First Experiences with the Intel Paragon

Peter Arbenz
Institute for Scientific Computing
Swiss Federal Institute of Technology (ETH)
Zurich, Switzerland

We give a brief overview on hardware and software on the Intel Paragon. By means of a simple example we introduce the most important message passing primitives and show how performance of a program depends on their proper choice.

1.Introduction

In December 1993, a massively parallel processor (MPP) computer—the Intel Paragon XP/S5+ of Intel's Supercomputing Systems Division—was installed at ETH Zurich's central computing facility. After a test phase during which hardware and software were improved considerably, the machine passed the acceptance tests by July 1994.

The Intel Paragon is a scalable distributed-memory multicomputer. Its architecture supports Multiple Instruction stream/Multiple Data stream (MIMD) and most notably Single Program/Multiple Data (SPMD) styled applications. It is based on Intel's i860XP RISC processor and a high-speed inter-connection network. The Paragon follows the Intel iPSC/1, iPSC/2, iPSC/860, and Touchstone Delta, as the fifth in a very successful and influential row of MPP computers. The first three machines were built around a hypercube network. The Delta and Paragon network topology is a planar mesh.

The purpose of this new machine is to give Swiss researchers the possibility to get first-hand experience with the use of massively parallel multicomputers. The Paragon system at ETH Zurich is already supporting large production applications running in the fields of computational fluid dynamics, ordinary differential equations, field theory and non-relativistic electron gas simulations. Many porting and development efforts are under way in such diverse areas as parallel numerical algorithms, computation of protein surfaces, combinatorial optimization, searching in large parameter spaces, electro-magnetic field computations, drought monitoring, molecular simulation and Monte Carlo simulations of polymers.

For technical and economic reasons, the future of high speed computing will be based on scalable massively parallel multicomputers. It is however not yet clear at all how the hardware and software environment will look like. Therefore, at ETH, Intel's MPP has been complemented by two powerful workstation clusters, the C^4-cluster with 20 IBM RS 6000/590 processors and the Convex/HP cluster (Meta Series) with 16 HP 735 processors. Also, a successor of A. Gunzinger's Music system [3] is under way, that is based on DEC's Alpha chip.

2. Hardware

The Intel Paragon at ETH Zurich consists of 112 nodes. Its compute power stems from the 96 *compute nodes* that are arranged in a grid of size 6 x 16, cf. Figure 1. Attached to this grid are one *boot node*, 9 *I/O* nodes, and 6 *service nodes*, that are acting as logical front-ends. The (parallel) execution of application programs takes place on the compute nodes.

The hardware configuration of compute and service nodes is identical. This means that nodes of one kind can be reconfigured to become nodes of the other kind. Each node consists of two i860XP RISC processors, see Figure 2. One of them is working as the *application processor*, the other one as the *message processor*. They share a common memory. The purpose of the message processor is to relieve the application processor from the overhead work related to message-passing. The message-passing processor sends and receives messages from the other nodes via the *network interface* (Line Transfer Unit, LTU, in Figure 2). The application processor can concentrate on the node's primary task, computing on compute nodes, compiling, editing, etc. on service nodes, accessing external storage devices on I/O nodes. All nodes have 32 MByte of main memory. This sums up to 3.6 GByte of main memory. 8 RAID disks of 4.8 GByte each are connected to the I/O nodes.

The i860XP processor runs at 50 MHz. It has two pipelines for floating-point operations, an adder that can produce a result (64-bit IEEE arithmetic) every cycle, and a multiplier which can produce a result every other cycle. This adds up to 75 Mflop/s *peak performance* of a single RISC processor and an overall peak performance of the complete system of $96 \cdot 75$ Mflop/s = 7.2 Gflop/s. The performance that can be expected realistically is around the Linpack benchmark performance of about 10 Mflop/s per node.

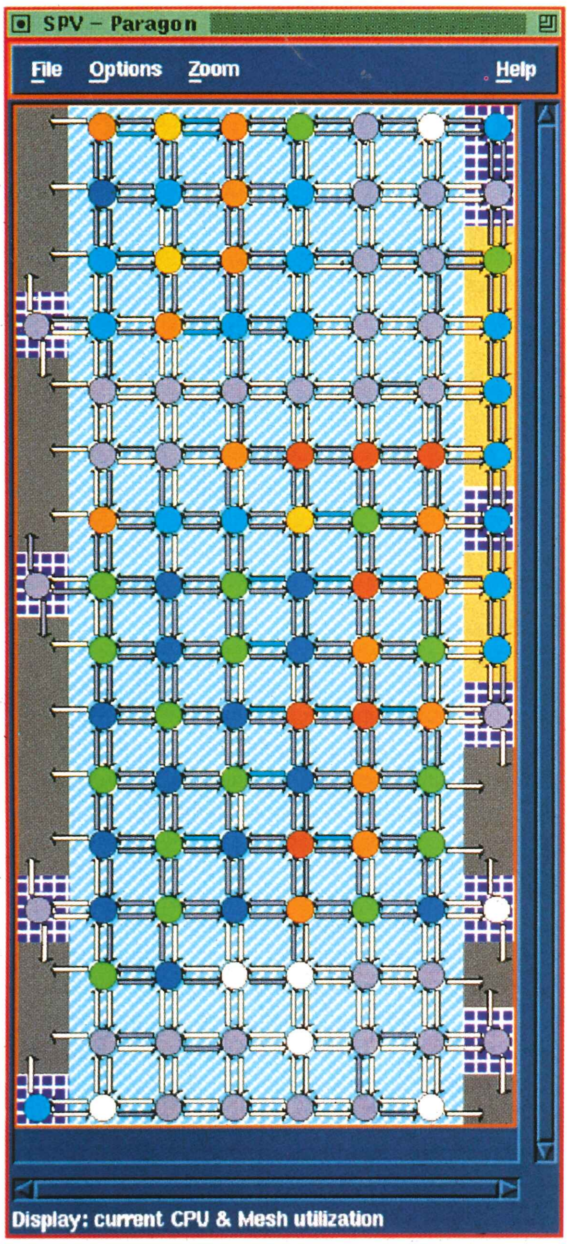


Figure 1. The grid structure of the ETH Intel Paragon XP/S.

The network has a theoretical bidirectional bandwidth of about 200 MByte/s. Measured performance under the currently installed operating system shows about 65 MByte/s for bandwidth and about 65 µsec for *latency*. Latency is mainly due to software overhead. So, the time to transfer a message of n 64-bit floating point numbers takes about

$$(65 + 0.12n)\ \mu\text{sec} \tag{1}$$

Notice that the bandwidth is in good balance with the 10 Mflop/s performance of the processor. The transfer time does not depend much (~5%) on the distance between communicating nodes, so that one need not bother where nodes are actually placed. Formula (1) indicates that short messages should be avoided. However, as we will see later,

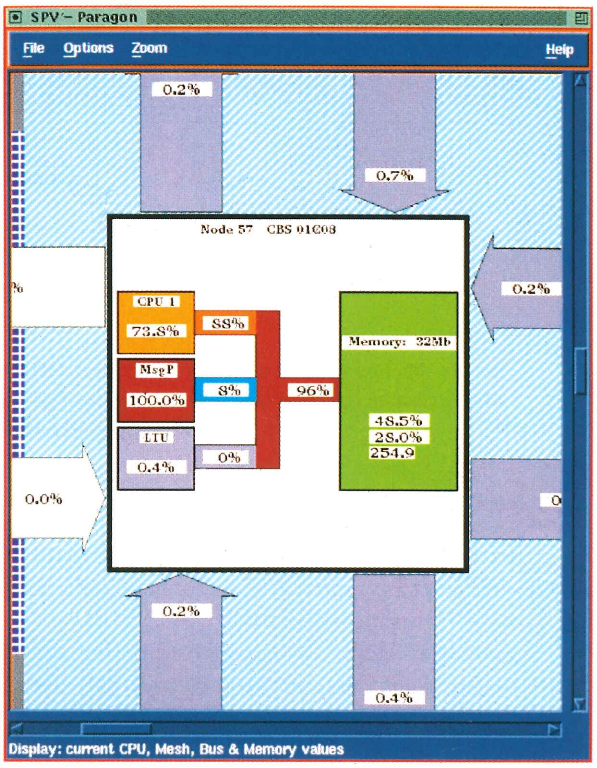


Figure 2. A view of the Paragon node

it is possible to hide some of the overhead regarding message-passing (*latency hiding*).

The theoretical processor-to-memory bandwidth is about 400 MByte/s.

The best known benchmark for floating point performance is the Linpack benchmark. In Table 1 we give the most recent numbers [2].

Linpack Benchmark, Mflop/s				
p	$n = 100$	TPP Best Effort $n = 1000$		Theoretical Peak
1	10	34		50
Highly Parallel Computing				
p	$r_{\max}$ (Gflop/s)	$n_{\max}$	$n_{1/2}$	r_{peak}(Gflop/s)
64	2.0	8000	2000	3.2
128	4.0	12000	3000	6.4
256	7.6	16000	4000	13
512	15.2	23000	9000	26
1872	72.9	55000	17500	94
3680	143.4	55700	20500	184

Table 1. Benchmarks for the Paragon

3. Software

The Paragon operating system is OSF/1, Release 1.2, a version of UNIX. A copy of the Mach 3.0 Microkernel resides on each node and implements core operating system functions. (Despite its name, the microkernel occupies about 6MB of the main memory.)

For simplicity of administration and performance reasons, at ETH only one user application process can run on one node. So-called time and space sharing mode are disabled.

UNIX services such as the file system server run on the service nodes with access transparently provided from each microkernel.

Although each node of the Paragon runs its own operating system, the highly parallel machine appears as a single system to the user with a single process id space and a single file system. Every file, every process and every network service is available to every (authorized) application.

Compilers are provided so far for the programming languages Fortran 77, C, C++ and Ada. Intel's High Performance Fortran (HPF) is promised for later this year. The Fortran compiler in particular the linker is quite slow. There are Fortran and C cross-compilers. So, a program can be edited and compiled by the programmer locally on his/her workstation and only the executable has to be transferred to the Paragon. At ETH, workstations and supercomputers are networked via the Andrew File System (AFS), which makes files visible from workstations as well as the Paragon.

The only parallel programming model that is provided so far on the Paragon is the message-passing programming model. The proprietary NX message-passing interface [7] is a rich library of routines for the application programmer, which provides various forms of send and receive functions. Portable message-passing environments as PVM, Parmacs, or MPI are mapped onto NX.

There are a number of tools available for the Paragon: The System Performance Visualization tool (SPV) displays CPU, mesh, and memory bus utilization values. The values are updated in short time intervals. Figure 1 shows a global view of the machine. In this picture a 16-processor and a 48-processor job can be observed. Figure 2 shows the utilization of a single node. It allows to observe the behavior of an application program on-line. Also it visualizes the usage of the machine and also what part of the machine is free to be used.

The Interactive Parallel Debugger (IPD), a source level debug tool, implements all the capabilities associated with traditional symbolic debugging, plus a variety of observing, controlling, and repairing complex, multi-node application programs.

ParaGraph is a performance visualization tool. It takes as input a trace-file that is produced by the Paragon performance monitoring subsystem during an actual run of a parallel program. The resulting trace data can then be replayed graphically with ParaGraph to provide a dynamic depiction of the behavior of the parallel program. Figure 5 shows a space-time diagram that visualizes the activity of the processors and the messages being sent. This tool is very useful. Unfortunately, it seems not to work properly if asynchronous message-passing is used.

Intel provides a small *library* of parallel scientific subroutines. ProSolver is a collection of routines for solving systems of linear equations. There are routines for matrices that are stored in dense, skyline or sparse form. The first two are direct solvers, the last solves the system iteratively with the conjugate gradient method with incomplete Cholesky factorization as preconditioner. Furthermore there is a FFT library, and a version of Lapack for scalable parallel computers, ScaLapack [1], available for the Intel Paragon.

Optimized for the execution on the nodes only are, among others, the BLAS, Linpack, Eispack, Lapack, and the subroutines of the NAG scientific library.

4. Operating the Intel Paragon

Presently, during the day from 8am to 7pm, jobs run interactively on the compute nodes. During this time the Paragon is divided into two *partitions* consisting of 32 and 64 compute nodes, respectively. An application program is executed within one partition. The smaller partition is provided for runs of applications which are in the development phase.

Very time consuming production runs are performed during the night, when all compute nodes belong to one big 96 node partition which can be accessed only in batch job mode. The batch jobs are scheduled by the NQS system.

On a weekly basis, the work load of the 96 compute nodes is 20-40%. Most of the load is due to large batch jobs. System breakdowns occur 1-3 times per week.

5. The Message-passing Programming Model

Message-passing is so far the only means of communication among user processes on the Paragon. Data exchange among processes via virtual shared memory is only planned. The NX message-passing interface contains several routines for sending and receiving messages [4]. We introduce some of them by means of the following simple example. We consider a parallel program running on nodes $0 \ldots p-1$ that are arranged logically in a ring. Each copy of the program holds a m x m matrix A and m x n matrices B, C. A and B are initialized differently on each node. We want C to be the matrix product BA of the matrices B and A of the *previous* node. A first approach could look like in the following program excerpt: the resulting matrix is sent in pieces of size m x *bsize*, where *bsize* is the width of the block of *C* from column *bstart* + 1 to column *bstart* + *bsize*.

```
subroutine  doit (m,n,A,B,C,...)

  include 'fnx.h'                 !  link NX library

  double precision A(m,n),B(m,m),C(m,n)
  ...

  p = numnodes()                  !  number of nodes
  myid = mynode()                 !  node id
  next = mod(myid+1,p)
  prev = mod(myid+p-1,p)

  bstart = 0
  ib = 0
1 continue
     ib = ib + 1
     ...

     call dgemm ('N','N',m,bsize,m,1.0,A,m,
  >        B(1,bstart+1),m,0.0,sndbuf,m)

     dummy = csend(ib,sndbuf,bsize*m*8,next,0)
     dummy = crecv(ib,rcvbuf,bsize*m*8)

     do j=1,bsize
        do i=1,m
           C(i,bstart+j) = rcvbuf(i+(j-1)*m)
        enddo
     enddo

     bstart = bstart + bsize
  if (ib .lt. nb) goto 1

  ...
```

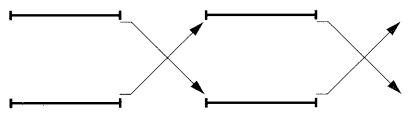
The program, which is loaded on all participating nodes, first computes portions of *BA* which are stored in a buffer variable. Then it calls the function csend. By this, a message of type *ib* is issued which sends $bsize \cdot m \cdot 8$ Byte of data stored in array *sndbuf* to the *next* processor. The type variable is an identifier that allows the receiver to distinguish between different messages from the same origin. We chose the type to be the number of the block being sent. The last parameter has no meaning if space sharing is disabled, but has to be 0. If the message is submitted the function call returns an unimportant value. When the programs returns from crecv the message has arrived and is stored in *rcvbuf* from which it can be copied into *C*.

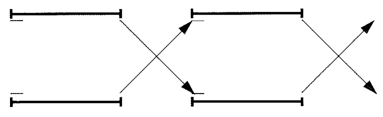
The functions csend and crecv are implementations of so-called *synchronous* or *blocking* send and receive. This means that the functions return only if the message-passing operation has completed. In Figure 3(a) we show the situation with two processors. The thick lines indicate computing, the thin lines message-passing. For the latter, horizontal lines indicate the time for startup, oblique lines for transmission.

In many instances, proceeding in this way is too time consuming. In the above example we could, e.g., receive the *ib*-th portion of *C* from the previous node while we are computing the local *ib*-th portion. We just have to make sure that the data have arrived when we use *rcvbuf*. Analogously, we just have to make sure that the send buffer has been emptied before we fill it anew in the next call to dgemm. The necessary *asynchronous* or *non-blocking* message-passing primitives are isend and irecv.

```
   ...
1 continue
   ...
   rcvmsg = irecv(prev,rcvbuf,bsize*m*8)
   if (ib .gt. 1) call msgwait(sndmsg)

   call dgemm (... ,sndbuf, m)

  sndmsg = isend(myid,sndbuf,bsize*m*8,next,0)
   call msgwait(rcvmsg)

   do j=1,bsize
      do i=1,m
         C(i,bstart+j) = rcvbuf(i+(j-1)*m)
      enddo
   enddo
   ...
 if (ib .lt. nb) goto 1
```

The functions isend and irecv return immediately. The returned handle can be used to test whether the operation has actually completed. There are a number of functions to that end [4]. Here, we only consider the subroutine msgwait, that returns when the completion of the respective asynchronous operation is guaranteed. The rule of thumb for asynchronous message-passing is: *Issue* isend/irecv *as early as possible, call* msgwait *as late as possible.*


(a) Synchronous message passing


(b) Asynchronous message passing

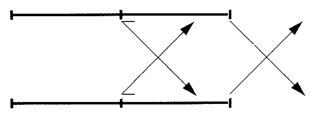

(c) Asynchronous message-passing with pipelining, *nb* = 2

Figure 3. Message-passing.

With asynchronous message-passing we gain most of the startup time (cf. Figure 3(b)). Further, if the irecv call is issued early enough the message processor is able to copy the message directly into the user variable, here *rcvbuf*, instead of storing it into a system buffer first.

Figure 3(b) indicates that the times for the message transfer can take a considerable portion of the overall time. The oblique arrows can be interpreted as the time for waiting until *sndbuf* is completely emptied. A simple remedy to avoid idle compute processors in this situation is *pipelining*. To that end, we introduce a second send buffer. dgemm writes in each of them alternatingly. In the following further improved version of the program we write the arriving message directly into *C* without resorting to the unnecessary buffer *rcvbuf*.

```
   ...
 1 continue
     ...
     rcvmsg(ib) = irecv(ib,C(1,bstart+1),bsize*m*8)
     if (ib .gt. 1) call msgwait (sndmsg1)

     call dgemm(...,sndbuf,m)
     sndmsg1 = isend(ib,sndbuf,bsize*m*8,next,0)
     ...

     if (ib .eq. nb) goto 2
     ...
     rcvmsg(ib) = irecv(ib,C(1,bstart+1),bsize*m*8)
     if (ib .gt. 2) call msgwait(sndmsg2)
     call dgemm (...,sndbuf(n*m/2+1),m)

     sndmsg2 = isend(ib,sndbuf(n*m/2+1),
  >          bsize*m*8,next,0)
     ...
     if (ib .lt. nb) goto 1
 2 do ib=1,nb
     call msgwait(rcvmsg(ib))
   enddo
```

We make sure that all the data have arrived in the appended do-loop.In Table 2 we give times for the three introduced versions, and, for comparison, the times for an equally blocked local matrix multiply (dgemm). Notice that dgemm performs above 40 Mflop/s for the larger matrices. The difference between these times gives the overhead of message-passing. The timings confirm the sketches of Figure 3 to some extent. The gain of asynchronous message-passing is in the order of a few percent. The gain obtained with the second version is substantial, in particular for short messages.

6. An Application

Consider a basic problem from numerical linear algebra: solving a system of linear equations $A\mathbf{x} = \mathbf{b}$, where A is a banded symmetric positive definite matrix with half-bandwidth k. The fastest *direct* algorithm to solve this problem is block cyclic reduction [5]. To that end we partition A as indicated in Figure 4 such that there are p large and p-1 small k x k diagonal blocks. In the first step of block cyclic reduction we eliminate the variables corresponding to the large blocks. The resulting reduced system is block tridiagonal and has order $(p\text{-}1)k$. It is solved by ordinary block cyclic reduction. The complexity of this algorithm is

$$C \approx 4k^2\frac{n}{p} + \left(\frac{7}{3}k^3 + 4\sigma + 2k^2\tau\right)\log_2(p) \text{ flop.} \qquad (2)$$

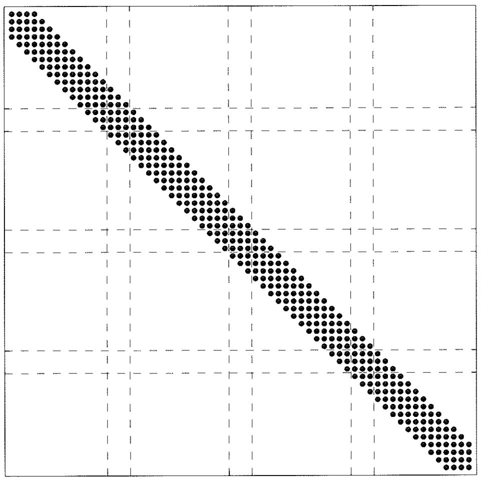


Figure 4. Band matrix A with n = 60, p = 4, k = 3.

Here, τ and σ denote the number of floating point operations that can be executed during the time for the transfer of one 64-bit floating point number and during message-passing startup, respectively.

The important thing to note here is that there are portions of C that *grow* with the number of processors. Often, constant terms or terms growing with p are solely due to communication, in which case latency hiding may be prerequisite to make an algorithm scalable. Otherwise, speedup is limited with increasing p (Amdahl's law) or there is, as in our case, a number of processors, p_{opt}, for which speedup is maximal. In the cyclic reduction algorithm above, $p_{\text{opt}} \approx 12n/7k$. The corresponding speedup is $S_{\text{opt}} \approx 3n/7k$. This number is not very high, mainly because the parallel algorithm has a high redundancy of ~4.

Also in this algorithm it is useful to hide latency. However, because of the k^3 term in (2), this technique can only improve p_{opt}. Thus, this algorithm and related direct algorithms for solving sparse equations (domain decomposition approach) are not scalable. In Figure 5 the execution behavior is shown for the program with latency hiding. Horizontal lines indicate the status of the processors, oblique lines visualize the messages among

n	*m*	*nb*	csend/crecv	isend/irecv	isend/irecv 2nd version	pure dgemm
4000	40	2	400.84	377.06	354.96	313.60
4000	20	2	140.13	133.18	110.49	90.01
800	40	2	80.11	77.30	71.04	62.30
800	20	2	28.30	27.13	22.57	17.99
800	10	2	12.80	12.08	9.27	6.45
400	40	2	40.35	39.55	37.41	31.06
400	20	2	14.71	14.04	11.78	9.08
400	10	2	7.00	6.51	4.89	3.37
200	40	2	20.75	19.98	18.63	15.82
200	20	2	7.88	7.55	6.45	4.71
200	10	2	3.74	3.64	2.67	1.65
200	4	2	2.81	2.96	2.17	1.68
200	2	2	1.48	1.57	0.94	0.60
200	11	2	1.25	1.15	0.62	0.28

Table 2. Timings with 8 processors and optimal *nb* on the Paragon [Milliseconds].

processors; the color of these lines corresponds to the volume of the messages.

7. Concluding Remarks

The previous sections should have shown that it takes a considerable effort to obtain well-performing programs for massively parallel processor computers. Message-passing programming is far away from traditional programming and can be quite tedious. It is sometimes easier to get a HPFortran program running. Whether it is easier to get (portable) *high performance* with one programming model or the other is still an open question.

The large step from a conventional single threaded to a message-passing program and the uncertainty of how future MPPs will be programmed have let numerous people hesitate to do the high effort of changing their programs. Concerning the Paragon, the raw performance of the single nodes may have been considered too low, although it is probably easier with an i860 processor to get a good fraction of the peak performance than on an Alpha chip, for example.

The Intel Paragon is now stable enough for efficiently developing and productively running programs. The NX message-passing library is more elaborate than other (popular) libraries, in particular it supports asynchronous message-passing. The Paragon hardware and software provide a good testbed to decide whether message-passing is the way to go.

References

[1] Choi, J., J. J. Dongarra, D. W. Walker and R. C. Whaley. *ScaLapack Reference Manual*. December 31, 1993.

[2] Dongarra, J. J. "Performance of Various Computers Using Standard Linear Equations Software." Technical Report CS-89-85, Computer Science Department (Univ. of Tennessee, Knoxville), August 31, 1994.

[3] Gunzinger, A. *et al.* "Achieving Supercomputer Performance with a Parallel Array Processor."*SPEEDUP* 7(2)(1993): 55-58.

[4] Intel Corp. *Paragon User's Guide*. October 1993.

[5] Johnsson, S. L. "Solving Narrow Banded Systems on Ensemble Architectures." *ACM Trans. Math. Softw.* 11 (1985): 271-288.

[6] Michl, T., S. Maier, S. Wagner, M. Lenke and A. Bode. "A Parallel Implementation of a Navier Stokes Solver on Intel Multiprocessor Systems." *SPEEDUP* 7(2)(1993): 19-23.

[7] Pierce, P. "The NX Message Passing System." *Parallel Computing* 20 (1994): 463-480.

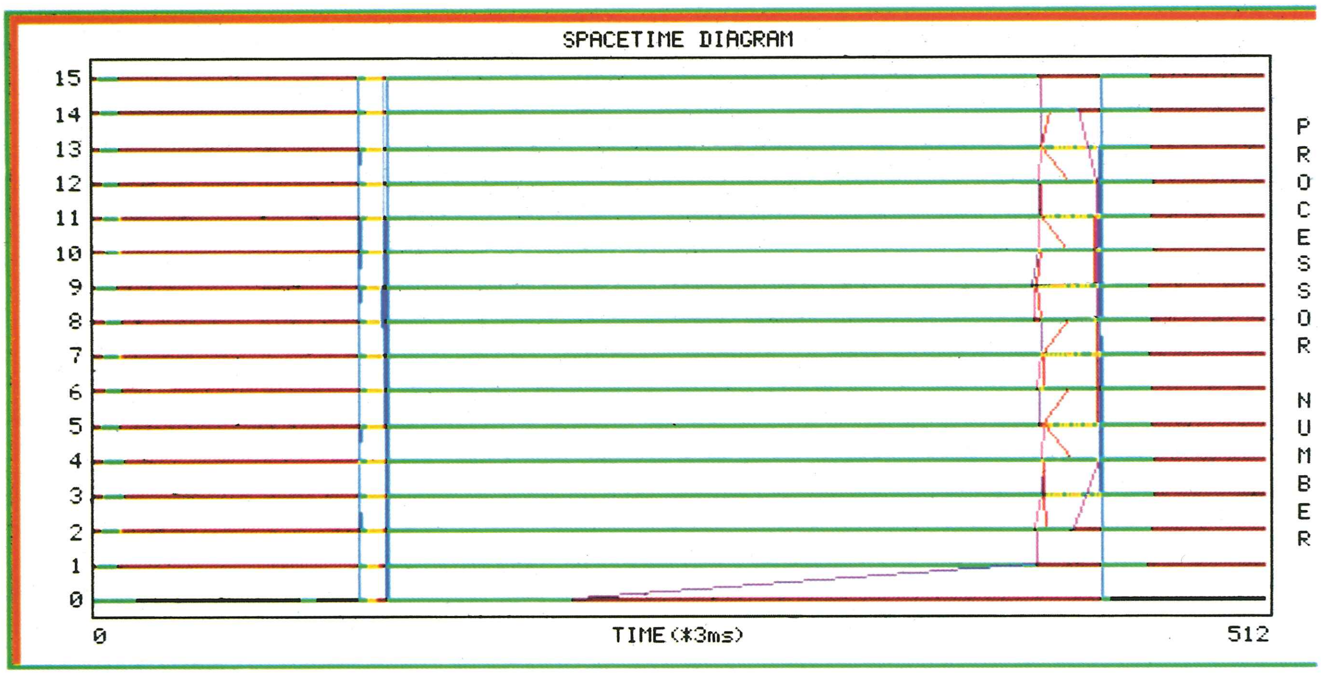


Figure 5. ParaGraph display of band solver based on cyclic reduction