP(JS)
        Z0=VP(JS+1)
        Z1=VP(JS+2)
        BM=VP(JS+3)
        AN=VP(JS+4)
        Q=VP(JS+5)
C
```

```
C-------------------------------
C  COPY STRESS FROM [SA] TO [S]
C-------------------------------
C
      SS(1) = SA(7)
      SS(2) = SA(8)
      SS(3) = SA(9)
      SS(4) = SA(10)
      SS(5) = SA(11)
      SS(6) = SA(12)
C
C-------------------------------------------------
C  COMPUTE THE DEVIATORIC STRESS [S] IN THE SUBCELL
C-------------------------------------------------
C
        TEMP = (SS(1) + SS(2) + SS(3))/3.
        S(1) = SS(1) - TEMP
        S(2) = SS(2) - TEMP
        S(3) = SS(3) - TEMP
        S(4) = SS(4)
        S(5) = SS(5)
        S(6) = SS(6)
C
C----------------------------------------
C  PREDICT THE AVERAGE PLASTIC STRAIN-RATE
C  IN THE SUBCELL (BODNER)
C----------------------------------------
C
        AJ2=0.5*(S(1)**2+S(2)**2+S(3)**2)+S(4)**2+S(5)**2+S(6)**2
        SQ3AJ = DSQRT( SS(1)**2 + SS(2)**2 + SS(3)**2 +
     &                    2*(SS(4)**2+SS(5)**2+SS(6)**2)  )
        SQ2=1.414215
        IF(SQ3AJ.EQ.0.) THEN
          CALL ZEROR(R,6)
        ELSE
          R(1) = SS(1)/SQ3AJ
```

```
          R(2) = SS(2)/SQ3AJ
          R(3) = SS(3)/SQ3AJ
          R(4) = SQ2*SS(4)/SQ3AJ
          R(5) = SQ2*SS(5)/SQ3AJ
          R(6) = SQ2*SS(6)/SQ3AJ
         ENDIF
C
C-----------------------------------------
C  IF D0=0 THEN ASSUME ELASTIC AND ZERO-OUT
C  [DSA] AND RETURN
C-----------------------------------------
C
       IF(D0.EQ.0) THEN
         DO 100 JJ=1,30
           DSA(JJ) = 0.0
100        CONTINUE
         RETURN
       ELSE
         ZEF = Z0 + Q*SA(20) +
     &     (1-Q)*(R(1)*SA(21)+R(2)*SA(22)+R(3)*SA(23)+
     &              R(4)*SA(24)+R(5)*SA(25)+R(6)*SA(26))
C
         IF(AJ2 .EQ. 0.) THEN
           AL=0.0
         ELSE
           ARG1=ZEF**2.0/(3.*AJ2)
           IF(ARG1.GT.1E6) ARG1=1E6
           CON=.5*(AN+1.)/AN
           ARG=CON*(ARG1)**AN
           IF (ARG.GT.50.)  ARG=50.
           AL=D0/(DEXP(ARG)*DSQRT(AJ2))
         ENDIF
C
C-------------------------
C  INELASTIC STRAIN RATES
C-------------------------
```

```
C
      DSA(13) = AL*S(1)
      DSA(14) = AL*S(2)
      DSA(15) = AL*S(3)
      DSA(16) = 2*AL*S(4)
      DSA(17) = 2*AL*S(5)
      DSA(18) = 2*AL*S(6)
C
C--------------------
C  PLASTIC-WORK RATE
C--------------------
C
      WPD =  S(1)*DSA(13) + S(2)*DSA(14) + S(3)*DSA(15) +
     &       S(4)*DSA(16) + S(5)*DSA(17) + S(6)*DSA(18)
C
C-----------------------
C  STATE VARIABLE RATES
C-----------------------
C
      DSA(19) = WPD
      Z0M=BM/Z0
      ZD=Z0M*(Z1-ZEF)*WPD
      DSA(20)=ZD
       DSA(21)=ZD*R(1)
       DSA(22)=ZD*R(2)
       DSA(23)=ZD*R(3)
       DSA(24)=ZD*R(4)
       DSA(25)=ZD*R(5)
       DSA(26)=ZD*R(6)
      ENDIF
     RETURN
     END
```

There are some intermediate variables, such as AJ2, SQ3Aj and ZEF. The expressions associated with these variables are in the form of $\sigma U*V$. Instead of computing the formula sequentially, Algorithm AllSums, developed in Section 3, is applied. These

variables can be evaluated in parallel if they are independent of each other, otherwise they are evaluated sequentially.

In our programming we mainly use the x-net constructs to conduct communications due to their steadily fast speed. Besides the distance, a factor that may affect the communication efficiency is the permutation of PEs. The code in List 4-2 tries to sum up the numbers located in a group of PEs with different permutations indicated in Figure 4-2. Table 4-2 shows the processing time with each permutation. The 2-SQR style is the best. As mentioned earlier in this section, some PEs are assigned to perform uniform operations and the data on them are independent, hence, they can work in parallel. We call the set of these PEs a parallel PE group. Note that the sizes of parallel PE groups in our program are all six. Also note that at sometime, some groups can work in parallel. Based on this definition, we can estimate the total PEs needed in a program. Let M denote the size of the parallel group which is the biggest of all the groups, and N denote the maximum number of parallel groups that may work in parallel at some time throughout the execution of the program. Then, the minimum number of PEs that should be employed is M*N. In other word, If we use fewer PEs than this number, we would get a longer processing time. In our program, the sizes of all groups equally are 6. The maximum number of groups that can work in parallel is 2(remember the groups for computing AJ2 and SQ3Aj). Thus the minimum number of PEs needed is 12 for each subcell. There are 16 subcells in a cell. Thus, the minimum number of PEs needed to process one cell is 192.

List 4-2 code for computing AllSums using x-net constructs with different PE permutations

```
#define SECTION_LENGTH 8
#define LOG_SECTION_LENGTH 3
main()
{
  int i,j,k;
  double dpu_t;
  plural double src, dest;

  /*1-D */
  dpuTimerStart();
    if(within_sec_ndx<6)
```

```
    src=(plural double)within_sec_ndx;
  else
    src=0;
  k=1;
    for(j=0;j<LOG_SECTION_LENGTH;j++){
      if(within_sec_ndx>=k && within_sec_ndx<SECTION_LENGTH){
        src+=xnetW[k].src;
      }
      k*=2;
    }
 dpu_t=dpuTimerElapsed();

/* 2-D */
 dpuTimerStart();
   src=(plural double)within_sec_ndx;
    src=xnetNW[1].src+xnetN[1].src+xnetNE[1].src;
    src+=xnetE[3].src;
 dpu_t=dpuTimerElapsed();

 /* circle */
 dpuTimerStart();
   src=(plural double)(row_ndx*3+within_sec_ndx);
   if(active){
     src+=xnetNW[1].src+xnetN[1].src+xnetNE[1].src+
          xnetW[1].src+xnetE[1].src;
   }
 dpu_t=dpuTimerElapsed();

/* 2-squares */
 dpuTimerStart();
   src=(plural double)within_sec_ndx;
   src+=xnetN[1].src;
     src+=xnetE[1].src+xnetE[2].src;
 dpu_t=dpuTimerElapsed();
}
```

Table 4-2 Processing Time with Different Permutation of PEs

1-D	2-D	Circle	2-Square
.032128	.022023	.025813	.019231

Intuitively, the larger the number of PEs, the faster of the speed. Thus, we always try to have as many as PEs working in parallel. Another point that should be mentioned here is the processing time also depends on communication schemes which in turn affects the number of PEs to be used and their permutations. For example, After performing communication with 2-SQR scheme, the accumulation value is only on PE 3(see Figure 4-2). Suppose the value is wanted by all the six PEs in the group, we must broadcast it in two steps. First, transfer it to the south; and then, transfer it to the east. If we add a row of three PE in bottom, as shown in Figure 4-2(e), only one broadcast step(to the east) is needed. Thus time is saved in the cost of PEs.

We took advantages of MasPar architecture and MPL language, however, we should avoid their limitations. For example, the power operation on DPU is much less time-efficient comparing to that on the front end. Therefore it is better to replace pow(x, 2) with x*x. The execution time of a plural power operation is 0.007261 seconds, compared to 0.000080 seconds of a multiplication operation.

The parallel algorithm for subcell computation is shown in List 4-3 . Using this algorithm the execution time to compute a cell is 0.005486 seconds, 58 times faster that that of the sequential algorithm running on DPU.

List 4-3 The MPL Code for Computing a cell

```
plural double d0=0, z0, z1, bm, an, q;   /* used in BODNER */
plural int subcell_ndx; /* index to subcells */
plural int within_sec_ndx; /* order number within a section */
plural int sec_ndx; /* index to section within a subcell */
int x_max, y_max; /* used to simulate PE arrays of different sizes*/
int rows_per_subcell;   /* # rows for each subcell */
plural int row_ndx;   /* row index within each subcell */
plural double coef0;  /* for s */
plural double coef1;  /* for aj2 and sq3aj (0.5, 1 and 2) */
plural double coef2;  /* for sq2 (1 and 1.424215) */
plural double coef3;  /* for dsa[12-17] (1 and 2) */
plural double sa6p, sa20p,temp, sa19, arg, zd;
```

```
plural active, active0, active1;
int bodner()
{
  plural register double src, dest;
  plural register double s, r=1;
  plural register double zef=0, al=0;
  plural register double arg1, wpd;
  register int i,j, k;

     /* compute deviatoric stress s in each subcell */
     s=sa6p-temp;
     /* predict average plastic strain-rate in each subcell */
     /* compute aj2 in 0th section, sq3aj in 1st sec. */
     src=coef1*s*s;
  src+=xnetS[1].src;
  if(active){
    src+=xnetE[1].src+xnetE[2].src;
    xnetcE[2].src=src;
  }

       if(src==0){
         r=0;  /* 1st sec */
         al=0;  /* 0th sec */
       }
       else{
         dest=p_sqrt(src);
         r=coef2*sa6p/dest; /*sa6p=sa[6+within_sec_ndx]*/
         zef=r*sa20p;
         zef+=xnetS[1].zef;
         if(active1){
           zef+=xnetE[1].zef+xnetE[2].zef;
           zef=sa19+(1-q)*zef;
           /* transfer zef to 0th section where it is used to compute al */
           xnetcW[SECTION_WIDTH].zef=zef;
         }
         arg1=zef*zef/(3.0*src);
```

```
        arg*=p_pow((arg1<=1e6?arg1:1e6),an);
        al=d0/(p_exp((arg<=50.?arg:50.))*dest);
      }

    /* inelastic strain rate */
    dest=coef3*al*s; /* here in section 0 dest corresponds to dsa[12-17] */
    wpd=s*dest;
    /* accumulate src to get wpd */
 wpd+=xnetS[1].wpd;
 if(active0){
   wpd+=xnetE[1].wpd+xnetE[2].wpd; /*dsa[18] */
   /* plastic-work rate */
   /* state variable rate */
   zd*=(z1-zef)*wpd; /* dsa[19] */
   /* here dest is original zd. transfer it to 1st section to
      compute dsa[20-25] */
   xnetcE[SECTION_WIDTH].zd=zd;
 }
 zd*=r;
}
```

B. Comparison between the Front End and the DPU

Performance comparison between the FE and DPU is helpful in algorithm design and task assignment. Table 4-3 shows some features of DECstation 5000 and the DPU MP-1104.

Table 4-3 Some Features of DECstation 5000 and MP-1104

	DECstation 5000	MP-1104
Number of PEs	1	4096
32-bit integer MIPS	24	6400
32-bit floating point MFLOPS	7	300
64-bit floating point MFLOPS	4	138

It should be noted that the MIPS of the DPU is the combination of all the PEs. For single PE on DPU, it is much less powerful than that of the front end. For example, the 64-bit floating MIPS for single PE on DPU is 138/4096=0.034, about 119 times slower than that of the FE. The early example of power computation proved this difference. On the other hand, the advantage of DPU over the front end is massive PEs working together. The example in List 4-4 demonstrates this advantage. In the example, we compute the sum of 4096 double precision numbers, and then each of these number is divided by the sum plus another number. On the front end this is done by two iterations, one for the sum, the other for the division. The execution time is 0.011718 seconds. On the DPU, this calculation is done just in two statements. The execution time is 0.000385 seconds.

List 4-4 Arithmetic Operations on FE and DPU

```
#include <stdio.h>
#include <stdlib.h>

extern mpl_sub();

#define size 4096

double flyingMonkeys[size];

main()
{
  double munckins[size];
  double glinda, wickedWitch;
  int i;

  for(i=0;i<size;i++)
    flyingMonkeys[i]=(double) i;
  wickedWitch=1.23;

  callRequest(mpl_sub, 20, &wickedWitch, &glinda, flyingMonkeys, munckins,&dpu_t);

  /* pure FE */
```

```
  glinda=0;
  for(i=0;i<size;i++)
    glinda+=flyingMonkeys[i];
  for(i=0;i<size;i++)
    munckins[i]=flyingMonkeys[i]/glinda+wickedWitch;
}

/* This is the mpl routine */
#include <mpl.h>

visible mpl_sub();
visible extern double flyingMonkeys[];

mpl_sub(fe_wickedWitch_p, fe_glinda_p, fe_flyingMonkeys_p,
        fe_munckins_p, fe_dpu_t_p)
  void *fe_wickedWitch_p;
  void *fe_glinda_p;
  void *fe_flyingMonkeys_p;
  void *fe_munckins_p;
  void *fe_dpu_t_p;
{
  plural double munckins, Monkeys;
  double glinda, wickedWitch;

  copyIn(fe_wickedWitch_p, &wickedWitch, sizeof(wickedWitch));
  blockIn(fe_flyingMonkeys_p, &Monkeys, 0, 0, nxproc, nyproc,
          sizeof(Monkeys));

  glinda=reduceAdd64(Monkeys);
  munckins=Monkeys/glinda+wickedWitch;

  copyOut(&glinda, fe_glinda_p, sizeof(glinda));
  blockOut(&munckins, fe_munckins_p, 0, 0, nxproc, nyproc, sizeof(munckins));

}
```

Another thing that can not be ignored in the example is its communication expense on transfering the numbers between the FE and the DPU. By including the communication time, the total time with DPU is 0.089838 seconds. The communication is the bottle neck.

MasPar supplies several kinds of timers. Each for a different purpose:
1. dpuTimer: this is an MPL routine, although it can be called from MPF. It measures DPU time only, and does not include the front-end. It is the finest grained timing routine, and measures time in increments of 80ns ticks. This timer is only good for measuring up to 5 minutes and 43 seconds, after which it loops back round to zero.
2. mpTimer: this an MPF routine. It measures overall run time (FE+DPU). It is based on the UNIX clock, and is quite coarse grained, and is not very accurate for timing parallel code. It can be significantly affected by the workload on the front-end.
3. mpCpuTimer: can be called from MPL or MPF. It measures dpu time only, but excludes job context switching time and I/O. It is accurate to approx 20ms, and wraps around after 1193 hours.

It should also be pointed out that:
1. DPU time refers to the amount of time a program has been executing on the dpu, and is the value printed in the RUNTIME field of mpq.
2. CPU time is similar to DPU time except that it excludes I/O time.
3. USER time is workstation terminology and refers to the amount of time actually spent executing the user's code and excludes system cpu overheads. (The closest thing to this on the MasPar is CPU time.)
4. REAL time is the elapsed or wall clock time. On the MasPar, this will include time spent queuing for the dpu, and the time other jobs sharing the dpu in execution. It is the WALLTIME field in mpstat, or the sum of the queuing time and the PMEMTIME in mpq.

In addition to above routines, there are other time-related routines and system calls, such as gettimeofday and clock. Practically, gettimeofday and clock are not very useful for MasPar programs because they will be affected by other jobs which are sharing the dpu.

Here is the guideline of using the timers for different purpose. If you want to time small parallel sections of a Fortran program, then you should use the dpuTimer or mpCpuTimer routines, which only record DPU cycles. dpuTimer is more accurate, but mpCpuTimer can time longer intervals and is not affected by the design flaw in the ACU. If you want to time the whole program (parallel+serial) parts, then you should use the mpTimer routine, but be aware that the performance of the serial parts of your

program will be affected by the load on the front-end, so you might want to arrange to do your timings when the machine is relatively quiet.

The original code for this project was written in FORTRAN 77. Because the MasPar machine does not a FORTRAN 77 compiler, some modifications in syntax and data structure were made and the code was compiled with MasPar FORTRAN(MPF).

By default Mpfortran compiler automatically assign tasks to both FE and DPU. MPF will use the DPU for parallel parts of the program (i.e. Fortran 90 constructs), and the FE for the serial parts. However, You can force everything to be executed on the FE by using the scalar compiler switch. This facilitates us to compare the execution time of each part of the program. Table 4-4 shows the times of MPL code, MPF code runing purely on FE and MPF code runing on both the FE and the DPU.

Table 4-4 Processing Time for One Cell

MPL on DPU	MPF on both FE and DPU	MPF on FE only
0.005486	0.0054	0.0038

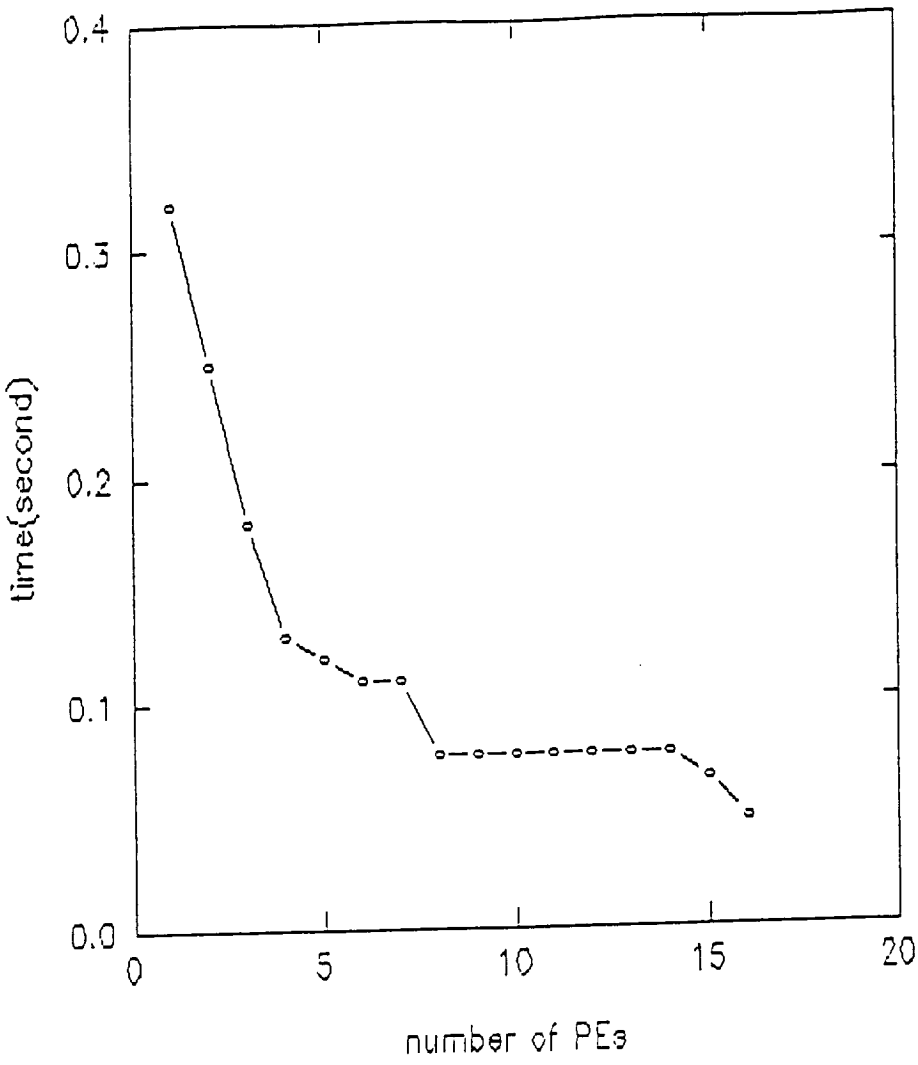
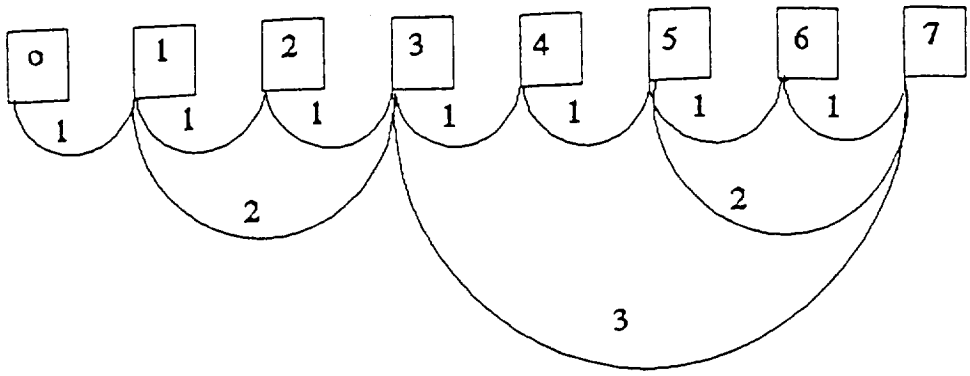
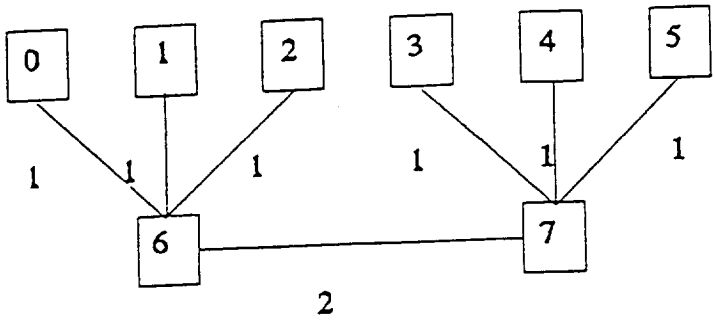


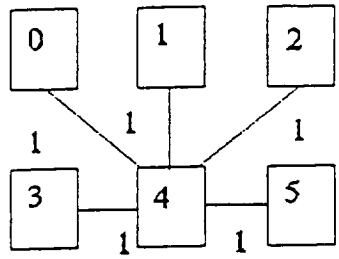
Figure 4-1 Processing time with different subcells in parallel



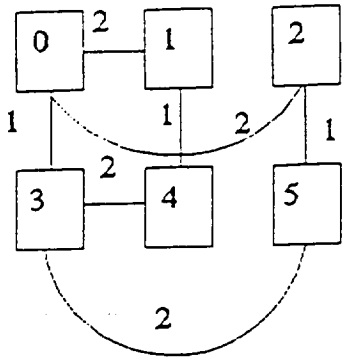
(a) 1-D, 3-step



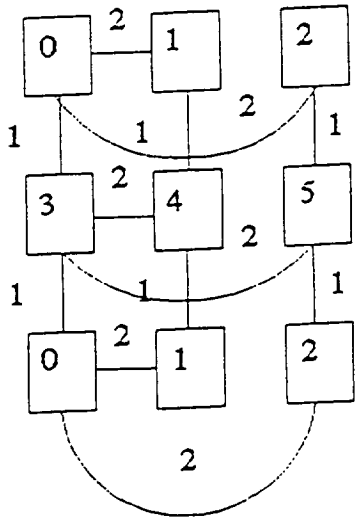
(b) 2-D, 2-step



(c) Circle, 1-step



(d) 2-SQR, 2-step



(e) modified 2-SQR

Figure 4-2 X-net communications with different PE permutation styles

5. SUMMARY

In the previous sections, we introduced the features of a parallel machine MasPar and its parallel programming language MPL. Based on its features, we had developed a parallel algorithm for the material analysis package.

Generally speaking, parallel processing can be divided into three parts: task assignment, data operation and communications, i.e. 1) before starting programming, we should make sure if the application fits for parallel programming and consequently, we should figure out those parts that can be parallelly processed; 2) during program execution, we should assign data to PEs and initialize these PEs and in fact an application can not be processed in wholely parallel way due to data dependency, thus communications among PEs are necessary.

In this project, we first concentrated on subcell computation because it is an ideal candidate for parallel processing. Although, by comparison of the parallel algorithm and the sequential one, we did not feel superiority of the parallel algorithm to its counterpart. However, if we look into the problem deeply, we can find a positive answer, i.e. 1) Our experiment on subcell computation was based on 16-subcell structure, and in applications, the number of subcells in a cell may be far more than sixteen. Suppose the number is 7x7, then for the sequential algorithm the processing time would be three time longer than that for 4x4 structure. On the contrary, in parallel processing, because the 49 subcells are independent of each other, computation time will be the same as long as there are enough PEs. 2) In this paper we mainly worked on subcell computation. Although it is hard to implement parallel processing in cell level due to the correlation between cells, it is possible to implement it in partially parallel model. One potential approach might be dividing the cell series into some subseries and processing these subseries in parallel or pipeline. Of course that needs us to do more work in algorithm design. 3) Currently, a single PE is much less powerful than the processor of the front end. Suppose the 4-bit store/load PE convoluted to 32-bit (which is already available), the processing speed will four times faster ideally. In summary, SIMD is appropriate for material analysis and has a great potential for the development of parallel algorithms.

6. REFERENCES

[1] R. Michael Hord, "Parallel Supercomputing in SIMD Architectures", CRC Press, 1990

[2] W. Daniel Hillis, "The Connection Machine", The MIT Press, 1985

[3] Selim G. Aki, "The Design and Analysis of Parallel Algorithms", Prentice Hall, 1989

[4] MasPar Computer Corporation, "MasPar MPL Manuals", 1992

[5] MasPar Computer Corporation, "MasPar Overview and MPPE Manuals", 1992

[6] MasPar Computer Corporation, "MP-1 System Hardware Manuals", 1992

[7] MasPar Computer Corporation, "MasPar Fortran Manuals", 1993

APPENDIX

MPL CODE FOR THE MATERIAL ANALYSIS PACKAGE

```
/*********************************************************/
/* This function calculates inelasticity. it is called by*/
/* gcmp where it was implemented with a loop structure.  */
/* Here two plural arrayes gjsigl[30] and gjdsig[30]     */
/* were defined to express subcells. This function may   */
/* calls bodner, model3m and model2m.                    */
/*********************************************************/
#include <mpl.h>
#include <math.h>
#include <time.h>

typedef struct { plural int *addr;
                 unsigned int rank;
                 unsigned int OnDpu;
                 unsigned int extent[7];
               } DopeVector;

/* the following structures  except SUBCELL are corresponding
   "common blocks" in FORTRAN code */
typedef struct { double *dsigal_p, dsigal[510];
                 double *sigal_p, sigal[510];
                 double *esp_p, esp[384];
               }CELL;

typedef struct { int    *matnum_p, matnum[4][4];
                 double *vf_p, vf;
                 int    *nm_p, nm;
                 int    *ns_p, ns;
                 int    *nst_p, nst;
                 int    *nmt_p, nmt;
                 int    *nb_p, nb;
                 int    *ng_p, ng;
                 int    *idp_p, idp;
                 int    *nsfd_p, nsfd;
                 int    *lop_p, lop;
                 int    *ncmd_p, ncmd;
               } MICRO;

typedef struct { int    *nvp_p, nvp[2][6];    /* original NVP(6,2) */
                 int    *matt_p, matt[16];
               }VPT;

typedef struct { double *vp_p, vp[50];
               }VPV;

CELL cell_s;
MICRO   micro_s;
VPT     vpt_s;
VPV     vpv_s;

int para_trans(), cell_on_pe(), para_back_trans();
int bodner();

plural double sigl[30], dsigl[30]; /* subcell */
plural double d0, z0, z1, bm, an, q;   /* used in BODNER */

/*********************************************************/
/* GCMPLOOP1(): transfers parameters and calls           */
/* corresponding function to get cell's information      */
/*********************************************************/
GCMPLOOP1(fe_arg_blk, dpu_arg_blk)
     void *fe_arg_blk[];
     DopeVector *dpu_arg_blk[];
{
  plural double sigl[30], dsigl[30]; /* subcell */
```

```
  /* para. transfer */
  para_trans(fe_arg_blk, dpu_arg_blk, dsigl, sigl);

  /* establish a cell and information associated with it on PE */
  cell_on_pe(dsigl, sigl);

  /* call constitutive model to get:
     inelastic strain rate (EPS)
     state variable rates */
  switch(micro_s.ncmd){
  case 1:
    bodner(dsigl, sigl);
    break;
  case 2:
/*    model3m(gjdsigl, gjsigl);*/
    break;
  case 3:
/*    model2m(gjdsigl, gjsigl);*/
    break;
  }

  /* cell inf. back-transfer */
  para_back_trans(fe_arg_blk, dpu_arg_blk);
}

/*********************************************************/
/* para_trans(): para. transfer from FORTRAN to MPL      */
/*********************************************************/
int para_trans(fe_par, dpu_par, dsigl, sigl)
     void *fe_par[];
     DopeVector *dpu_par[];
     plural double *dsigl, *sigl;
{
  DopeVector *Dopep;

  /* duplicate pointers */
  copyIn(&fe_par[1], &cell_s.dsigal_p, sizeof(cell_s.dsigal_p));
  copyIn(&fe_par[2], &cell_s.sigal_p, sizeof(cell_s.sigal_p));
  copyIn(&fe_par[3], &cell_s.esp_p, sizeof(cell_s.esp_p));

  copyIn(&fe_par[4], &micro_s.matnum_p, sizeof(micro_s.matnum_p));
  copyIn(&fe_par[5], &micro_s.vf_p, sizeof(micro_s.vf_p));
  copyIn(&fe_par[6], &micro_s.nm_p, sizeof(micro_s.nm_p));
  copyIn(&fe_par[7], &micro_s.ns_p, sizeof(micro_s.ns_p));
  copyIn(&fe_par[8], &micro_s.nst_p, sizeof(micro_s.nst_p));
  copyIn(&fe_par[9], &micro_s.nmt_p, sizeof(micro_s.nmt_p));
  copyIn(&fe_par[10], &micro_s.nb_p, sizeof(micro_s.nb_p));
  copyIn(&fe_par[11], &micro_s.ng_p, sizeof(micro_s.ng_p));
  copyIn(&fe_par[12], &micro_s.idp_p, sizeof(micro_s.idp_p));
  copyIn(&fe_par[13], &micro_s.nsfd_p, sizeof(micro_s.nsfd_p));
  copyIn(&fe_par[14], &micro_s.lop_p, sizeof(micro_s.lop_p));
  copyIn(&fe_par[15], &micro_s.ncmd_p, sizeof(micro_s.ncmd_p));

  copyIn(&fe_par[16], &vpt_s.nvp_p, sizeof(vpt_s.nvp_p));
  copyIn(&fe_par[17], &vpt_s.matt_p, sizeof(vpt_s.matt_p));

  copyIn(&fe_par[18], &vpv_s.vp_p, sizeof(vpv_s.vp_p));

  /* copy values */
  copyIn(cell_s.dsigal_p, cell_s.dsigal, sizeof(cell_s.dsigal));
  copyIn(cell_s.sigal_p, cell_s.sigal, sizeof(cell_s.sigal));
  copyIn(cell_s.esp_p, cell_s.esp, sizeof(cell_s.esp));

  copyIn(micro_s.matnum_p, micro_s.matnum, sizeof(micro_s.matnum));
  copyIn(micro_s.vf_p, &micro_s.vf, sizeof(micro_s.vf));
  copyIn(micro_s.nm_p, &micro_s.nm, sizeof(micro_s.nm));
```

```
  copyIn(micro_s.ns_p, &micro_s.ns, sizeof(micro_s.ns));
  copyIn(micro_s.nst_p, &micro_s.nst, sizeof(micro_s.nst));
  copyIn(micro_s.nmt_p, &micro_s.nmt, sizeof(micro_s.nmt));
  copyIn(micro_s.nb_p, &micro_s.nb, sizeof(micro_s.nb));
  copyIn(micro_s.ng_p, &micro_s.ng, sizeof(micro_s.ng));
  copyIn(micro_s.idp_p, &micro_s.idp, sizeof(micro_s.idp));
  copyIn(micro_s.nsfd_p, &micro_s.nsfd, sizeof(micro_s.nsfd));
  copyIn(micro_s.lop_p, &micro_s.lop, sizeof(micro_s.lop));
  copyIn(micro_s.ncmd_p, &micro_s.ncmd, sizeof(micro_s.ncmd));

  copyIn(vpt_s.nvp_p, vpt_s.nvp, sizeof(vpt_s.nvp));
  copyIn(vpt_s.matt_p, vpt_s.matt, sizeof(vpt_s.matt));

  copyIn(vpv_s.vp_p, vpv_s.vp, sizeof(vpv_s.vp));
}

/*********************************************************/
/* para_back_trans(): para. transfer from MPL to FORTRAN */
/*********************************************************/
int para_back_trans(fepar, dpupar)
     void *fepar[];
     DopeVector *dpupar[];
{
  DopeVector *Dopep;
}

/*********************************************************/
/* cell_on_pe(): establish a cell on PE                  */
/*********************************************************/
int cell_on_pe(dsigl, sigl)
     plural double *dsigl, *sigl;
{
  int i,j,k,n,m,s,t;

  micro_s.ns=micro_s.nb*micro_s.ng;

  for(j=0; j<micro_s.nb; j++)
    for(i=0; i<micro_s.ng; i++){
      n=micro_s.ng*j+i;          /* original NS */
      m=micro_s.matnum[i][j];        /* original NM */
      t=vpt_s.matt[m-1];               /* original IT */
      s=vpt_s.nvp[0][t-1];              /* original JS */

      proc[n].d0=vpv_s.vp[s-1];
      proc[n].z0=vpv_s.vp[s];
      proc[n].z1=vpv_s.vp[s+1];
      proc[n].bm=vpv_s.vp[s+2];
      proc[n].an=vpv_s.vp[s+3];
      proc[n].q=vpv_s.vp[s+4];

      for(k=0; k<30; k++){
        proc[n].dsigl[k]=cell_s.dsigal[30*(n+1)+k];
        proc[n].sigl[k]=cell_s.sigal[30*(n+1)+k];
      }
    }

}

/*********************************************************/
/* bodner(): Bodner-Parton viscoplastic model            */
/* 1. sa[] and dsa[] contain the "micro" quantities      */
/* 2. arrangement of dsa[] & sa[] arrays                 */
/*********************************************************/
int bodner(dsa, sa)
     plural double *dsa, *sa;
{
```

```
  plural double ss[6], s[6], r[6];
  plural double temp, aj2, sq3aj, sq2, zef, al, arg1, arg, wpd;
  plural double z0m, zd, con;
  int i, j;
  unsigned long t;

dpuTimerStart();
  /* copy stress from sa to s */
  if(iproc < micro_s.ns){
    ss[0]=sa[6];
    ss[1]=sa[7];
    ss[2]=sa[8];
    ss[3]=sa[9];
    ss[4]=sa[10];
    ss[5]=sa[11];

    /* compute deviatoric stress s in each subcell */
    temp=(ss[0]+ss[1]+ss[2])/3.;
    s[0]=ss[0]-temp;
    s[1]=ss[1]-temp;
    s[2]=ss[2]-temp;
    s[3]=ss[3];
    s[4]=ss[4];
    s[5]=ss[5];

    /* predict average plastic strain-rate in each subcell */

    aj2=0.5*(p_pow(s[0],2)+p_pow(s[1],2)+p_pow(s[2],2))+
             p_pow(s[3],2)+p_pow(s[4],2)+p_pow(s[5],2);
    sq3aj=p_sqrt(p_pow(ss[0],2)+p_pow(ss[1],2)+p_pow(ss[2],2)+
              2*(p_pow(ss[3],2)+p_pow(ss[4],2)+p_pow(ss[5],2)));
    sq2=1.414215;

    if(sq3aj == 0.0){
      for(i=0;i<6;i++)
        r[i]=0;
    }
    else{
      r[0]=ss[0]/sq3aj;
      r[1]=ss[1]/sq3aj;
      r[2]=ss[2]/sq3aj;
      r[3]=sq2*ss[3]/sq3aj;
      r[4]=sq2*ss[4]/sq3aj;
      r[5]=sq2*ss[5]/sq3aj;
    }

    /* if d0=0 then assume elastic and zero-out dsa[] and return */
    if(d0 == 0){
      for(i=0;i<30;i++)
        dsa[i]=0;
/*      return(0); */
    }
    else{
      zef=z0+q*sa[19]+
        (1-q)*(r[0]*sa[20]+r[1]*sa[21]+r[2]*sa[22]+
               r[3]*sa[23]+r[4]*sa[24]+r[5]*sa[25]);
      if(aj2 == 0.0){
        al=0.0;
      }
      else{
        arg1=p_pow(zef,2.0)/(3.0*aj2);
        if(arg1 > 1e6)
          arg1=1e6;
        con=0.5*(an+1.0)/an;
        arg=con*p_pow(arg1,an);
        if(arg > 50.)
```

```
          arg=50.0;
        al=d0/(p_exp(arg)*p_sqrt(aj2));
      }

      /* inelastic strain rate */
      dsa[12]=al*s[0];
      dsa[13]=al*s[1];
      dsa[14]=al*s[2];
      dsa[15]=2*al*s[3];
      dsa[16]=2*al*s[4];
      dsa[17]=2*al*s[5];

      /* plastic-work rate */
      wpd=s[0]*dsa[12]+s[1]*dsa[13]+s[2]*dsa[14]+
          s[3]*dsa[15]+s[4]*dsa[16]+s[5]*dsa[17];

      /* state variable rate */
      dsa[18]=wpd;
      z0m=bm/z0;
      zd=z0m*(z1-zef)*wpd;
      dsa[19]=zd;

      dsa[20]=zd*r[0];
      dsa[21]=zd*r[1];
      dsa[22]=zd*r[2];
      dsa[23]=zd*r[3];
      dsa[24]=zd*r[4];
      dsa[25]=zd*r[5];
    }
  }
t=dpuTimerTicks();
printf("time(parallel) = %f\n", t*dpuTimerConst());
for(i=0;i<5;i++){
  for(j=0;j<6;j++)
    printf("%e10.3 ", proc[3].dsa[6*i+j]);
  printf("\n");
}
  printf("\n");
}
```