The DATIS-P Parallel Machine

W. Paul and D. Scheerer.*

Institute for computer architecture and parallelism
University of Saarland, D-6600 Saarbrücken, Germany

Abstract: This paper summarizes results of a project which was conducted over the last 3 years at the University of Saarbrücken. Goal of the project was to construct a fault tolerant parallel machine for scientific applications and for running APL programs. The machine is supposed to tolerate failures of single slave processors, single disks and single routing chips in the interconnection network. After failure the machine reconfigures itself and continues work at the original speed.

The interconnection network is a 3 level Benes network capable of supporting 256 processors. The network has 8+1 bit planes. Bit planes are built out of semi custom 16x16 routing chips which can be run in pipelined or transparent mode. The machine has presently 33 processors (68020 + 68881 coprocessor + 1 Mbyte SRAM) and 9 hard disks.

The machine can be programmed in an extension of PASCAL which allows the user to define virtual networks and in APL. First performance measurements show good speedups on numerical problems and on most APL functions but only moderate speedup on some APL functions (like reduction) due to network bandwidth. An interface between the VME bus and the German high bandwidth wide area network VBN (140 Mbit/s) was constructed as part of the project. On the CeBit 90 exhibition in Hannover a graphics terminal was connected via the VBN network to the DATIS-P machine in Saarbrücken.

1 Two basic paradigms

DATIS-P is a MIMD machine with distributed memory and with up to 256 processors interconnected by a 256-permutation network [1]. A p-permutation network N has p inputs and p outputs. The switches of the network can be configured in such a way that input i is routed to output $\pi(i)$ and these paths are disjoint. This can be done for any permutation π of $\{1, \ldots, p\}$.

DATIS-P is designed to run programs of the following particular kind.

1.1.a) The communication pattern of processors is known at compile time and every processor communicates with at most d other processors.

1..1.b) Similar to 1.1.a) but with changes of communication patterns that are rare compared to overall computation time (due to dynamic load balancing say).

1.2 The programs have serial and parallel portions. The parallel portions proceed in rounds. There are two kinds of rounds:

compute: all processors compute autonomously until the last one is done.

communicate: each processor exchanges arrays of data through the network with all its neighbors.

Because the edges of each graph G = (V, E) of degree d can be decomposed into d classes E_1, ..., E_d such that for each i the edges in E_i define a partial permutation of V [2], [3] it is clear that these programs can be supported well by a permutation network: realize each communicate round by d subrounds, one for each class of edges.

This very simple mechanism was used before in several special purpose machines including one for molecular dynamics simulations which the first author helped to design and build from 1984 to 1986 at the IBM research lab at Almaden [4]. Although the machine was special purpose the design team did not find an example of a numerically intensive application where a synchronization mechanism more powerful than 1.2 would have brought any speed up. Finding such a numerically intensive application was identified as a (probably easy) problem and the statement that no such problem exists was (jokingly) named *Auerbachs thesis* after the groups manager.

* This work was partially supported by DFG, SFB 124 and the Leibniz program

2 Goals of the Project

In 1987 the DATIS-P parallel machine project was started at the university of Saarbrücken. The machine was to be suited for general numerical applications. The design of the machine was to be scalable up to 1024 processors. Because we expected reliability in machines with many processors to become a serious issue the machine was to be able to handle failures of single processors and of single chips in the communication network very gracefully. Programming was to be done in an extension of a conventional imperative programming language.

3 Early Design Decisions

Partly in order to facilitate debugging, some manufacturers and developers of parallel machines tended to provide powerful asynchronous communication mechanism [5], [6] although the price for this comfort was high both in terms of development time and in terms of run time of the software protocol which establishes communication between processors. We had talked with several groups which designed such machines in search of counterexamples to what we called Auerbach's thesis but with no success. Because the overall system design promised to be very simple we decided to implement the *mechanism of section 1 as the sole parallel mechanism* of the machine.

Because the machine was to be designed and programmed entirely by students it was decided not to push the issue of high processor performance. Node processors were to be just powerful enough to exercise the communication network. For the same reason it was decided to equip processors with *static* memory. We decided in 1987 for a state of the art microprocessor (MC68020), an off the shelf arithmetic coprocessor (MC68881, 0.1 MFLOP/s) and 1 Mbyte of RAM per processor for reasons of board area.

In [7] it is shown how to achieve in permutation networks simultaneously low component count and good properties of fault tolerance. This construction is based on Benes networks [8]. Therefore a Benes network was chosen as communication network.

One particular processor (called *master*) is set apart for the purposes of running a comfortable operating system, serving peripheral devices, running serial portions of programs, controlling the communication network and synchronising rounds between the remaining processors (called *slaves*).

The operating system MIRAGE [9] was chosen over UNIX because it is extremely fast.

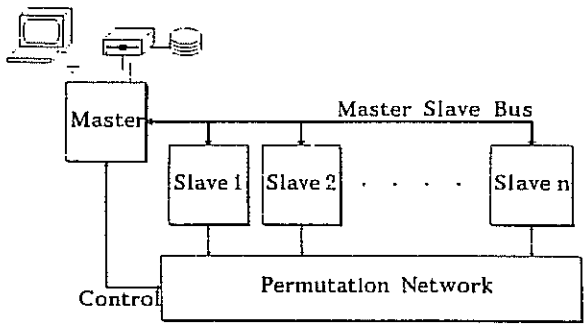

Figure 1

The classical language for programming numerically intensive applications is FORTRAN. However it was decided to base programming on an extension of PASCAL, simply because it is more popular among students of computer science. Finally it was decided to implement the language APL [10], [11] because it matches the machine well and because it permits to exploit parallelism without requiring any extensions at all.

Overview: Section 4 describes the hardware of the machine with emphasis on the features which support fault tolerance. Section 5 describes our extension of PASCAL and section 6 the parallel implementation of APL. Section 7 reports the results of a study of numerical algorithms concerning the generality of the basic paradigms and performance measurements of some applications. Section 8 sketches the connection to the VBN network. Section 9 summarizes the results and the lessons learned from this project.

4 Hardware

4.1 Communication between master and slaves is done by making the memories of the slaves shared between the master and the slaves. This is very comfortable for debugging purposes. Besides reading from and writing to the memories of each particular slave the master can perform the operations *broadcast write* (D,j) (write D to address j of each slave) and *brodcast read (i,j)* (write to address j of the master the OR of the contents of addresses i of all slaves). Both commands are directly supported by hardware. The latter one is e.g. used for detecting the end of compute rounds.

4.2 Handling slave failures is very easy on DATIS-P. One keeps a certain fraction of the processors as spare processors on the network. During long computations checkpoints are set periodically. At these checkpoints enough data from each processor are saved such that a warm start from the checkpoint is possible. If a

processor P_i fails the computation is rolled back until the last checkpoint. Some spare processor P_j takes over the work of P_i. For all permutations π involved and all processor numbers x set $\pi(x) = j$ if it was i, otherwise leave $\pi(x)$ unchanged. The setting of the switches of the network is computed for the new permutations. After this the computation can proceed *at the original speed.*

Observe that as long as no faults occur the spare processors are wasted. Thus it is tempting to use all processors in regular operation. As soon as one processor P_i fails *distribute* its work over the remaining processors or *add* it to the work of some processor P_j.

If processors cooperate on a single task, then distributing work in a balanced way requires detailed knowledge of the structure of the program. For certain cases this can be done, e.g. if one uses domain decomposition one could repartition the domain among the remaining processors. A general method for this however seems to be very hard to find.

Giving the work of one processor to a spare processor or adding it to the work of a busy one can be done in a uniform way with very little knowledge of the programs structure. But if work before failure was perfectly balanced, i.e. processors finished their computation rounds simultaneously, then the processor which now does the work previously done by two *will slow down the whole computation by a factor of two.*

In order to speed up the saving and restoring of data checkpoints every group of 4 processors share a hard disk which can be controlled by each of the processors via a common SCSI bus [12].

4.3 Failure of the master. No attempt was made to handle failures of the master. Thus if we are trying to make the machine tolerant against *all* failures of single processors or if an adversary is allowed to choose the processors which fail, then nothing is gained. But suppose processors fail randomly and independently of each other. We consider the conditional probability of a failure of the whole machine given that a single processor is faulty. If we would have no spare slaves, then this probability would be 1, i.e. failure of *any* processor would shut down the machine. With spare slaves only failure of the master shuts down the machine. The probability that the master is at fault given one processor is faulty is only $1/(p+1)$. By this factor the probability of a breakdown of the whole machine given one processor is faulty has decreased by the introduction of spare slaves.

4.4 Benes networks. Let p be divisible by c. Figure 2 shows the classical construction of p permutation networks from c-permutation networks and p/c permutation networks. If one realizes c-permutation networks as crossbar chips and $p = c^2$ one gets a 3 level network. Commercially available crossbar chips would have supported the normal mode of operation but not the fault tolerant mode which will be described below. Therefore an appropriate chip was designed. Funding was insufficient for c = 32 but sufficient for c = 16. This allowed to build a 3 level network for up to 256 processors.

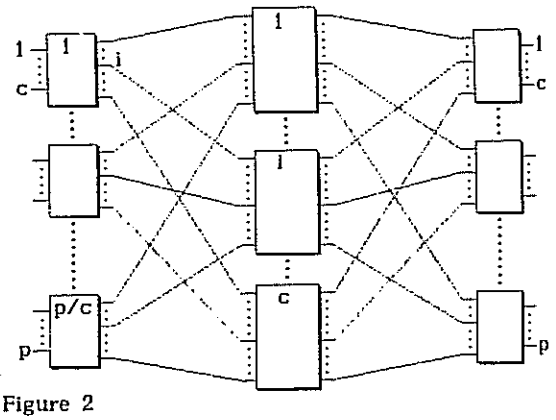

Figure 2

Data paths in the network are 9 bits wide (1 byte data + 1 parity bit). Wider data paths would have been desirable but this was beyond our budget.

Processors are partitioned into groups of 16 processors in such a way that processors in the same group are connected to the same network chips at levels 1 and 3. Each group of processors is housed in a box together with two boards which contain levels 1 and 3 of the network for this box. Again for budget reasons only two of these boxes have been build, thus the machine has presently 32 slave processors. Picture 1 shows one box and the level 2 network boards.

Level 2 of the network is realized on 9 boards each of which realizes 1 bit plane of level 2. These boards came out surprisingly large (14 x 14.5 inch) although they have 6 signal planes and each of them contains only 16 network chips and a few drivers. This seems to be an inherent problem: a lower bound from theoretical computer science asserts that laying out a hypercube with n nodes in a 2-dimensional technology (like circuit boards) requires area $\Omega(n^2/(\log n)^2)$ [13], [14].
The underlying graphs of Benes networks are very similar to hypercubes. Picture 2 is a good illustration of this; the constant in the Ω-notation is apparently not negligible.

4.5 Faults of network chips. Efficient routing algorithms for networks of this kind can be found in [15]. The algorithms work for any c but they are faster if c is a power of 2. They even work if one of the c-permutation networks in level 1 or 3 gets stuck in any fixed permutation. Following [7] this suggests a very simple way to deal with failures in levels 1 and 3: add to any crossbar in levels 1 and 3 a multiplexer whose inputs are the inputs of the crossbar and the outputs of the crossbar (see figure 3).

Because the multiplexer has very few gates compared to the crossbar the probability of failure should be much lower for the multiplexer than for the crossbar. In regular operation select the outputs of the crossbar as inputs for the multiplexer. If a failure of the crossbar is suspected select the inputs of the crossbar. The arrangement crossbar/ multiplexer still computes the identity permutation, thus the whole network is still capable of routing any permutation.

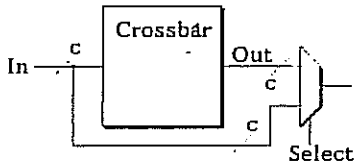

Figure 3

Faults at level 2 are handled in a different way. In order to handle processor faults we keep a certain fraction of all processors as spare processors anyway. We partition processors in groups of 16 in such a way that processors in the same group are connected to the same chips at levels 1 and 3. During regular operations we use only 15 out of every group of 16 processors. For routing one can nevertheless use the fast routing algorithms, because a full 16^2-permutation network is available. Only in case a chip at level 2 fails one uses the slower routing algorithms to compute routings which make only use of a 15^2-permutation network. These routings require only 15 chips at level 2. Thus they can avoid any failed chip at level 2.

4.6 Setting the switches in the network chips is done by a particularly simple kind of wormhole routing [16]. It occurs at the beginning of each subround of a communicate round. Routing information is held in each chip in 16 address registers A_i. During transmission of data (as opposed to setting of the address bits) data are routed from input i to output $i + A_i \bmod 16$ ($0 \le i \le 15$). This particular addressing makes testing easy after reset, which makes all registers $A_i = 0$.

Activation of a control signal s of a chip changes data transmission mode into the mode of updating the address registers. If s is activated then for one $j \in \{0, ..., 3\}$ and for all i the signal on input i is clocked into bit j of register A_i. The number j is specified via two address pins B<0:1> of the chip.

Changing the routing of the whole network takes 12 cycles. During the first 4 cycles signal s is activated for all chips in level 1 (and j assumes the 4 values from 0 to 3). During the second 4 cycles s is deactivated for the chips in level 1 and activated for the chips in level 2. During the last 4 cycles s is deactivated for the chips in levels 1 and 2 and activated for the chips in level 3. After this signal s is deactivated for all pins and data transmission proceeds.

4.7 Putting a fence around faulty processors. Wormhole routing allows each processor which is up and running to generate the routing information for the path on which it wants to send its data. In order to speed up the time for routing all address registers of each chip are updated simultaneously under the control of very few pins. This has the undesirable side effect that routing information is received from *all* processors, even those which are known to be malfunctioning. In particular it is possible that faulty processors route their data to the same output as some functioning processor. During data transmission time the data from these two processors are ORed together at the outputs of the network chips. If one can make shure that during data transmission time only zeros are sent into the network by processors which are known to be faulty then no harm will be done. This is achieved by disabling (under the control of the master) the network drivers of processors which are known to be faulty, using pull*up* resistors on the outputs of these drivers and by using *active low* logic on the ORs involved.

4.8 Clocking. The whole machine is run synchronously from a central 16 MHz clock. 300 copies of this clock are generated on a fanout board by means of a 2 level driver tree consisting of AS 1004 drivers and then distributed via 50 Ω coax cables. The root of the driver tree consists of 2 driver chips. All 12 outputs of these chips are wired together. The power and ground planes of this board are particularly thick (70 µm). In addition bus bars are used. The layout of the board is inspired more by plumbing than by the theory of antennas [17]. It works very well. Measured clock skew between any 2 signals is at most 1.5 ns [18].

4.9 Hardware performance. The following data have been measured: changing the routing of the network for a new permutation: 1.5 µs. Transmitting every byte: 1.25 µs. This amounts to a bandwidth of 800 Kbyte/s. This bandwidth is available to each processor even if all processors send and receive simultaneously. By a few changes in hardware this bandwidth could be increased to 16 Mbyte/s

available to each processor: adding DMA-controllers would give a factor of 2, making the network 4 times wider would of course give a factor of 4. Running the network in pipelined mode rather than transparent mode (the network chips have input and output latches) in combination with DMA-controllers would give a factor of 5.

Making a backup of 4 slave memories each of 1 Mbyte takes time 15 sec. due to the bandwidth of the SCSI busses.

5 PAPASA - PArallel PAscal SAarbrücken

The user defines at compile time a virtual network of processors. The compiler maps the virtual network on the hardware of the parallel machine.

5.1 Description of the language extension. The constructs which are described in this section can be used to extend any imperative programming language.

5.1.1 Extensions of the declaration part of programs. The declaration part of the main program has three parts. The usual declaration part is followed by a *network declaration* and a *declaration part for the parallel portion* of the program.

Net declarations specify directed graphs. The nodes of the graphs are indices for slave processors. The edges specify unidirectional communication lines between processors. The format of network definitions is

net NAME node specific. part edge specific. part.

A node specification part specifies a set of indices for an r-dimensional array of processors, where r ≥ 1. It has the format

$$[\ l_1 : u_1\ ,\ \ldots\ ,\ l_r : u_r\ ].$$

For each i l_i and u_i are integers. They specify the range of the i'th coordinate, namely $[\ l_i\ ,\ \ldots\ ,\ u_i\ ]$.

The set of edges is partitioned into *directions* d such that for each node n and direction d there is at most one edge in dimension d which enters n and at most one edge in direction d which leaves n. Edge specification parts with m directions therefore have the format

with S_1 . .... S_m **end**

where for each d S_d specifies the edges in direction d. This can be done in two ways:

i) direct enumeration. In this case a link from processor with index f to processor with index t is simply specified as (f, t). The specification of direction d with s edges has the format

d: $((f_1\ .\ t_1\ ).\ \ldots.\ (f_s\ .\ t_s\ ))$.

example 1: net line [1 : 6] **with**
right: ((1, 2), (2, 3), (3, 4), (4, 5), (5, 6));
left: ((2, 1), (3, 2), (4, 3), (5, 4) , (6, 5))
end

Unless the number of edges is very small this format is cumbersome.
ii) the second format to specify the edges in direction d is

d: **from** $[v_1, \ldots, v_r]$ **to**
$[expression_1(v_1, \ldots, v_r), \ldots, expression_r(v_1, \ldots, v_r)]$.

This is the declaration of a function. It has r formal parameters v_1 , ... v_r and no global parameters. The formal parameters are the coordinates of processor indices. For each processor with index f this function is (implicitely) called with f as formal parameter. If for all i $expression_i$ (f) is defined and in the range of processor indices, then the edge

(f, ($expression_1$ (f), ... $expression_r$ (f))

is included in direction d.

example 2: net grid [1 : 16; 1 : 16]

with
north: **from** [x, y] **to** [x, y+1];
south: **from** [x, y] **to** [x, y-1];
east: **from** [x, y] **to** [x+1, y];
west : **from** [x, y] **to** [x-1, y]
end

In the case r = 1 certain brackets can be omitted:

example 3: net ring [1 : 100]

with
right: **from** x **to** (x **mod** 100) + 1
end

The declaration part for the parallel part of the program begins with the keyword **parvar**. Following this keyword the usual PASCAL declarations are possible. An instance of each object declared is created on every slave processor.

Objects declared in the sequential resp. parallel part of the program are not visible in the parallel resp. sequential part of the program. Exchange of values between the sequential and the parallel part is done explicitely. The mechanisms for this are explained in the next section.

5.1.2 Extensions of the statement part provide possibilities for i) communication between the sequential and the

parallel part of the program. ii) specification of statements in the parallel part of the program and iii) synchronisation and communication between slave processors executing the parallel part of the program.

Communication between the sequential and the parallel part of the program is done with three standard procedures **sendone, sendall** and **receive** which are only available in the sequential part of the program.

Let se be an expression of the sequential part of the program, let q be an index of a slave processor defined in the node declaration part of the network declaration and let pi be an identifier of the parallel part of the program. Then **sendone** (se, q, pi) assigns the value of se to the instance of pi on processor q provided the types match. **sendall** (se, pi) broadcasts the value of se to all instances of pi on the slave processors.

example 4: in the following program a line of 100 slave processors is declared and for each slave processor q the instance of variable pindex on processor q is initialized to q.

```
var i : integer;
net line [1 : 100] with right: from x to x+1 :
                        left: from x to x-1
    end
parvar
pindex : integer;
for i := 1 to 100 do
sendone (i, line[i], pindex)
```

Let si be an identifier of the sequential part of the program, let q be an index of a slave processor and let pe be an expression of the parallel part of the program. Then **receive** (si, q, pe) assigns the value of the instance of pe on processor q to identifier si.

In the above procedures we also allow names of whole arrays for si, se, pi and pe. In this case the whole arrays are copied provided types match.

Statements for the parallel part of the program. Let S be a sequence of ordinary PASCAL statements. Then **parbegin** S **parend** is called a *parallel statement*. Parallel statements can be inserted anywhere into the statement part of the program, but they cannot be nested. The effect of the parallel statement **parbegin** S **parend** is that S is executed on every slave processor. When all are done the next statement is executed.

We have here formally single program multiple data parallelism, but branching on the processor index permits different slave processors to execute different portions of this program.

example 5 continues example 4. We want all slave processors first to execute the sequence of statements S. Then we want processor 1 to execute S_l (l like left), processors 2 to 99 to execute S_m (m like middle) and processor 100 to execute S_r (r like right).

```
parbegin S:
  if pindex = 1 then begin S_l end
                else if pinedex = 100 then begin S_m end
                                      else begin S_r end
parend
```

Synchronisation and communication between slave processors is achieved by a single standard procedure **sendreceive**. It is only available within parallel statements.

Procedure **sendreceive** is only executed when all slave processors have reached a call of this procedure. Slave processors which reach such a call before others wait. A call of this procedure has the form **sendreceive** (e, d, i), where e is an expression and i an identifier of the parallel part of the program and d is a direction declared in the edge declaration part of the network declaration. The parameters e and i can also be names of arrays.

Suppose for all processor indices q Slave processor q has reached a call **sendreceive** (e_q, d_q, i_q). If the directions d_q are not all identical the result is not defined. Otherwise a *communicate round* occurs: each processor sends the value of his instance of e_q to his neighbor in direction d provided such a neighbor exists. Each processor q which receives a value assigns it to its instance of i_q. If the types or the array bounds do not match the result is not defined.

example 6: also extends example 4. We extend the declaration part by

```
j : integer;
y : array [0..9] of real;
```

and the parallel declaration part by

```
k : integer;
x : array [0..9] of real;
```

Let f be a function of one variable. We extend the statement part by

```
for i := 0 to 99 do
begin
    for j := 0 to 9 do
        y[j] := f(10*i+j);
    sendone(y, line[i+1], x)
end
```

This initializes the local arrays x with f(0), ..., f(999) The parallel part of the program begins with a local computation. Then each processor sends its value x[9] to its neighbor to the right. This neighbor assigns this value to x[0].

```
parbegin
        for k := 1 to 9 do
            x[k] := (x[k-1] + x[k]) / 2;
        sendreceive(x[9].right, x[0])
parend
```

Note that no special treatment of edge nodes (line[0], line[100]) is necessary.

5.2 Operating System. On each slave one needs a command interpreter which receives commands from the master. In particular the master has to be able to start and to stop programs on the slaves. For the distribution of programs and for broadcasting data one needs one routine which broadcasts a block of data to identical locations on all slaves.

For the implementation of procedures **sendone** and **receive** one needs routines which transfer blocks of data between the master and a particular slave. For the implementation of **sendreceive** one needs on the master and each slave a set of routines which together do four things: i) synchronization. ii) routing of the permutation network iii) sending a block of data into the permutation network and iv) simultaneously receiving a block of data from the permutation network. In case the slave processors are not able to send and to receive simultaneously one needs two routines which do i), ii) and iii) resp. i), ii) and iv).

5.3 Compiler. The scanner has to be able to handle a few new keywords. Syntax analysis now has to deal with network declarations, declarations for the parallel part, parallel statements and the new standard procedures. The symbol table has to be expanded in various ways: i) one must distinguish between objects declared in the sequential and parallel part of the program. ii) one must be able to store sets of indices for slave processors and iii) one has to be able to store directions as an explicit sequence of edges. All these extensions are routine work.

In case a direction is defined by an expression this expression has to be evaluated for all processor indices before it can be stored. Therefore an interpreter has to be incorporated for such expressions into the compiler.

The virtual net of processors must be mapped on the processors of the real hardware. For each direction the corresponding routing of the permutation network has to be computed. Once the routings are known code generation for the communication routines only has to produce the proper operating system calls and to ensure, that the right parameters for routing and data are passed.

Let P_1, ..., P_s be the parallel statements of the program being compiled and for each i let Code (P_i) be the code produced for P_i. The program which is being generated for each slave has to do the following: Stay in a busy wait loop until the master sends an integer i between 1 and s. Branch on i to Code (P_i). After execution of Code (P_i) return to the busy wait loop.

example 7:

```
var i : integer
net line[1 : 2]
    with left : (2, 1);
         right : (1, 2)
end
parvar
    pindex, y : integer;
    a : array [1..2] of integer;
begin
    sendone(1, line[1], pindex); sendone(2, line[2], pindex);
    parbegin
        y := 1;  a[1] := 1;  a[2] := 1;
        if pindex = 1 then sendreceive (y, right, y)
                      else sendreceive (a, left, a)
    parend
end
```

There are two ways in which incorrect calls of the procedure **sendreceive** can be produced: i) different slave processors could send in different directions and ii) formats of data which are transmitted do not match. Example 7 above is incorrect in both ways. For reasons of efficiency no runtime check for these conditions is implemented. Supplying such a check as a debugging option is planned.

Implementation of the language extension described above is completed and is currently being tested. Table 1 shows startup times and bandwidths for the various communication primitives of the language extension. Table 2 reports speedups and efficiency of the Gaussian elimination on n × n matrices using 32 slaves. Efficiency is the ratio of sequential run time to p times the runtime on p processors.

	startup [μs]	bandwidth [Mbytes/s]
sendall	5.9	4.550
sendone	6.4	4.096
receive	6.5	3.794
sendreceive	5.1	0.550

Table 1
Startup and bandwidth for communication primitives

n	n div p	sequen. [ms]	parallel [ms]	speedup	efficiency [%]
32	1	800	40	20.0	62.5
64	2	5860	260	22.5	70.4
96	3	19060	720	26.5	82.7
128	4	44400	1580	28.1	87.8
160	5	85760	2960	29.0	90.5
192	6	147120	4960	29.7	92.7
224	7	232380	7740	30.0	93.8
256	8	345520	11400	30.3	94.7

Table 2
Speedups and efficiency of the Gaussian elimination on n × n matrices using 32 slaves

6 APL

APL is an interpreted programming language. Although the language is rather old it has a remarkably powerful set of basic operations on vectors, matrices and tensors. This set of basic operations is still very up to date. Indeed it is tempting to say that FORTRAN8X is FORTRAN77 + APL. Many operations on vectors and matrices are naturally suited for parallelism. Thus it is a natural question how well one can speed up the interpretation of APL (or other interpreted languages with powerful primitives for vectors and matrices) using a parallel machine. This question was very thoroughly investigated in [21]. Short summaries of parts of these results with a limited amount of technical detail (due to limitations of space) can be found in [22], and [23].

Here we only give a very short overview and then treat in some technical detail a simple version of the parallel reduction mechanism from [21]. We then present the results of some measurements from [24].

Elementary data types in APL are integer, real, boolean and character. Composite data types are arrays of elementary data. The arrays have dimension (in APL called *rank*) at most 7. Indices in each dimension start at 1. Thus an array A in APL can be specified by a 4-tuple (d, t, s, r) where $d \in \{1, \ldots, 7\}$ is the dimension of A, t the type of the elements of A, $s = (s_1, \ldots, s_d)$ is the vector of upper bounds for the indices in each direction of A (called *shape*) and r is the sequence of elements of the array in the following recursively defined order: For each $i \leq d$ let A(i) be the (d-1) dimensional subarray of A consisting of those elements whose index has first component i. Thus if d = 2 the A(i) is the i'th row of matrix A. Let r(i) be the sequence of elements of A(i). Then r = r(1) ... r(d). Thus matrices are stored in row order. The sequence r is called the *ravel* of A.

Our interpreter now divides work between the master and the slaves in the following way: slaves only manipulate ravels of arrays. For each j the j'th element of r is stored on slave j mod p. The master does everything else.

In [25] the following key observation is made: although APL has a very rich set of operations on arrays, the vast majority of them (in the sense of [11]) can be parallelized by combining parallel algorithms for 6 basic kinds of operations:

1. Vector operations on the ravel elements (in APL called *skalar* functions). It is well known that vector operations can be performed efficiently on parallel machines if vector elements are distributed across processors in the above fashion.

2. Sorting. One uses Batcher's bitonic sort [26].

3. Reshape. This APL operation changes the shape vector of an array. It is not required to keep the number of array elements the same. If the number decreases say from e to e' one simply discards the last e - e' ravel elements (in this case the slaves have to do nothing at all). If the number increases one has to generate the new elements by making a sufficient number of copies of the original ravel and then discarding surplus elements. Implementation in this case is by recursive doubling.

4. Reduction. Let f be a function with two arguments and let $v = (v_1, \ldots, v_n)$ be a vector. The f-*reduction* $/f(v)$ of v is defined in the following way:

$$u_n = v_n, \; u_{j-1} = f(v_{j-1}, u_j), \; /f(v) = u_1.$$

In APL reduction can be performend in many ways: on vectors, on every row or column of a matrix, in general along arbitrary directions of arrays. Two kinds of difficulties occur if one tries to parallelize reduction.

4.1: The function f might not be associative. There are two techniques which help to reduce this case to the assiciative case for most functions f. If f is subtraction, then replace v_j by $(-v_j)$ for all even j and compute +/ instead of -/. If f is division proceed in a similar fashion. If $f: Y \times Y \to Y$ and Y is finite (e.g. f is NAND, $Y = \{0, 1\}$) then the following helps: each $y \in Y$ defines a function $g_y : Y \to Y$ namely $g_y(z) = f(y,z)$. Compute $/f(v) = g_{v1} \circ \ldots \circ g_{vn-1}(v_n)$ where $\circ$ is function composition which is associative.

4.2: For associative functions f we will treat here the case of reducing the columns of a q × u matrix A. For

simplicity we assume that the number q of rows is a power of 4. There is an obvious algorithm which works in $\log_2$ q rounds. After the i'th round rows with numbers t such that t ≡ 1 mod 2^i contain the result of the reduction of rows t, t+1, ... t+2^i-1 (see figure 4). The permutation used during round i is a cyclic left shift over a distance of u2^i.

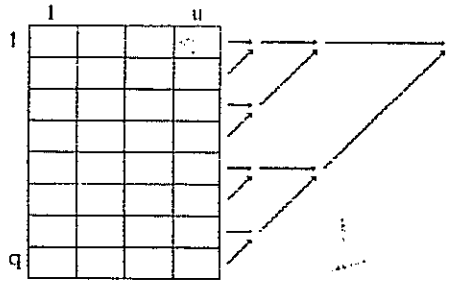

Figure 4

This method gives good speedup if u ≥ p. But if u is small then there is a risk that elements accumulate quickly on very few slaves. Suppose for example that u and p are powers of 2 and q = 1 (this is simply the reduction of a vector). Then after $\log_2$ p rounds all elements are on slave 0. If u is much larger than p then efficiency becomes very poor.

This difficulty is overcome with the following recursive algorithm: In order to reduce the columns of q × u matrix A

i) compute A[i, j] := f(A[i, j], A[i+1, j]) for all odd i ≤ q and all j ≤ u (reduce adjacent pairs, see figure 5). The permutation used is a cyclic left shift by u.

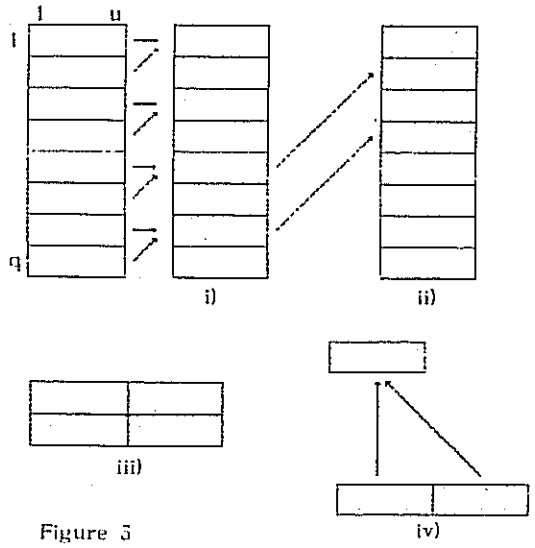

Figure 5

ii) A[i+1, j] := A[q/2 + i, j] for all i and j as above (shuffle the remaining rows). The permutation used is a cyclic left shift by (q/2 + 1)u.

iii) Treat the remaining data as the ravel of a q/4 × 2u matrix B. Reduce the columns of B.

iv) A[i, 1] := f(B[i, 1], B[u+i, 1]) for all i ≤ u. The permutation used is a cyclic left shift by u.

5. Outer product. Let A resp. B be 1-dimensional arrays of length a resp. b and let f be a function with two arguments. The *outer product* of A and B with respect to f is the a × b matrix C with C[i, j] = f(A[i], B[j]) for all i and j. The computation of the outer product is reduced to the computation of a skalar function once one has generated two a × b matrices A' and B' such that the columns of A' are copies of A and the rows of B' are copies of B. B' is computed from B by a reshape operation. A' is computed from A by running the reduction algorithm backward.

6. Scatter resp. gather (in APL called *indexing*). If A, B and I are vectors one has to compute A[I[i]] := B[i] resp. A[i] := B[I[i]] for all i. If one stores ravels in the way we do there is no way to get any speed up in the worst case (all I[i] mod p are equal). Using a combination of routing by sorting [2] and random routing [27] one gets an algorithm with good average behaviour [21].

Not all algorithms described here are implemented yet. Measured speedup for scalar functions is of course very good. For + - reduction on integers *measured* speedup with 32 processors with a problem size of 10^5 elements is by a factor of 3.7. Quadrupling the bandwidth of the network would increase the speedup to 8.1. This demonstrates on one hand that complicated APL functions can be evaluated faster using parallelism in reality. On the other hand requirements on network bandwidth and latency are quite severe because in relation to computation steps there are many rounds of communication and in each round only few data are transmitted.

7 Applications

The paper [28] contains a systematic theoretical study of 12 - 17 parallelizable algorithms from applied mathematics. The following algorithms were considered:

- Solvers for sparse or banded systems:
  Gaussian Elimination, LU Factorization, Gauß-Jordan-, Jacobi-, Gauß-Seidel-, SOR- and CG-method.
- Implementations of numerical integration, extrapolation and ADI-algorithms
- Different solving strategies for ODE's and PDE's based on explicite or implicite discretization rules:
  shooting method, finite difference method and multi-

grid method with different smoothers, restrictions and prolongations.

The study was originally motivated by the question how much poor communication network performance affects performance of parallel machines on typical applications.

The following model of parallel machines is used: There are p processors interconnected by a crossbar or permutation network. Double precision floating point operations cost 1 unit of time. Sending an array of n floating point numbers through the network costs time S + n/B. S is the *startup time* and B is the *bandwidth* of the communication mechanism. We consider some examples.

DATIS-P has 0.1 MFLOP/s processor performance, a bandwidth of 0.8 Mbyte/s and a startup time of 5 µs if parallel PASCAL is used. Thus 10 µs amount to 1 unit of time. S = 5/10 = 1/2. One floating point number has 8 bytes. Not counting startup time the transmission of n numbers takes $8n/(8*10^5)$ seconds which is n/(10/10) units of time. Thus B = 1.

SUPRENUM has a bandwidth of 5 Mbyte/s available for each processor. Startup time is above 1 ms and peak processor performance without chaining is 10 MFLOP/s. One gets B = 1/16, S = 10^4.

An Intel IPSC/i860 has a bandwidth of 2 Mbyte/s available to each processor, a startup time of 70 µs and a peak processor performance of 30 MFLOP/s. One gets B = 1/120, S = 2100.

The reader should keep in mind that S and B are relative values. The unit of measurement for S and B is the time to perform a floating point operation which is for SUPRENUM 100 times smaller and for the Intel machine 300 times smaller than for DATIS-P. Because the machines of SUPRENUM and Intel are multi user systems they have to support asynchronous communication, simply because the processes of different users communicate at different times. Even measured in absolute terms this increases the startup time severely.

The results of the study can be summarized as follows:

i) If the memory of the processors is large enough (≥ 8 Mbyte) *and applications are large enough* then due to surface effects the efficiency of numerical applications will be reasonable (40 - 80%) even if startup time is high (S = 1000 or more) and bandwidth is moderate (B = 1/10) [28], [29]. Although these conditions are mild none of the above machines quite meets them. However the results indicate that numerical algorithms are extremely forgiving against poor balance of computing power versus communication mechanism and that reasonable efficiency can be expected from all three machines considered above.

ii) All parallel algorithms studied came out naturally to fit the basic paradigms. Using *multicast* would give speedups in quite a few cases. But no situation occured where asynchronus communication would have helped.

Some experiments have been performed in order to validate the model described above. A general method from [28], [30] for parallelizing solvers of tridiagonal systems of equations was implemented on DATIS-P and the measured speedups were compared with the predicted ones. The differences were less than 4 % [29].

Where to search for an example, where synchronous communication is bad. In [28] only basic mathematical algorithms are investigated. The underlying communication graphs are very regular, usually grids and trees. In applications however irregular graphs G arise by glueing together two or more regular communication graphs, e.g. in simulations of flows around airplanes at the boundary between body and wings. Using asysnchronous communication primitives it is easy to specify communication along all edges of G. The programmer has not to worry in which order to pair up processors as is required by the usual synchronous mechanisms. The asynchronous communication systems takes care of this and given the graph G this could even be automated. An opinion held by some developpers of application programs is that this cannot be achieved with a synchronous mechanism.

We show next how to handle this situation with the language extension described above if the graph G stays the same for the whole computation. First of all observe that the language extension requires no regular graphs. There is of course a danger that the programmer specifies irregular graphs with more than a minimal number of directions, particularly in situations where two (regular) graphs G_1 and G_2 are glued together. Suppose the programmer has used a set of directions d ∈ D in order to desrcibe G_1 and a disjoint set of directions e ∈ E in order to describe G_2 but at the border between the graphs only few directions from E and D occur. Then an obvious implementation will generate |E| + |D| rounds of communication whereas an asynchronous mechanism might do much better.

But there is no need to compile this the way it is specified by the programmer. As mentioned in section 1.1 if graph G has degree d, then there are always d permutations which help to realize communication along all edges of G and these permutations can be found algorithmically, e.g. by the Hopcroft-Karp maximum matching

algorithm [31]. Those familiar with this algorithm will object that it is too slow and that the asynchronous mechanism is much faster. Indeed it is because it only produces approximations to maximum matchings, and these matchings can be produced very easily by the compiler: all it has to do is to simulate the behaviour of the asynchronous mechanism *once* (!) on G.

The next case we treat occurs in numerical algorithms which do automatic grid refinements. Such algorithms are presently studied intensively [32]. In this case the graph G changes during the computation but between changes there are many rounds of computation. The same mechanisms can be used for compilation. The language extension described above has to be modified in order to allow dynamical changes of the communication graphs.

We can however imagine a situation where asynchronous communication might indeed be better: if the graph G changes so quickly that the time to "compile" the graph is not amortized over enough rounds of fast synchronous communications. Presently we are not aware of numerically intensive applications which produce this situation.

8 Interface to the German wide area high bandwidth network

Germany has a wide area network for video conferences. The network has a bandwidth of 140 Mbit/s. Because the fee for using the network is about $ 100 every 15 minutes the network is not used much. Therefore Telecom, the German telephone company, allows the network also to be used under a different name (VBN) for transmission of data between computers. The first interface between the VBN network and ethernets has been constructed at the university of Stuttgart. The connection between the supercomputers of Stuttgart and Karlsruhe which was established with this interface has deservedly received much attention.

Efforts are now under way to establish a high speed network between the major academic computing centers and research institutes of Germany. More applications are beginning to evolve. For example publishers and catalog warehouses are interested to use high bandwidth wide area networks for the transmission of images. Thus there is some interest in base technology for interconnecting computers to networks like VBN.

Producing the Stuttgart interface was a difficult task because at the time the only interface provided by the VBN terminal devices (this is the box which is installed at the user) was 1 bit serial with a cycle time of 7 ns. This is too fast for TTL technology. Also the clock had to be derived from the signals on the 1 bit interface by means of a phase locked loop.

In this situation we convinced Telecom as well as ANT, the manufacturer of the VBN terminal devices, that a parallel interface with a slower cycle time would make it much easier to construct adapters between computers and the VBN network. Subsequently a 10 bit interface with a 70 ns cycle time was defined and incorporated into the VBN terminal devices. It was also agreed that we would build an adapter between this new interface and the VME bus [33].

Figure 6 gives a schematic view of this adapter. Besides the interfaces to the VME bus and the VBN network it has 2 banks of static RAM. These banks are used as send buffers and receive buffers. Each bank is internally subdivided into two subbanks in order to make simultaneous access from the VME bus side and the VBN side possible. Each subbank has 32 K words which are 12 bits wide (for technical reasons). Data are transmitted through the VBN network in packets of up to 32 Kbyte. Maximum transmission rate measured from VME bus to VME bus was 8 Mbyte/s due to limitations of bandwidth of the VME bus and because no DMA controller was used. With DMA the transmission rate should increase to 16 Mbyte/s.

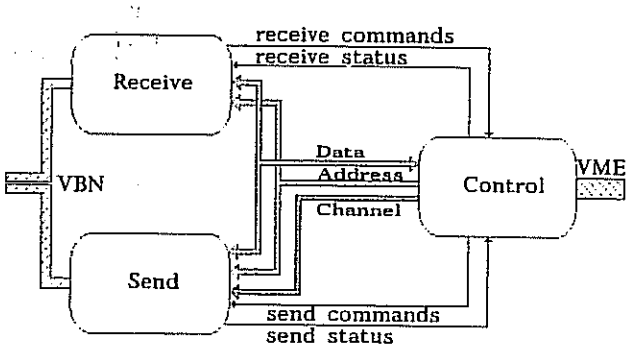

Figure 6

The interface was demonstrated by ANT and Telecom at the CeBit 90 computer exhibition at Hannover. A graphics terminal at Hannover was connected to the master processor of DATIS-P in Saarbrücken. Net transmission rate from DATIS-P to the graphics monitor was 1.2 Mbyte/s due to the bandwidth of the graphics card used in the terminal. Our system had no crashes during the exhibition.

9 Conclusion

We have built a fault tolerant parallel machine which is based on some nontrivial theory and on two very simple and at first glance restrictive basic paradigms. Careful analysis indicates that as far as performance is concerned not much is lost by these restrictions if algorithms from numerical mathematics are run. The conceptual simplicity gained by the restrictions has on

the other hand permitted us to realize hardware, software and some applications of the machine reasonably fast and reliably with the limited resources of a university environment. The speed of the simple communication mechanism used allows even the efficient parallelization of most APL functions.

10 Acknowledgement

We thank the referees for helpful suggestions.

References

[1] Waksman, A., "A Permutation Network", *J. ACM 15*, 1 (January 1968), 159 - 163

[2] Galil, Z., Paul, W.J., "An Efficient General-Purpose Parallel Computer", *JACM*, Vol. 30, No. 2, April 1983, 360 - 387

[3] Valiant, L.G., "A Scheme for Fast Parallel Communication", *SIAM Journal on Computing*, 11, 1982, 350 - 361

[4] Auerbach, D.J., Paul, W.J., Bakker, A.F., Lutz, C., Rudge, W.E., Abraham, F.F., "A Special Purpose Parallel Computer for Molecular Dynamics: Motivation, Design, Implementation, and Application", *J. Phys. Chem.*, Vol. 91, No. 19, 1987, 4881 - 4890

[5] Giloi, W.K., "SUPRENUM: A Trendsetter in Modern Supercomputer Development", *Parallel Computing*, Vol. 7, North Holland, 1988, 283 - 296

[6] Intel, *iPSC/2 User's Guide*, Intel Scientific Computers, August 1988

[7] Paul, W.J., Pippenger, N., "Parallel Computers Coupling a Permutation Network", *IBM Technical Disclosure Bulletin*, Vol. 28, No.7, Dec. 85, 3050 - 3051

[8] Benes, V.E., *Mathematical Theory of Connection Networks and Telephone Traffic*, Academic Press, New York, 1965

[9] Swifte, *The Multi-User Operating System for the 68000 Family*, Swifte Computer System Ltd., 1987

[10] Iverson, K.E., *A Programming Language*, Wiley, New York/London, 1962

[11] Association for Computing Machinery, "Draft Proposed Standard Programming language APL", *APL Quote Quad*, Vol. 14, No. 2, 1983

[12] Seagate, *Seagate SCSI Interface Manual*, Rev. B, Seagate

[13] Preparata, F.P., Vuillemin, J., "The Cube-connected-cycles: A Versatile Network for Parallel Computation", *Conf. Rec. 20th Ann. IEEE Symp. on Foundations of Computer Science* (Puerto Rico, Oct. 79), IEEE, New York, 1979, 140 - 147

[14] Thompson, C.D., "Area-Time Complexity for VLSI", *Proc. of the 11th Ann. ACM Symposium of Theory of Computing*, 1979, 81 - 88

[15] Lev, G., Pippenger, N., Valiant, L.G., "A Fast Parallel Algorithm for Routing in Permutation Networks", *IEEE Trans. Comp.* C - 30, 2 (February 1981), 93 - 100

[16] Dally, W.J., Seitz, C.L., "The Torus Routing Chip", *Distributed Computing*, Vol. 1, No. 4, Oct. 86

[17] Ramo, S., Whinnery, J.R., van Duzer, T., *Fields and Waves in Communication Electronics*, Second Edition, John Wiley & Sons, New York, 1984

[18] Glaes, I., *Lösung spezieller Probleme im Multiprozessorsystem DATIS-P-256*, Diplomarbeit, FB 14, Universität des Saarlandes, 1989

[19] Bergmann, Schäfer, *Lehrbuch der Experimentalphysik*, Band I, de Gruyter, Berlin, 1974.

[20] Forsythe, Wasow, *Finite Difference Methods for Partial Differential Equations*, John Wiley & Sons, New York - London, 1960

[21] Sauermann, J., *Ein paralleler APL-Rechner*, Dissertation, FB 14, Universität des Saarlandes

[22] Sauermann, J., "APL on the DATIS-P Parallel Machine", to appear in: *Proc. DMCC 5*, 1990

[23] Sauermann, J., "A Parallel APL Machine", to appear in: *Proc. APL 90*, 1990

[24] Bingert, A., *Parallelisierung der APL-Funktion Reduction*, Diplomarbeit, FB 14, Universität des Saarlandes, 1990

[25] Saal, H.J., Weiss, Z., "An Empirical Study of APL Programs", *Computer Languages*, Vol. 2, 1977

[26] Batcher, K.E., "Sorting Networks and their Applications", *Proc. AFIPS 1968 SJCC*, Vol. 32, 1968

[27] Valiant, L.G., "Universal Circuits", *Proc. 8th. Ann. Symp. on Theory of Computing* (Hershey, Pa, May 1976), ACM, New York, 1976, 196 - 203

[28] Müller, S.M., *Die Auswirkungen der Startup-Zeit auf die Leistung paralleler Rechner bei numerischen Anwendungen*, Diplomarbeit, FB 9, Universität des Saarlandes, 1989

[29] Müller, S.M., Scheerer, D., "A Method to Parallelize Tridiagonal Solvers", submitted to *Parallel Computing*, North Holland, 1990

[30] Paul, W., Müller, S.M., "Contributions of Theoretical Computer Science, Applied Computer Science and Numerical Mathematics to Design of Parallel Computers", *Proc. IFIP 11th., World Computer Congress*, San Francisco, 1989, 459 - 460

[31] Hopcroft, J., Karp, R., "An $n^{5/2}$ - Algorithm for Maximum Matchings in Bipartite Graphs", *SIAM J. Comput.*, Vol. 2, No. 4, Dec. 1973

[32] Deuflhard, P., Leinen, P., Yserentant, H., *Concepts of an Adaptive Hierachical Finite Element Code*, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, September 1988

[33] Motorola, *The VMEbus Specification*, Motorola, Series in Solid-State Electronics, Rev. C.1, October 1985