J. Indian Inst. Sci., May–June 1996, **76**, 337–361.

Design, programming environment and applications of a simple low-cost message-passing multicomputer

D. NANDA KISHORE[†] AND S. K. GHOSHAL*
Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560 012, India.
email: ghoshal@cadl.iisc.ernet.in

Received on March 1, 1993; Revised on December 28, 1994 and September 12, 1995.

Abstract

A parallel computer with hypercube topology, constructed with 16 Intel 80386ATZ motherboards, is described. They are interconnected using bi-directional full-duplex first-in first-out byte-wide links. The hardware, system software and application programming methodologies are given. Example programs are listed and explained.

Keywords: Parallel processing, hardware implementation, hypercube, logic programming, many-body problem, message passing.

1. Introduction

With limitations on speed and performance in a single CPU, parallel processing has become a necessity to get the required performance for many applications. Parallel computing offers many other advantages like more memory and better quality of solution, etc[1]. With advances made in VLSI and boardlevel system integration technology, the cost of a single processing element has come down considerably over a short period of time. Hence, building parallel computing hardware has become practical. We have developed a technology to realize parallel computers which use IBM PC motherboards as processing elements. This is done because

- they are available locally;
- are relatively inexpensive;
- have an acceptable price/performance ratio;
- relevant software is widely available;
- they keep on improving in performance and capability with time.

We have developed the hardware for interconnecting motherboards or full-fledged personal computers within a short geographical distance so that an inexpensive but efficient parallel computer is developed for education and research in parallel computation. We have assembled a 16-PE machine as a prototype, developed software (both system and application) on that platform and used the machine effectively for teaching and

[†]Present address: NCORE Technology Pvt Ltd, Leo Complex, 44 & 45, Residency (Cross) Road, Bangalore 560 025, India. email;kishor@ncore.soft.net

*For correspondence.

research in parallel computation. This paper describes the architecture and programming of the machine.

Parallel programming remains difficult despite many years of research and a few implementations of parallel programming languages having been made on commercially available parallel computers. The situation is really bad in cases of imperative programming languages like Fortran which have been extended to parallel programming[3]. Thus, while designing the programming model, we have taken care to introduce as little implementation-specific features as we could. That makes teaching also easy. We also have developed a paradigm which we call *dimension-independent parallel programming* which enables one to write scalable programs without even having to recompile the programs for different target implementations of the same architecture. We demonstrate in this paper how that works.

The rest of the paper is organized as follows. Section 2 describes the design consideration of the architecture. Section 3 describes the hardware mechanism of interprocessor communication. The software that runs on this architecture is introduced in Section 4. The application programming model is developed from broad-based fundamental considerations (Section 5). After a brief survey, in Section 6, of the topological issues relevant to multicomputing, we recapitulate those properties of the hypercube that are used for user programming on this machine (Section 7). After that the actual description of the application programming environment follows on a language-by-language basis. First the assembly language implementation of the primitive operations is described in Section 8. Then the extensions made to Fortran are mentioned in Section 9 and elaborated in Appendix D. Extensions made to C are discussed in Section 10 and the routines are described in Appendix F. Description of an example application, coded in C, from the area of image processing, completes Section 10. Writing parallel applications in Pascal is discussed in Section 11 with the list of procedures added to Pascal given in Appendix G. A benchmark application program in Pascal, that exposes potential deficiencies in message-passing parallel computers, is run on our machine and described in continuation of Section 11. Prolog programming is described in Section 12 with implementation details in Appendix I. An application program in parallel Prolog completes Section 12 with program segments listed in Appendix J. We conclude the paper in Section 13.

2. Design considerations of architecture

Once we have decided to use IBM PC motherboards as processing elements, and want to support MSDOS[2] as one of the operating systems as it is the most common operating system for the IBM personal computer, we have to decide on a suitable architecture to interconnect them. We can have either of the following:

- A shared bus/memory architecture
- A message-passing architecture

About shared bus/memory, we observe the following:

- There is a limitation on the maximum number of processors.

- The size of the memory that can be configured as shared memory (necessarily by designing and implementing extra hardware) cannot be large enough (considering the IBM PC architecture and the limitation of the MSDOS operating system) for most large and practical application programs.
- The latency of the shared memory may be different from that of the local memory at each processor, requiring synchronization primitives in software and arbitration mechanisms in hardware which are difficult to design and implement for large systems.
- Loading the executable code may not place the shared object in the shared memory area as MSDOS compilers would not normally accept such directives and we do not have their source codes.

Thus we decided to have a message-passing hardware architecture and programming environment.

A good way of visualizing and representing parallel computation in a distributed memory multicomputing system is to think of many asynchronous processes which interchange intermediate data during the progress of computation. Hoare's CSP[4] (communicating sequential processes) is a good way of representing such computation. We adopt the general guidelines[5] for designing the hardware of the communication link and for programming the multicomputing system from CSP. We improve the performance of the system by relaxing the strict requirement of handshaking during each message transfer. This we do by buffering the message in a special memory which enforces a strict first-in first-out protocol in hardware. This results in permitting asynchronism between two processes and in improving performance. If the writer process comes to the point where it wants to write in CSP protocol, it has to wait until the reader process comes to its counterpart. In our FIFO protocol, if there is room in the FIFO, the writer can write and proceed. The reader, when it comes to the corresponding place, just takes the data and proceeds. So none of them have to wait for a communication to take place. Of course, if the reader comes to this place first, it has to wait and we implement that protocol to preserve the correctness of the program. So just like CSP, we require no assumption on the relative speeds of the processes in order to produce correct results. The FIFO is managed by a special-purpose on-chip controller. This relieves the CPU(s) of any overheads. The control logic of the FIFO chip is given in Fig. 1.

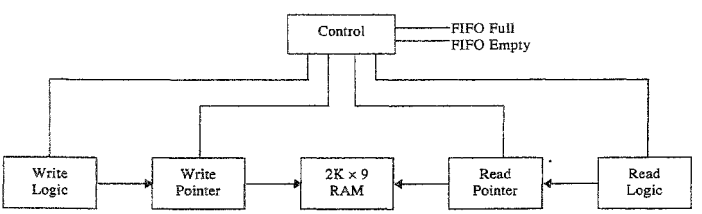

FIG. 1. IDT7203 FIFO chip.

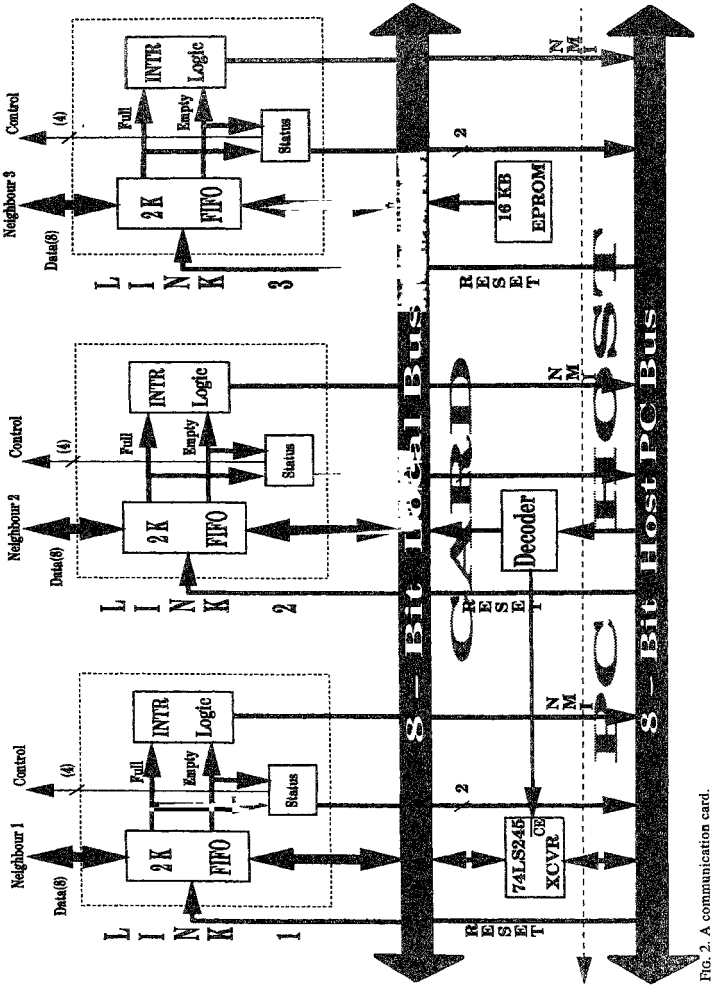

FIG. 2. A communication card.

3. Hardware architecture of communication link

PCs are linked using add-on cards (developed by us and described in Appendix A) in their empty I/O expansion slots. Each card has three bi-directional full-duplex FIFO links. The block diagram of one such card is shown in Fig. 2. One such link between two computers is shown in Fig. 3. Each link has a storage capacity of 2 kilobytes. It is built around the IDT 7203 FIFO chip[6].

The link[7] transfers unformatted streams of data over a byte-wide path between PCs using a first-in first-out queueing discipline. The link can be as long as 10 m and has been verified for proper operations at CPU clock speeds up to 40 MHz. The raw speed of the link is limited by the flow-through time of the FIFO chip and other propagation delays involved. It is estimated to be around 10 Mbytes/s. In practice, it is dictated by parallel program execution behavior and CPU speed used on the PC. The link works for different PC/XT and AT386 motherboards. Different topologies can be realized with PCs already existing in a laboratory by plugging a card into each PC and interconnecting the cards by 40-core flat ribbon cables. The architecture of the interprocessor communication mechanism provides hardware support for synchronization using instruction retry. Alternatively, it also allows polling of the links to support non-blocking communication. The physical addresses of the FIFO links range from 0B0000H to 0B0007H, which ensure multicasting between the PC and its three neighbors. The next card, if used, has an address ranging between 0B0008H and 0B000FH, and so on. The card has an EPROM to hold the low-level drivers of the links and other programs needed by the parallel processing system. This EPROM is of size 16 kbytes and has an address spanning from 0D0000H to 0D3FFFH. In case of a clash with existing devices at those addresses, the addresses of the communication card links and EPROM can be changed easily by reprogramming a PAL. The system software can be easily modified to accommodate such changes. Except the FIFO chip, all other components are easily available in the local market. The component cost of each card is about Rs 5,000. Versatile diagnostics and continuous torture-testing software of the links have been developed and will be made available. Except the FIFO chips and the EPROM, the card has only SSI components and 16L8 PALs. It is not difficult to diagnose and service the cards, in case of failure.

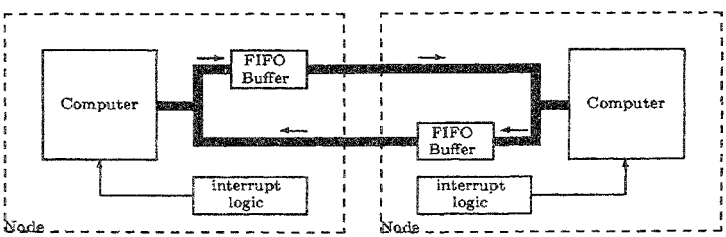

FIG. 3. FIFO link between two computers.

4. Software

The software that works on this multicomputer can be classified into two categories.

1. Operating systems. They are described in Section 4.1.
2. High-level application programming languages and the extensions made to them for supporting parallel computation. They are described in Section 4.2.

4.1. *Operating systems*

The multicomputer at present supports the MSDOS Operating system (Versions 3, 4 and 5) from Microsoft. It also runs an efficient Unix microkernel,[9] particularly suitable for parallel programming, developed by us. Using these two operating systems, only a single user can be supported at a time. Using a VM86 mode microkernel[10] developed by us, multiple users can use the machine.

4.2. *Languages available for parallel programming*

One can program the machine using any of the following high-level languages:

1. Turbo C
2. Turbo Pascal
3. Turbo Prolog
4. Microsoft Fortran

Utilities written in 80X86 assembly language and making system calls to IBM PC BIOS[11] and MSDOS[2] can also be executed in parallel.

In addition, if any language can make external subroutine calls to subroutines (written in any of the above languages or 80X86 assembly language), then programs written in such languages can also be made to execute in parallel.

Before we discuss the syntax and semantics of parallel programs on this machine, a few observations about the programming model recommended for message-passing multicomputing systems and the requirements of a parallel programming environment are in order. These are considered in the next section.

5. Programming model

Communicating sequential processes[4] is a way of representing parallel computing in progress on message-passing architectures. This model is augmented with the notion of message buffering at each link to reduce synchronization delays. The resulting model is used as a guideline to program this machine. The practical aspects of developing a parallel program on a physical platform are given in Section 5.1. Essential requirements of that platform are discussed in Section 5.2. We will see how our implementation meets these requirements in Section 8.

5.1. *Design of parallel programs*

From a parallel algorithm one arrives at a parallel program by doing the following:

- Break up the computation task into subtasks which do not have to wait for one another.
- Allocate subtasks to different processors.
- Identify intermediate results that have to be sent across links during the execution of the parallel program.
- Determine and visualize the schedule of computation and communication.

One gets the maximum performance out of the machine if he can schedule the computation and communication in such a way that no processor ever waits for intermediate results. This may not be possible for all parallel algorithms and problem sizes.

5.2. *Operations relevant to multicomputing*

Anyone having designed a parallel algorithm would like to be able to do the following on a given hardware platform on which he is to implement it:

- Create processes across links
- Terminate them after the work is done
- Find out the identity of a process
- Send a message to another process
- Receive a message from another process
- Examine the state of a process at any time

He would like to express these operations in the language familiar to him.

6. Topological issues

The way a multicomputing system is interconnected with message-passing links is called its topology. Examples of topologies are hypercubes, trees, meshes, rings, toroids, etc. The more well understood and regular a topology is the more usable it is for the purpose of application programming. The hypercube[8] is one such topology. We have used it as it is well known and has nice mathematical properties that help one develop systems algorithms for parallel computing on it and facilitate teaching. Hypercubes of different dimensions are shown in Fig. 4. The multicomputer we built is a four-dimensional hypercube.

Mask is the address of a link. It distinguishes a link from other links emanating from a computer.

In any regular topology, there is a relationship between the label of each node (we call it Nodeid in this paper), the mask of the links that emanate from that node and the Nodeids of the other end of the links. There is also a parameter that characterizes the size of the multicomputer. For hypercubes, this parameter is called the dimension.

It appears to us, in general, that on all regular topologies, for some problems, parallel programs can be written in such a way that the program will run without any modifica-

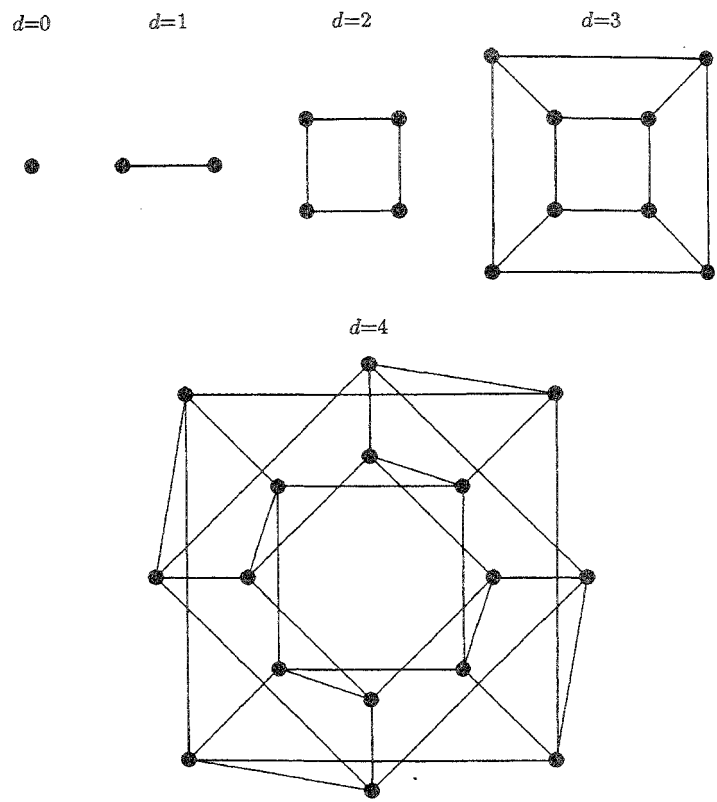

FIG. 4. Hypercubes of dimensions 0, 1, 2, 3 and 4.

tion, on any multiprocessor whose nodes are interconnected according to the rules governing that topology[14]. We have shown how to do it with hypercubes, by way of this implementation. For our machine, not even a recompilation is needed, for correct execution, when the size of the multicomputer is changed while retaining the topology. Such programs are written for our machine; we call them 'dimension-independent' parallel programs.

7. Hypercubes

We summarize the properties of a hypercube in the next two paragraphs.

An N-dimensional hypercube has 2^N computers. Each computer is called a node. Computers are labelled as binary numbers. This number is called the Nodeid. An N-dimensional hypercube needs N-bits to label each node. The nodeids range from 0 to $2^N - 1$. Two nodes are connected if and only if the Hamming distance between their labels is one. They are connected in the dimension where their labels differ in bit-position. Such nodes are directly connected and their distance is one. The mask of the link that connects these two nodes has a value equal to the difference of their nodeids. This value should be used at either end of the link. Messages transit in one 'Hop' between two directly connected nodes.

Other node pairs are not directly connected. Messages need multiple hops to transit between these pairs. This routing can be done by making system calls or the application program might prefer to do this itself. The maximum number of hops is N for an N-dimensional hypercube.

8. Primitive multicomputing operations

This section describes the firmware that implements primitive operations of multicomputing. All other high-level languages, in turn, invoke the firmware to get these operations done.

The routines that implement the primitive operations, *i.e.*, drive the hardware to create and terminate processes, send and receive messages, etc., are written in assembly language and kept in EPROMs at each node. Just as one can make BIOS system calls[11] to drive the IBM PC hardware in order to perform low-level services, one can execute the software instruction whose mnemonic is INT 60H and trap to the appropriate routine stored in the EPROM in order to get FIFO-link-related services performed. The details of the assembly language calls are given in Appendix B. Notice their correspondence with the requirements indicated in Section 5.2. How process states are examined is given in Appendix C.

A high-level language is augmented by adding some library subprograms in assembly language. These subprograms accept parameters from the high-level caller, set up the context for invoking a software trap to INT 60H and then invoke the trap. On return from the trap they go back to the caller. The only exception is TERMINATE, which as such does not return. When the next parallel program is executed, TERMINATE hands over control to the caller of the corresponding RFORK..

9. Writing Fortran application programs

The extensions made to Fortran to facilitate parallel computing are given in Appendix D. The source code of the subprograms in assembly language is in a file. This file is assembled by the implementor of parallel Fortran using Borland's Turbo assembler and the resulting object file is kept in a commonly accessible path as a library. Application programmers use the Microsoft Fortran linker utility FL to link this library with the object

code of their Fortran source. The resulting executable file is ready to run on the multicomputer.

9.1. *An example of parallel Fortran program*

Appendix E describes a parallel program to compute π which is the theme of Karp and Babb[3]. The same algorithm of Karp and Babb[3] has been used by us. One can see that our code size is small compared to most parallel Fortran implementations given in Karp and Babb[3]. It is larger than only the codes of Alliant FX/8 and Sequent Balance. It should be noted here that both these are shared memory machines with common memory bus and so their code sizes are expected to be smaller. Our code which is listed in Fig. 5 runs without recompilation on hypercubes of any dimension, including zero-dimension (*i.e.*, a sequential personal computer). Note that the code size is much smaller than that of Intel IPSC/2 which is also a hypercube architecture.

10. Writing C application programs

Appendix F describes the routines that extend C to facilitate parallel computation on our machine. These routines are implemented in C itself. Turbo C has an in-built mechanism to invoke software traps. Using that the implementor invokes INT 60H with the code for the intended operation in the AH register. All these functions which are essential for multicomputing are stored in an EPROM as described in Section 8. A header file containing the definitions of the C-callable functions is kept in the INCLUDE subdirectory. It is included by the user program while compiling. The executable file generated by Turbo C runs on our machine. We illustrate with an application described in the sub-sections that follow.

10.1. *The problem*

The problem of two-dimensional image recognition[12] can be posed as follows:

- Input: Two images A_1 and A_2.
- Output: Rotation θ, Translation $\vec{x}$, and scaling s done to A_1 so that the two images match $\theta \in \Re^1 \vec{x} \in \Re^2 s \in \Re_+$

```
      PROGRAM PARPI16
      REAL ERR,F,PI,SUM,W,PSUM; INTEGER I,INTRVL,ITIME,MYNODE
      F(X)=4.0/(1.0+X*X); PI = 4.0*ATAN(1.0); READ(*,*)INTRVL
      ITIME=ITIMER(); CALL FORKHYP()
      MYNODE = NODEID(); W=1.0/INTRVL; SUM=0.0
      DO 10 I=MYNODE+1, INTRVL, NPROC
      SUM=SUM+F((I-0.5) *W)
10    CONTINUE
      SUM = SUM*W
      CALL GATHREAL(SUM)
      ERR = SUM-PI; ITIME = ITIMER() - ITIME
      WRITE(*,*)'SUM,ERR,TIME=',SUM,ERR,ITIME
      STOP; END
```

FIG. 5. A dimension-independent parallel Fortran program to compute π.

In this study, we have reduced the problem to one of optimization. That, in turn, can be described as follows:

- Apply θ, $\vec{x}$ and s to map $A_1 \rightarrow \overline{A_1}$.
- Count the number of mismatches between A_2 and $\overline{A_1}$.
- Define this to be $f(\theta, \vec{x}, s)$
- Minimize $f(\theta, \vec{x}, s)$

10.2. *Algorithm*

The following algorithm was used:

1. Find centroids x_{A_1} and x_{A_2} of A_1 and A_2.
2. Find areas a_{A_1} and a_{A_2} of A_1 and A_2.
3. Estimate for displacement $\vec{x}$ is taken as $\vec{x}_{A_1} - \vec{x}_{A_2}$.
4. Estimate for scaling s is taken as $\sqrt{\frac{a_{A_1}}{a_{A_2}}}$.
5. Over θ optimize f at fixed $\vec{x}$ and s using simulated annealing and get $\hat{\theta}$.
6. Optimize f over a small neighbourhood of $(\hat{\theta}, \vec{x}, s)$ such that:

$$\theta \in [\hat{\theta} - 10°, \hat{\theta} + 10°]$$
$$x_1 \in [x_1 - 2, x_1 + 2]$$
$$x_2 \in [x_2 - 2, x_2 + 2]$$
$$s \in [s - 0.05, s + 0.05]$$

For optimizing, a variant of the simulated annealing algorithm[13] was used. The details of this variant are described using proper mathematical notations in Appendix K.

10.3. *Implementation*

A dimension-independent program was written in Turbo C to implement the image-recognition algorithm. The program starts executing on one of the processors with Node-id = 0. This is the master processor. The main program reads in the image file of the object to be recognized and the library object image file from the disk. Then it finds out the dimension of the hypercube on which the program is running. It computes the number of processors present in the system. Then it brings all other processors up. Each one finds out its own Nodeid. Then the computation proceeds in SPMD mode, with the matrix of the image files being partitioned by the rows among the PEs. The centroids of the two images are computed and the necessary scaling, translation and rotation are done in parallel. The number of mismatches is counted as cost and minimized. If the cost is less than a limit, then the two images are declared to be identical. After that both

Table I
Speedup on a 16-PE machine for image recognition

Number of rows (pixels)	*Image size (bytes)*	*Sequential time (seconds)*	*Parallel time (seconds)*	*Speedup (ratio)*
160	25600	1029	68	15.13
144	20736	845	54	15
128	16384	666	43	15.48
122	12544	515	33	15.606
96	9216	379	25	15.16
80	6400	264	18	14.667
64	4096	114	12	9.5

the pictures are displayed on PE_0 and the program terminates. The slaves get ready to execute another parallel program while the master returns to MSDOS.

The speedup for different problem sizes is given in Table I. The number of rows is always taken as a multiple of 16 to facilitate task-partitioning on a 16-processor machine. For small images the fork overhead dominates and so the speedup is low.

11. Parallel programming in Pascal

The procedures that extend Pascal for parallel programming are given in Appendix G. These have similar semantics to their C equivalents described earlier. Turbo Pascal can directly invoke software traps. This feature has been used to call the EPROM firmware and exchange parameters with it. The definitions and implementation of these functions and procedures are in a file that is included in the user program during compilation. The resulting executables run on our machines. We illustrate with an example that follows.

11.1. *The problem*

The many-body problem[14] can be posed as follows:

Let there be a system of N_p particles that interact *via* a pair potential. The total potential energy of a system of such particles is given by

$$U = 1/2 \sum_{i=1}^{N_p} \sum_{j=1}^{N_p} \phi(\vec{x}_i, \vec{x}_j) \tag{1}$$

where the pair potential $\phi(\vec{x}_i, \vec{x}_j)$ between two particles i and j is given by

$$\phi(\vec{x}_i, \vec{x}_j) = -\frac{Gm_i m_j}{|\vec{x}_i - \vec{x}_j|} \tag{2}$$

$\vec{x}_i$ being the position of particle i and m_i, its mass.

The problem is to compute the total potential energy of the system.

11.2. *Motivation*

In analyzing the performance of any parallel computer, two parameters are particularly important.

- t_{calc}: The typical time required to perform a generic calculation. For scientific problems, this can be taken as a floating point calculation.
  $a = b * c$
  or
  $a = b + c$
- t_{comm}: The typical time taken to communicate a single word between two nodes.

As the many-body problem requires the updated positions of each particle to be communicated to all the processors, it serves as a good benchmark to estimate t_{calc}/t_{comm} of a parallel computer. In particular, if a message-passing multiprocessor is comparatively slow in interprocessor communication, that fact comes out when one runs the many-body problem on that machine.

11.3. *Implementation*

Appendix H lists a dimension-independent parallel program for solving a many-body problem in Figs 6 and 7. Table II gives the speedup for different problem sizes. At low sizes the fork overheads dominate.

```
program parpoten;
uses Dos;
{$I$ PARPAS.INC}
const ARRAYSIZE = 1100;
(* particle description *)
type particle = record
                    a:real; b real; c:real; m:real,
                end;
type oned = array[φ..ARRAYSIZE] of particle;
const G = 6.671e-11;
(* Calculate potential energy between the two particles *)
function pot_2arr(xi,xj:particle):real;
var
   t1, t2, t3, temp1, temp2:real;
begin
   t1: = xi.a-xj a; t2: = xi.b-xj.b; t3: = xi.c-xj c;
   temp2: = sqr(t1) + sqr(t2) + sqr(t3);
   temp2: = sqrt(temp2); temp1: = x1.m*xj.m;
   pot_2arr: = -temp1/temp2;
end;
var
   p:oned;
   u,temp,pu:real;
   i,j,np,lmi,lfin,chunksize:integer;
   dim,mask,junk,myid,chunk:integer;
   time1, time2, time3, t:longint;
```

FIG. 6. A parallel program to solve the many-body problem.

Table II
Speedup on a 16-PE machine for many-body problem

Number of bodies	*Sequential time (seconds)*	*Speed up (ratio)*
16	0.38	0.034
32	2	0.1818
64	6	0.5
128	22	1.6923
256	90	5.0
320	139	6.95
384	201	8.04
480	314	9.8125
512	358	10.2286
640	558	10.7308
800	874	11.9726
864	1020	12.4878
928	1176	12.6452
1024	1440	12.8571

```
(* main *)
begin
    time1:=0; time2:=0; time3:=0; t:=0;
    write('Enter No. of particles in the system:'); read(np);
    time1:=pastime;
    (* Determine dimension of the hypercube *)
    Dimhyp(dim);
    (* Fork all *)
    forkall;
    (* Find out node id *)
    myid:=Nodeid;
    time2:=pastime;
    (*Assign Boundaries of each processor *)
    chunk:=np div numpro; lini:=chunk * myid;
    lfin:=chunk * (myid + 1): lfin:=lfin -1;
    (*Initialise particle description *)
    for i:=lini to lfin do
    begin
        p[i].a:=(i+1) *4.0/10.0; p[i].b: = (i+1)*2.0/10.0;
        p[i].c:=(i+1) *5.0/10.0; p[i].m: = (i+1)/10.0
    end;
    u:=0.0; pu:=0; chunksize:=chunk * Size of (particle);
    (* Broadcast u *)
    for i:=0 to 15 do fromany(p[t1*i],t2,i);
    (* Calculate total potential energy of the system *)
    for i:= lini to lfin do for j:= 0 to (np-1) do
        if (i<>j) then u:= u+pot_2arr(p[i],p[j];
    (* Gather the partial energy value from all
    and accumulate it in id 0 *)
    gathreal(u,pu); u:=pu*G/2.0; time3:=pastime;
    writeln('Total potential energy of the system is', u);
    if (myid <> 0) then terminate;
    t:=time3 - time1;
    writeln('Time including fork overhead', t, 'sec');
    t:=time3 - time2;
    writeln('Time excluding fork overhead', t, 'sec');
end.
```

FIG. 7. A parallel program to solve the many-body problem.

12. Application programming in parallel Prolog

The strict type-checking enforced by Prolog and the fact that Prolog has no notion of the address of a DOMAIN in the physical memory of the computer, necessitate many more predicates for parallel programming in Prolog than the minimum of subprograms needed for a typical imperative programming language. The predicates that have been added to Prolog and their implementation details are given in Appendix I. The subsections that follow describe an interesting application of parallel Prolog.

12.1. *The problem*

Let $\mathcal{N}$ be the set of natural numbers {0, 1, 2,...}, and S_i, $i = 1, 2, 3...k$ be k subsets of $\mathcal{N}$ whose elements are chosen at random. Let $\hat{S} = S_1 \times S_2 \times ... S_k$. Let $\tilde{S}$ be a subset of $\hat{S}$ defined as:

$$\tilde{S} = \left\{ (q_1, q_2, \ldots, q_k) \in \hat{S} \mid p_L(q_1, q_2, \ldots, q_k) \text{ holds} \right\} \tag{3}$$

where p_L is a k-ary Diophantine predicate[15].

We are looking for $|\tilde{S}|$, where the symbol $|\cdot|$ stands for the cardinality of a set.

12.2. *Motivation*

The implementation of this problem illustrates how a CSP-like parallelism can be efficiently supported in a logic programming system. The presence of data-parallelism within the problem has been effectively exploited while retaining the logic programming style. How to dynamically partition the data-set can also be understood. This program can be used to solve linear and nonlinear integer inequalities. Note that, in a way, what Prolog does in a domain of symbols, this problem does it in $\mathcal{N}$.

12.3. *Implementation*

The program needs no extra domain than what is already defined in Turbo Prolog. Appendix I gives the implementation details of this problem.

12.4. *Speedup*

To solve the following set of linear inequalities

$$3x_1 + 4x_2 + 5x_3 - 2x_4 = \text{RHS} \tag{4}$$

$$2x_1 - 3x_2 + 4x_3 - 6x_4 < 500 \tag{5}$$

$$x_1 + 2x_2 - 3x_3 - 3x_4 > 50 \tag{6}$$

$$-4x_1 + 5x_2^2 - x_3 + 5x_4 > 100 \tag{7}$$

a program was written in parallel Prolog.

The program was run both on a sequential machine and a 16-processor parallel machine. $|S_1|$ was taken to be 256. $|S_2| = 160$, $|S_3| = 32$, and $|S_4| = 32$.

Table III
Speedup on a 16-PE machine for Diophantine predicates

RHS (User gives)	*No. of solutions (computer replies)*	*Sequential time (seconds)*	*Parallel time (seconds)*	*Speedup (ratio)*
100	0	2596	174	14.91
200	672	2595	174	14.91
300	5198	2598	175	14.85
400	11492	2595	173	15
500	18477	2598	173	15
600	27678	2592	174	14.91
700	31893	2595	175	14.81
800	31952	2598	172	15.1
900	30499	2591	175	14.81
1000	24181	2595	175	14.83

The speedups for different RHS values are given in Table III.

13. Conclusion

We have demonstrated that it is possible to build efficient multicomputers using inexpensive technology. We have designed and built such a moderate-sized multicomputer and have programmed it in a number of programming languages. Various application programs have also been developed on this machine. The programs can be developed both in MSDOS and Unix environments. Satisfactory speedups have been obtained on this machine for different benchmark problems. The machine and the programming environment have been used successfully for teaching and research in parallel computation.

The machine has a few drawbacks. The firmware is not intelligent enough to recover from all possible types of deadlocks. Operations like routing and packetizing of messages consume enormous processor overheads which makes the machine unsuitable for fine-grained task-partitioning. Debugging is extremely difficult due to lack of proper tools. The interprocessor communication speed is low on systems with slow clocks and bus speeds. The hardware and system software are being redesigned and upgraded in order to alleviate some of these difficulties.

The machine is currently being used extensively by five people round the clock for their own research. A parallel file system particularly suitable for hypercubes has just been developed on this machine. Benchmarking is being conducted on this PFS, as of now.

Acknowledgements

We thank the KBCS project, with whose support this work was done. We thank Mr A. R. Gnanakumar, SERC, for drawing Fig. 2 and Mr A. D. Desai of Nelsis Project for proofreading the revised draft.

References

1. RAJARAMAN, V. *Elements of parallel computing*, 1990, Prentice-Hall.
2. *Microsoft MSDOS user's guide and reference*, 1991, Microsoft Press.
3. KARP, A. H. AND BABB, R. G. A comparison of 12 parallel Fortran dialects, *IEEE Software*, September, 1988, 52–67.
4. HOARE, C. A. R. *Communicating sequential processes*, 1987, Prentice-Hall.
5. GHOSHAL, S. K. *The numerical integration of ordinary differential equations on multiprocessing systems*, Ph. D. Thesis, Indian Institute of Science, Bangalore, 1988.
6. *CMOS parallel first-in/first-out FIFO*, Data sheet, Integrated Device Technology, Inc., Feb. 1986.
7. GHOSHAL, S. K., GUHA, S., ARIFF, S. M. AND RAJARAMAN, V. Simple low-cost multiprocessor based on message passing FIFO links, *Microprocessors Microsystems*, 1990, **14**, 297–300.

8. SEITZ, C. L. — The cosmic cube, *Commun. ACM*, 1985, **28**, 22–33.

9. GHOSHAL, S. K. AND KALAI SELVI, S. — MUSIX—A simple microkernel for parallel computing in Unix, *Proc Southern Regional Convention-94*, June 27–30, Hyderabad, pp. 65–71.

10. GHOSHAL, S. K., KALAI SELVI, S., YOUREN, Z. AND NAGENDRA PRASAD, P. S. — A multimode operating system for distributed computing in MSDOS, *Proc. Southern Regional Convention-94*, June 27–30, Hyderabad, pp. 19–25

11. — *PC/AT Technical reference manual*, 1985, IBM

12. SANCHETI, N. K. AND VENKATESH, Y. V. — Two-dimensional object recognition using simulated annealing, *J. Indian Inst. Sci.*, 1990, **70**, 197–212.

13. KIRKPATRICKS, S., GELATT, C. D. AND VECCHI, M. P. — Optimization by simulated annealing, *Science*, 1983, **220**, 671–679.

14. FOX, G., JOHNSON, M., LYZENGA, G., OTTO, S., SALMON, J. AND WALKER, D. — *Solving problems on concurrent processors*, Vol. I–*General techniques on regular problems*, 1988, Prentice-Hall.

15. GAREY, M. R. AND JOHNSON, D. S. — *Computers and intractability: A guide to the theory of NP-completeness*, 1975, W. H. Freeman.

16. GOLDBERG, D. — What every computer scientist should know about floating point arithmetic, *ACM Comput. Surv.*, 1991, **23**, 5–48.

Appendices

A. Details of communication card

Each card (See Fig. 2) can be sub-divided into the following blocks:

- *Communication link with FIFO storage*: There are three such links on each card. The FIFO chip is written into by the host PC. The neighbor reads it from the other end (see Fig. 3). The data, status and control signals are buffered and terminated at both ends of the flat ribbon cable transmission line whose alternate cables are grounded.
- *Decoders and local bus*: The links and the EPROM are memory mapped by the decoder implemented by PALs. The local bus is isolated from the host-PC bus by a 74LS245 transceiver which is enabled only when the card is accessed.
- *EPROM for storing firmware*: This EPROM contains the firmware described in Section 8 and Appendix B. The EPROM also captures control of the host-PC during power-on self test (POST) phase[11] and performs diagnosis and initialization of the links and the host-PC.
- *Interrupt generation logic*: Attempts to read from an empty link or write into a full link generates an NMI (non-maskable interrupt). This causes the defaulting instruction to be re-tried. Details are given in Ghoshal[7].
- *Status ports and LEDs*: The status of all the links can be ready by the host-PC from a memory mapped port. They are also displayed by a number of LEDs on the card.

B. Assembly language calls to firmware

The firmware initializes and tests the links at power up and drives the links during operation. The source code of this firmware is in a disk file which also has some documentation of the usage of these routines. This file is assembled, linked and is made a binary file which is written into an EPROM which is plugged thereafter into the respective communication card at every node when the system is assembled.

The INT 60H is invoked with different values in AH register to indicate which kind of service is being requested. The more important services are listed below:

- AH = 0 is to reset the interrupt vectors used for multicomputing.
- AH = 1 is to do a blocking write. The writer will get blocked if the FIFO is full.

- AH = 2 is to do a blocking read. The reader will get blocked if the FIFO is empty.
- AH = 5 is to spawn a process.
- AH = 6 is to terminate a process.
- AH = 7 is to find out the Nodeid.
- AH = 8 is to find out the dimension of the hypercube.

C. Examining process states

The display adaptor outputs of 16 processors are multiplexed and fed to one video monitor. The programmer can see any screen at any time by pressing a *Hot Key* which activates the firmware to drive the multiplexer. For the convenience of debugging, the digital display multiplexer has also been made controllable from high-level languages. The programmer can freely use these calls at any stage during program execution, for debugging or for any other purpose. In a specially programmed mode, the controller of the multiplexer can traverse a hypercube's spanning tree in order to track a given message in transit. That relieves the overhead from the processor and ensures that even when there is a fatal deadlock, the programmer can still see the screens of his interest. The display multiplexer controller is a special-purpose ASIC chip which is implemented on a Field Programmable Gate array.

Controlling the display multiplexer does not affect the state of any process in any processor in any way. So there is no probe effect while debugging.

D. Fortran language routines for parallel computing

The following Fortran-callable library subroutines have been added to implement operations relevant to parallel computing:

- SUBROUTINE RFORK(MASK) spawns a process in a processor at the other end of the link identified by MASK.
- SUBROUTINE TERMIN terminates a process. After this is executed, the processor is ready for another process to be spawned on it again.
- FUNCTION NODEID returns the Nodeid.
- FUNCTION NDIMHYP returns the dimension of the hypercube.
- SUBROUTINE INTSEN(MASK, INTVAL) writes an integer value INTVAL on the link identified by MASK.
- SUBROUTINE INTRCV(MASK, INTVAR) reads an integer from the FIFO link and places it in the integer variable INTVAR.
- SUBROUTINE RELSEN(MASK, RELVAL) writes a real value RELVAL on the link identified by MASK.
- SUBROUTINE RELRCV(MASK, RELVAR) reads a real number from the FIFO link and places it in the real variable RELVAR.
- SUBROUTINE DPRSEN(MASK, DPRVAL) writes a double precision value DPRVAL on the link identified by MASK.
- SUBROUTINE DPRRCV(MASK, DPRVAR) reads a double precision number from the FIFO link and places it in the double precision variable DPRVAR.
- SUBROUTINE ANYSEN(MASK, ANYADR, LEN) writes LEN bytes of data starting from ANYADR on the link identified by MASK. It can be used to send arrays. *The way Fortran orders multidimensional array elements in memory has to be kept in mind when sending subarrays of multidimensional arrays.*
- SUBROUTINE ANYRCV(MASK, ANYADR, LEN) *overwrites* LEN bytes of data starting from ANYADR with consecutive bytes read from the link identified by MASK. It can be used to receive arrays. *The way Fortran orders multidimensional array elements in memory has to be kept in mind when getting subarrays of multidimensional arrays from other processors.*
- FUNCTION ITIMER( ) returns the timer ticks elapsed since power up. This is used for timing programs.

- SUBROUTINE SHOW(NSCREEN) displays the screen of the processor whose PE_ID equals NSCREEN. See Appendix C.
- SUBROUTINE FORKHYP brings up all processing elements of a hypercube. The child threads execute the instructions following this call in parallel. Its calling syntax is CALL FORKHYP. It has no parameter.
- SUBROUTINE TERMHYP terminates all threads at nodes. The processing elements are ready to execute another parallel program. Its calling syntax is CALL TERMHYP. It has no parameter.
- FUNCTION NUMPRO returns the number of processors in the multicomputing system. Its calling syntax is:

  NPR = NUMPRO
- SUBROUTINE SCATINT scatters an integer INTPAR from PE 0 to all other processors. Its syntax is CALL SCATINT(INTPAR).
- SUBROUTINE GATHINT sums up the values of an integer parameter INTPAR at all nodes, and returns that value at node 0. Its syntax is: CALL GATHINT(INTPAR).
- SUBROUTINE SCATFLT scatters a floating point number FLTPAR from PE 0 to all other processors. Its syntax is CALL SCATFLT(FLTPAR).
- SUBROUTINE GATHREAL sums up the values of a floating point parameter FLTPAR at all nodes, and returns that value at node 0. Its syntax is:

  CALL GATHREAL(FLTPAR).
- SUBROUTINE SCATDBL scatters a double precision floating point number DBLPAR from PE 0 to all other processors. Its syntax is CALL SCATDBL(DBLPAR).
- SUBROUTINE GATHDOUB sums up the values of a floating point parameter DBLPAR at all nodes, and returns that value at node 0. Its syntax is:

  CALL GATHREAL(DBLPAR).
- SUBROUTINE SCATANY scatters any data object with name ANYVAR and occupying NUMBYT bytes of storage from PE 0 to all other processor. Its syntax is:

  CALL SCATANY(ANYVAR, NUMBYT).

```
procedure fromany(var varany; numbyt:integer; srcpe:integer);
var mask, dim, myid, indx, virtid : integer;
begin
        dimhyp(dim);
        myid := nodeid;
        virtid := (nodeid xor srcpe);
        mask :=1;
        for indx:= 1 to dim do
            begin
if (virtid>=mask) then
            begin
if (virtid < (mask shl 1)) then
            begin
              bread(mask,varany,numbyt);
            end;
        end
      else
        begin
            bwrite(mask,varany,numbyt);
        end;
      mask := mask shl 1;
    end;
end;
(* fromany *)
```

FIG. 8. A routine to scatter bytes from any processor.

- SUBROUTINE FROMANY scatters any data object with name ANYVAR and occupying NUMBYT bytes of storage from PE with Nodeid equal to NSRCPE to all other processors. It uses an algorithm which is developed by us and is explained in Fig. 8. Its syntax is:
  CALL FROMANY(ANYVAR, NUMBYT, NSRCPE).

E. An example of application program in parallel Fortran

The program listed in Fig. 5 reads the number of partitions (called INTRVL in the program) that the interval of integration ranging between [0, 1] is to be divided into. Then it spawns a thread in each available processor. The processors handle sub-intervals in a round-robbin fashion. The partial sums are gathered from each processor and summed at PE 0 to yield the value of π. This value and the error incurred are printed and the program terminates.

As INTRVL increases, the speedup approaches the number of processors present asymptotically. The error decreases with increasing INTRVL initially, but eventually gets dominated by the cumulative roundoff error.

F. Routines in C for parallel computing

The following functions have been implemented to facilitate parallel programming:

1. int fork(int mask) spawns a process in a processor at the other end of the link identified by mask.
2. void terminate(void) terminates a process. After this is executed, the processor is ready for another process to be spawned on it again.
3. int nodeid(void) returns the Nodeid.
4. void dimhyp(int *dim) assigns to dim the dimension of the hypercube.
5. void bwrite(int mask, char far *addr, int size) writes size bytes of data starting from addr on the link identified by mask.
6. void bread(int mask, char far *addr, int size) *overwrites* size bytes of data starting from addr with consecutive bytes read from the link identified by mask.
7. int timetick(void) returns the time-ticks elapsed since power-up. It is used for measuring the time taken by different portions of programs.
8. void scatint(int in, int *out) scatters an integer value, originally in PE_0, to the integer variable out in all the processors.
9. void gathint(int psum, int *sum) gathers an integer from values psum at all processors and sums it up in sum.
10. void sci(int *out) scatters the integer variable out from PE_0 to all other processors.
11. void gpi(int *sum) gathers sum overwriting its previous value at all processors.
12. void gathdoub(double psum, double *sum) gathers an IEEE754[16] double-precision floating point number from values psum at all processors and sums it up in sum.
13. void gpr(float *sum) gathers sum overwriting previous value.
14. void gprd(double *sum) gathers as IEEE754[-16] double a precision floating point number sum overwriting previous values.
15. void scatany(int *varany, int numbyt) scatters *anything* that begins at address varany at PE_0 and occupies numbyt bytes of storage into all other processors.
16. void fromany(char far *value, int numbyt, int srcpe); scatters *anything* occupying numbyt bytes of storage from any processing element with nodeid = srcpe to all other processors. It uses an algorithm which is developed by us and is explained in Fig. 8.
17. void show(int scrnum); displays the screen of the processor whose nodeid equals scrnum by controlling the display multiplexer. See Appendix C.

G. Extensions to Pascal for parallel computing

The following procedures and functions have been added to Turbo Pascal:

- function Fork(Mask; integer):integer;
- procedure Terminate;

- function Nodeid:integer;
- procedure Dimhyp(var d:integer);
- procedure Bwrite(Mask:integer; var buffer; len:integer);
- procedure Bread(Mask:integer; var buffer; len:integer);
- procedure forkall; brings up all the available processors in the hypercube.
- procedure gathint(inint: integer; varoutint:integer); gathers an integer from values inint at all processors and sums it in outint.
- procedure gpi(var varint:integer); gathers varint overwriting the previous value.
- procedure gathreal (inreal:real; var outreal:real); gathers a real from values inreal at all processors and sums it in outreal.
- procedure gpr(var varreal:real);
- function numpro:integer; returns the number of available processors.
- procedure scatint(inint: integer; var outint:integer); scatters an integer value inint originally in PE_0 to the integer variable outint to all processors.
- procedure sci (var varint:integer); scatters the integer variable varint from PE_0 to all other processors.
- procedure scatany(var varany;numbyt:integer); scatters *anything* that begins at address varany in PE_0 and occupies numbyt bytes of storage into all other processors.
- procedure fromany(var varany; numbyt:integer; srcpe:integer); scatters *anything* occupying numbyt of storage from any processing element with Nodeid = srcpe to all other processors.
- procedure show (numscr:integer) shows the screen of the processor whose pe_id equals numscr. See Appendix C.

The implementation of fromany is given in Fig. 8. It is important to understand how it works Note that the hypercube is a symmetric topology. Any PE can be labelled with a *Nodeid* = 0. Once that is done, the *Nodeid* of all other PEs becomes fixed. So any parallel program object code will execute correctly on such a relabelled hypercube. So a scatter routine will work, provided the Nodeids used in comparison are as per the relabelling scheme. The scheme is relative to the srcpe being zero. fromany is crucial to *dimension-independent programming*. Section 11 and Appendix H describe an application of fromany.

H. Code of many-body problem

Figure 6 has the declarations and the Pascal function computing the potential energy between two particles. Figure 7 contains the main program.

I. Implementation of parallel Prolog

The predicates are implemented partly in assembly language (the source code is kept in PARPRO.ASM and is assembled by the Turbo assembler) and partly in C (the source code is kept in CPRO.C and is compiled by the Turbo C compiler). The executable module is linked by the Turbo Prolog linker, according to the directives given in the .PRJ file by the user. The linker can be run from the Turbo Prolog integrated environment. Alternatively, the TLINK utility can be used to link the modules. TLINK is invoked from the command line.

The following global predicates have been added to Turbo Prolog.

- FORK(MASK) spawns a parallel process down MASK. Thereafter, both the parent and child processes evaluate the predicates on the right-hand side of FORK from left to right as in normal Prolog.
- NODEID(PEID) returns the Nodeid in PEID.
- DIMHYP(DIMENSION) returns the dimension of the hypercube in DIMENSION.
- TERMINATE(PEID) terminates a process. It has to be supplied PEID, which is the Nodeid.
- TIMER(TICKS) returns the time elapsed since power-up in TICKS.
- INTSEN(MASK, INTEGER) sends an integer.

```
predicates
diaphanous(integer,integerlist,integerlist,integerlist,integerlist)
in_list   (integer, integerlist)
eq_1(integer, integer, integer, integer)
eq_2(integer, integer, integer, integer)
eq_3(integer, integer, integer, integer)
eq_4(integer, integer, integer, integer)
poly-true (integer,integer, integer, integer, integer)
writelist(integerlist)
make-list(integer,integerlist)
make-part(integer,integer,integerlist)
cartprod(integer,integerlist,integerlist,integerlist,integerlist)
query(integerlist,integerlist,integerlist,integerlist)
further(integer,integerlist,integerlist,integerlist,integerlist)
```

FIG. 9. The predicates used.

- INTRCV(MASK, INTEGER) gets an integer.
- CHRSEN(MASK, CHRVAL) sends a character.
- CHRRCV(MASK, CHRVAR) gets a character.
- RELSEN(MASK, RELVAL) sends a real value.
- RELRCV(MASK, RELVAR) gets a real variable.
- STRSEN(MASK, STRVAL) sends a string
- STRRCV(MASK, STRVAR) gets a string.
- SYMSEN(MASK, SYMVAL) sends a symbol.
- SYMRCV(MASK, SYMVAR) gets a symbol.
- INLSEN(MASK, INLVAL, LENGTH) sends a list of integers.
- INLRCV(MASK, INLVAR, LENGTH) gets a list of integers.
- CHLSEN(MASK, CHLVAL, LENGTH) sends a list of characters.
- CHLRCV(MASK, CHLVAR, LENGTH) gets a list of characters.
- RLLSEN(MASK, RLLVAL, LENGTH) sends a list of reals.
- RLLRCV(MASK, RLLVAR, LENGTH) gets a list of reals.
- STLSEN(MASK, STLVAL, LENGTH) sends a list of strings.
- STLRCV(MASK, STLVAR, LENGTH) gets a list of strings.
- SMLSEN(MASK, SMLVAL, LENGTH) sends a list of symbols.
- SMLRCV(MASK, SMLVAR, LENGTH) gets a list of symbols.
- INITCOUNT(INTEGER) initializes an internal counter.
- TERMCOUNT(INTEGER) returns the final value of the counter.
- INCRCOUNT(INTEGER) increments the counter.
- ISUBLIST(integerlist, STILST, LNILST, integerlist) returns a sublist of an integer list.
- SCATINT(INTEGER, INTEGER) scatters an integer from PE_0.
- GATHINT(INTEGER, INTEGER) gathers an integer.
- TEMPLATE(INTEGER, INTEGER) is a C program that just copies the input parameter into the output parameter. What is easy in C is hard in Prolog and *vice versa*. So at times when something that can easily be done in C needs to be done in Prolog, one can use this procedure, after replacing the body with his own code. This allows one to continue programming the Prolog application without having to bother with parameter-passing conventions between C and Prolog. Of course, after any change, CPRO.C has to be recompiled by running the Turbo C compiler.

```
clauses
poly_true(RHS,X1, X2, X3) :-
eq_1(RHS, X1, X2, X3, X4),
eq_2(X1, X2, X3, X4),
eq_3(X1, X2, X3, X4),
eq_4(X1, X2, X3, X4).
eq_1(RHS, X1, X2, X3, X4) :-
3*X1 + 4*X2 + 5*X3 -2*X4 = RHS
eq_2(X1, X2, X3, X4) :-
2*X1 -3*X2 + 4*X3 -6*X4 <500.
eq_3(X1, X2, X3, X4) :-
X1 + 2*X2 - 3*X3 - 3*X4 >50.
eq_4(X1, X2, X3, X4) :-
-4*X1 + 5*X2*X2 - X3 + 5*X4 >100.
in_list(Int, [IntL]).
in_list(Int, [_|Tail]) :- in_list(Int, Tail).

diaphanous(RHS, List1, List2, List3, List4) :-
in_list(X1, List1), in_list(X2, List2),
in_list(X3, List3), in_list(X4, List4),
poly_true(RHS,X1, X2, X3, X4).

writelist([]) :- !.
writelist([H|T]) :- write("elem=",H,"\n" ), writelist(T).
```

FIG. 10. Some clauses used in this problem.

- FORKALL Brings up all the available processors in the hypercube.
- NPROCS(INTEGER) returns the number of processors.
- SHOW(INTEGER) controls the display multiplexer to show a particular processor's screen. The Nodeid of that processor is given as an input integer parameter

In Prolog one has to be particular about the direction of parameter passing between the predicates. A parallel Prolog program must be executed *only* from the command line. In order to compile into an .EXE file[2], a program *must* have a goal.

J. Implementation of Diophantine predicates

Figure 9 shows the predicates used for this problem. Figures 10–12 show the clauses. Figure 13 shows the goal of the parallel program. The program is dimension-independent. It runs on a hypercube of any dimension, including zero (*i e* , a uniprocessor).

```
/* makes a list of numbers where each list
      contains N numbers */
make_list(0,[]) :- !.
make_list(N,[X1|T1]) :-N1 = N - 1, X1=N,
make_list(N1,T1),!.
/* makes 1 list of numbers. The cardinality is N div NPROCS */
make_part(_, X, []) :- X<0, !.
make_part(_, 0, []) :- !.
make_part(NPROC, N, [X1|T1]) :- N1 = N - NPROC, X1=N,
make_part(NPROC, N1, T1), !.
/* enumerates the Cartesian product set */
cartprod(L,S1,S2,S3,S4):- diaphanous(L,S1,S2,S3,S4),
            _increount(1), fail
```

FIG. 11. Note the task-partitioning.

```
/* prompts the user for query */
query(0,S1,S2,S3,S4):- write("Specify the rhs:"), readint(L),
_scatint(L, LOUT), further(0,L,S1,S2,S3,S4).
query(P,S1,S2,S3,S4):- L=0, _scatint(L, LOUT),
further(P, LOUT, S1, S2, S3, S4).
/* determines whether more queries are to be solicited */
further(N,_,_,_,_):- N<0,clearwindow, _terminate(0),!.
further(N,L1,L2,L3,L4):- timer(T1), _tods(NS1), _initcount(0),
not(cartprod(N,L1,L2,L3,L4)), _termcount(Z),
_gathint(Z, ZOUT), _tods(NS2), _timer(T2), NS = NS2 - NS1,
T = T2-T1, not(writeans(P, ZOUT, T)), query(P,L1,L2,L3,L4).
writeans(P,Z,T) :- P=0,
write("No. of Solutions=",Z,", Time=",T),nl, fail.
```

FIG. 12. Note the handling of input-output.

K. Simulated annealing algorithm

The original algorithm is described in Kirkpatrick *et al.*[13] This is a variant of the algorithm, specially developed for two-dimensional image recognition.

- Aim: To minimize $f(\vec{x},\vec{y})$ where $\vec{x} \in \Re^m$ varies continuously and $\vec{y} \in Q^k$ assumes discrete values.
- Given: $\vec{x}_0, \vec{y}_0$ as initial values, T_0 as initial temperature, T_f as final temperature, β as the temperature controlling factor, and Δx, Δy the typical neighbourhood size.

Method:

1. Set $T = T_0$, $\vec{x} = \vec{x}_0$, $\vec{y} = \vec{y}_0$, $f_1 = f(x_0, y_0)$.
2. Generate $(m + k)$ uniformly distributed random numbers $u_1, u_2, \ldots, u_{m+k} \in [-0.5, 0.5]$
3. Compute the Euclidean norm:
   $$\eta = \|u\| = \sqrt{\sum_{i=1}^{m+k} u_i^2}$$
4. Normalize the random numbers:
   $\forall i, i \in [1,m]$, Make $u_i = u_i/\eta$
   and
   $\forall i, i \in [1,k]$, Make $u_{i+m} = u_{i+m}/\eta$
5. Perturb $\vec{x}$ and $\vec{y}$:
   $\forall i, i \in [1,m]$, Make $\hat{x}_i = x_i + u_i * \Delta x_i$
   and
   $\forall i, i \in [1,k]$, Make $\hat{y}_i = \lceil y_i + u_{i+m} * \Delta y_i \rceil$
6. Compute $\hat{f} = f(\hat{x},\hat{y}), \Delta f = \hat{f} - f$.

```
goal
  _forkall, _NODEID(NID),
_NPROCS(NPR), CARD1 = 256, CARD2 = 160, CARD3 = 64, CARD4 = 64,
STARTVAL = CARD1 + NID
make_part(NPR,STARTVAL,S1),
make_list(CARD2,S2), make_list(CARD3,S3),
make_list(CARD4, S4), query(NID,S1,S2,S3,S4).
```

FIG. 13. Goal of parallel program.

7. If $\Delta f < 0$ then $x = \hat{x}$, $y = \hat{y}$ (Accept),
   Goto Step 9
   else set $p_{acc} = \exp(-\Delta f/T)$.
8. Generate $r \in u[0,1]$.
   If $r \leq p_{acc}$ then $x = \hat{x}, y = \hat{y}$ (Accept).
9. Make $T = \beta T$ If $T < T_f$ then Stop.
10. Goto Step 2.