North-Holland
Microprocessing and Microprogramming 29 (1990) 67-82

Vectorization and parallelization of the conjugate gradient algorithm on hypercube-connected vector processors

C. Aykanat*, F. Özgüner** and D.S. Scott***

*Bilkent University, Department of Computer and Information Science, P.K. 8, 06572 Maltepe, Ankara, Turkey[1]

**Department of Electrical Engineering, The Ohio State University, Columbus, Ohio 43210, USA

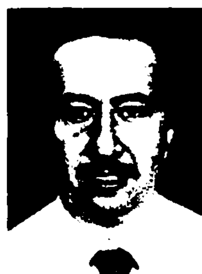
***Intel Scientific Computers, Beaverton, Oregon 97006, USA

Solution of large sparse linear systems of equations in the form $Ax = b$ constitutes a significant amount of the computations in the simulation of physical phenomena [1]. For example, the finite element discretization of a regular domain, with proper ordering of the variables x, renders a banded $N \times N$ coefficient matrix A. The Conjugate Gradient (CG) [2,3] algorithm is an iterative method for solving sparse matrix equations and is widely used because of its convergence properties. In this paper an implementation of the Conjugate Gradient algorithm, that exploits both vectorization and parallelization on a 2-dimensional hypercube with vector processors at each node (iPSC-VX/d2), is described. The implementation described here achieves efficient parallelization by using a version of the CG algorithm suitable for *coarse grain* parallelism [4,5] to reduce the communication steps required and by *overlapping* the computations on the vector processor with internode communication. With parallelization and vectorization, a speedup of 58 over a μVax II is obtained for large problems, on a two dimensional vector hypercube (iPSC-VX/d2).

Keywords: Vectorization, Parallelization, Conjugate gradient algorithm, Hypercube-connected vector processors.

Submitted: 5 July 1989
Submitted for modification: 25 September 1989
Accepted: 1 June 1990

[1]The author was with the Department of Electrical Engineering, The Ohio State University, Columbus, Ohio 43210, USA.



Cevdet Aykanat received the M.S. degree from the Middle East Technical University, Ankara, Turkey, in 1980 and the Ph.D. degree from The Ohio State University, Columbus, Ohio, in 1988, both in Electrical Engineering. From 1977 to 1982, he served as a Teaching Assistant in the Department of Electrical Engineering, Middle East Technical University. He was a Fulbright scholar during his Ph.D. studies. He spent the summer of 1987 at Intel Scientific Computers, Portland, Oregon. Currently, he is an Assistant Professor at Bilkent University, Ankara, Turkey. His research interests include parallel computer architectures, parallel algorithms, applied parallel computing and fault-tolerant computing.

Füsun Özgüner received the M.S. degree in electrical engineering from the Technical University of Istanbul in 1972, and the Ph.D. degree in Electrical Engineering from the University of Illinois, Urbana-Champaign, in 1975. She worked at the I.B.M. T.J. Watson Research Center for one year and joined the faculty at the Department of Electrical Engineering, Technical University of Istanbul. She spent the summer of 1977 and 1985 at the I.B.M. T.J. Watson Research Center and was a visiting Assistant Professor at the University of Toronto in 1980. Since January 1981 she has been with the Department of Electrical Engineering, The Ohio State University, where she presently is an Associate Professor. Her research interests include fault-tolerant computing, parallel computer architecture and parallel algorithms.

David Scott received his PhD. in Mathematics from Berkeley in 1978. He worked at the Oak Ridge National Laboratory for three years and taught in the Computer Sciences Dept at the University of Texas at Austin for four years. For the last five years he has worked at Intel Scientific Computers. He is interested in numerical linear algebra, sparse matrices, and parallel computing.

1. Introduction

Solution of large sparse linear systems of equations in the form $Ax = b$ constitutes a significant amount of the computations in the simulation of physical phenomena [1]. For example, the finite element discretization of a regular domain, with proper ordering of the variables x, renders a banded $N \times N$ coefficient matrix A. In a domain discretized by rectangular elements, each non-boundary node interacts with only its 8 neighbours as shown in *Fig. 1(top)*. Hence, in the corresponding A matrix (*Fig.*

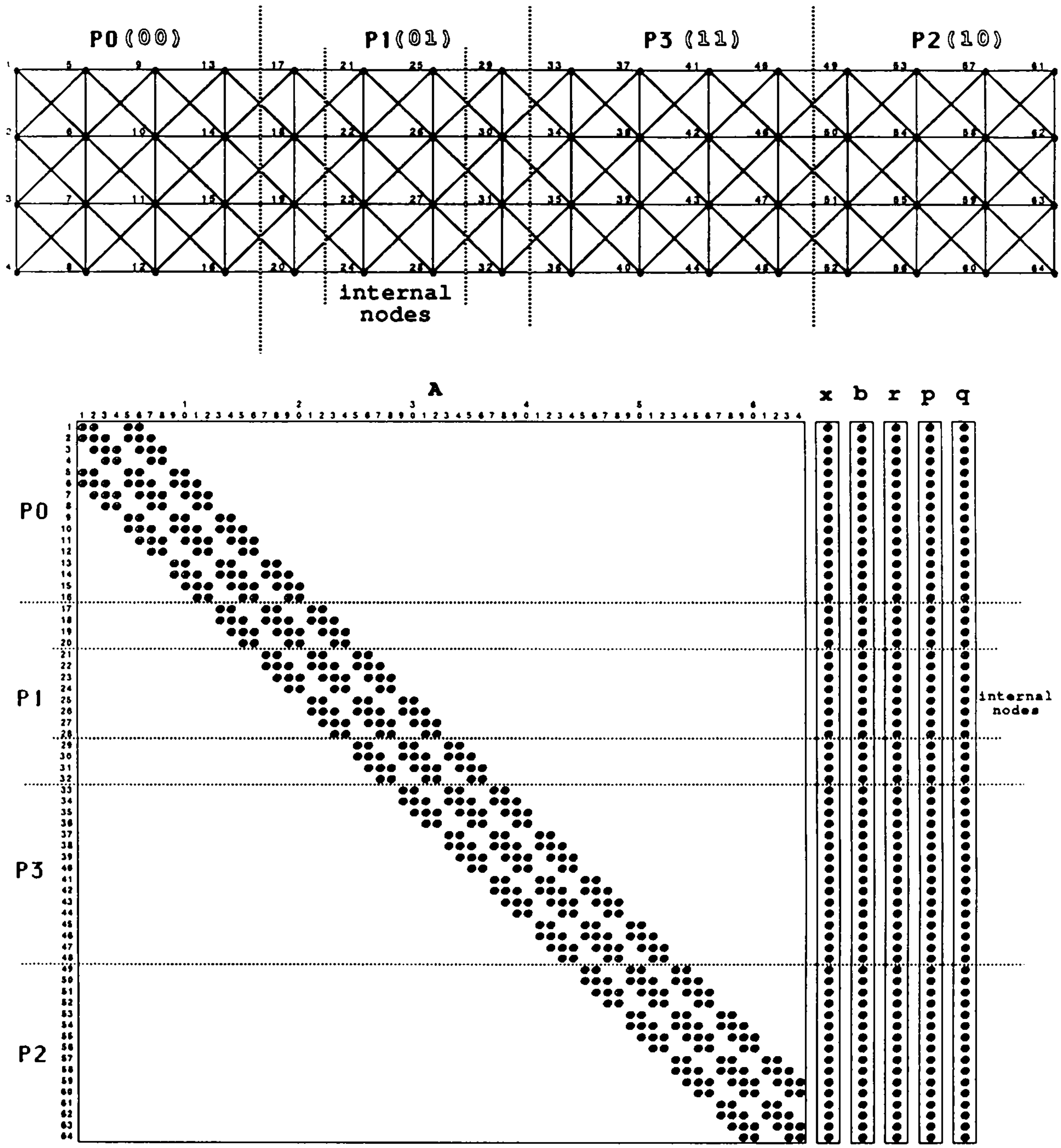


Fig. 1. Mapping of (*top*) a finite element domain and (*bottom*) the corresponding A matrix onto a 2-d hypercube.

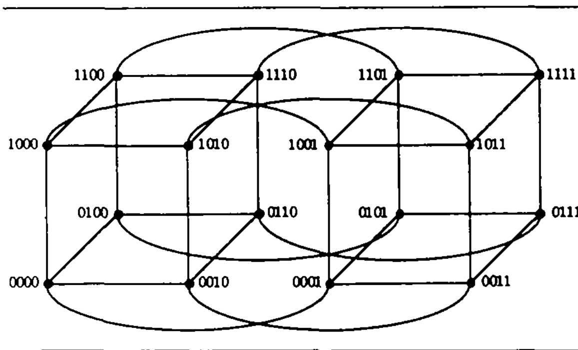


Fig. 2. 4-dimensional hypercube.

1(bottom)), there are at most 9 non-zero entries in a given row.

The Conjugate Gradient (CG) algorithm [2, 3] is an iterative method for solving sparse matrix equations that is widely used because of its convergence properties and can be parallelized on distributed memory multiprocessors [6, 5]. On the other hand, the computations in the CG algorithm consist mainly of matrix operations that can be vectorized. In this paper an implementation of the CG algorithm, for finite element simulation of metal deformation problems [7], that exploits both vectorization and parallelization is described. The machine used in this research was a hypercube multiprocessor with a vector processor attached to each node (iPSC-VX manufactured by Intel Scientific Computers).

In a hypercube [8] multiprocessor, each processor has its own local memory and processors communicate by exchanging messages. A d-dimensional hypercube consists of $p = 2^d$ processors (nodes) with a link between every pair of processors whose binary addresses differ in one bit. Thus each processor is directly connected to d other processors. A 4-dimensional hypercube, with binary encoding of the nodes is shown in *Fig. 2*. In a message passing multiprocessor, interprocessor communication speed is affected by the message set-up time (T_s) as well as the transmission time per byte (T_{tr}) and interprocessor communication time (T_{comm}) can be modeled as $T_{comm} = T_s + mT_{tr}$, where m is the number of bytes transmitted. The implementation described here achieves efficient parallelization by using a version of the CG algorithm suitable for *coarse grain* parallelism [4, 5] to reduce the communication steps required and by *overlapping* the computations on the vector processor with inter-node communication. With parallelization and vectorization, a speed-up of 58 over a μVax II is obtained on a two dimensional vector hypercube (iPSC-VX/d2), for large finite element meshes.

2. Parallelization of the conjugate gradient algorithm

2.1. The basic conjugate gradient algorithm

The computational steps of the CG algorithm are given below where, A is an N by N *sparse, symmetric*, and *positive definite*, coefficient matrix; x and b are the vectors of the unknown variables and right-hand sides respectively.

Initially, chose x_0 and let $r_0 = p_0 = b - Ax_0$; compute $<r_0, r_0>$. Then, for $k = 0,1,2,...$

1. *form* $q_k = Ap_k$
2. a. *form* $<p_k, q_k>$
   b. $\alpha_k = \dfrac{<r_k,r_k>}{<p_k,q_k>}$
3. $r_{k+1} = r_k - \alpha_k q_k$
4. $x_{k+1} = x_k + \alpha_k p_k$
5. a. *form* $<r_{k+1},r_{k+1}>$
   b. $\beta_k = \dfrac{<r_{k+1},r_{k+1}>}{<r_k,r_k>}$
6. $p_{k+1} = r_{k+1} + \beta_k p_k$

Here, r_k is the residual error associated with the trial vector x_k, i.e. $r_k = b - Ax_k$ which must be null when x_k is coincident with x^* which is the solution vector. p_k is the direction vector at the kth iteration. A suitable criterion for halting the iterations is $[<r_k,r_k> / <b,b>]^{\frac{1}{2}} < \varepsilon$.

The convergence rate of the CG algorithm can be improved by using a preconditioner. Powerful preconditioners such as the Incomplete Cholesky decomposition method have been proposed [9] and shown to significantly reduce the number of iterations for convergence. A simple preconditioning method, namely *scaling* was used here as the original structure of the CG algorithm is not disturbed by scaling and the computations required for scaling can be parallelized. In the Scaled CG algorithm, the rows and columns of matrix A are individually

scaled by its diagonal,

$D = \mathrm{diag}[a_{11}, a_{22}, \ldots, a_{NN}]$[10]. Hence,

$$\tilde{A}\tilde{x} = \tilde{b} \quad (2)$$

where $\tilde{A} = D^{-1/2} A D^{-1/2}$ with unit diagonal entries, $\tilde{x} = D^{1/2}x$ and $\tilde{b} = D^{-1/2}b$. Thus, b is also scaled and $\tilde{x}$ must be scaled back at the end to obtain x. Hence, in the SCG algorithm, the CG method is applied to Equation (2) obtained after scaling. The scaling process during the initialization phase requires only $\simeq 2 \times z \times N$ multiplications, where z is the average number of nonzero entries per row of the A matrix. Our numerical results show that symmetric scaling increases the convergence rate of the basic CG algorithm approximately by 50% for a wide range of sample metal deformation problems. In the rest of the paper, the scaled linear system will be denoted by $Ax = b$.

2.2. *Concurrent SCG algorithm on the hypercube*

The SCG algorithm has three types of operations: matrix vector product $q_k = Ap_k$, inner products $< r_{k+1}, r_{k+1} >$ and $< p_k, q_k >$, and the vector updates in steps 3, 4, and 6. All of these basic operations can be performed concurrently by distributing the rows of A, and the corresponding elements of the vectors b, x, r, p and q among the processors of the hypercube as shown in *Fig. 1*. With such a mapping, each processor is responsible for updating the values of those vector elements assigned to itself. In a system of equations obtained from a Finite Element Model, the row partitioning of the coefficient matrix corresponds to mapping a set of FE nodes onto each processor. Mapping schemes applicable to irregular geometries and their communication requirements are analyzed in [5]. Although the regular narrow geometry shown in *Fig. 1* (*top*) is not typical of finite element problems, it is used here as a simple example to explain the parallelization scheme. The one-dimensional strip mapping [11] scheme (*Fig. 1* (*top*)), partitions the A matrix intro groups of rows corresponding to a number of consecutive finite element nodes, the number of partitions being equal to the number of processors and requires the least number of communication set-ups. Nearest neighbour communications are obtained by mapping the slices of the A matrix and the vectors onto a linear array of processors (*Fig. 1* (*bottom*)) ordered using the binary-reflected gray code. For the $q_k = Ap_k$ computation, all but the first and the last processors in the linear array have to perform four nearest neighbour communication steps per iteration to exchange p_k values with left and right neighbours. Note that, under perfect load balanced conditions (i.e. $n = N/p$ variables mapped to each processor), these four one hop communications are performed concurrently in the hypercube. *Scaling* is also performed concurrently at the very beginning.

To perform the distributed inner products in Steps 2a and 5a, processors concurrently compute the partial sums corresponding to their slices of the vectors. Then the inner product value is accumulated, from these partial sums, in a selected root processor using the *Global Sum* (GS) algorithm [5], which requires d concurrent nearest neighbour communication steps. At the end of the GS communication step, the root processor calculates and passes the updated values for the global scalars α and β to all the other processors using the *Global Broadcast* (GB) algorithm [12], which also requires d concurrent nearest neighbour communication steps. The distributed vector update(s) in Steps 3, 4 and 6 can be performed concurrently without interprocessor communication, after each processor receives the updated global scalar value β (α).

The two inner product computations degrade performance in an architecture supporting coarse grain parallelism, because of the high set-up cost for each communication step. New formulations of the CG algorithm have been proposed to overcome the inner product dependencies on shared memory multiprocessors [13] and on the Cray X-MP [14] for parallel computation of inner products. This latter formulation is more suitable for a coarse grain parallel implementation, on a hypercube since the two inner products can be accumulated and distributed in the same GS-GB communication step [5]. The steps of the *coarse grain parallel* SCG algorithm (CG-SCG) can be given as follows:

Choose x_0, let $r_0 = p_0 = b - Ax_0$ and compute $< r_0, r_0 >$.

Then, for $k = 0,1,2,\ldots$

1. *form* $q_k = Ap_k$
2. *form* $< p_k, q_k >$ *and* $< q_k, q_k >$
   (in one GS – GB communication step)

3. a. $\alpha_k = \dfrac{<r_k,r_k>}{<p_k,q_k>}$

b. $\beta_k = \alpha_k \dfrac{<q_k,q_k>}{<p_k,q_k>} - 1.$

c. $<r_{k+1},r_{k+1}> = \beta_k <r_k,r_k>$

4. $r_{k+1} = r_k - \alpha_k q_k$

$x_{k+1} = x_k + \alpha_k p_k$

$p_{k+1} = r_{k+1} + \beta_k p_k$

The parallelization of the other computations is identical to the scheme described for the basic SCG (B-SCG) algorithm. Our numerical results for a wide range sample problems show that the proposed algorithm introduces no numerical instability and it requires exactly the same number of iterations to converge as the B-SCG algorithm. A more extensive study and results are given in [5].

3. Implementation of the CG-SCG algorithm on the iPSC-VX vector hypercube

The sparse matrix-vector product (Step 1), inner products (Step 2) and vector updates (two of them are DAXPY's in BLAS notation [15]) (Step 4) performed at each iteration of the CG-SCG algorithm are very suitable for vectorization. A vector processor (VP) board manufactured by *Sky Computer Inc.* is tightly coupled to each node processor of the iPSC-VX (vector extension) via MULTIBUS IIiLBX. *Fig. 3* illustrates the basic architecture of an iPSC-VX computational node. The 80286-based node processor board serves as a general purpose microcomputer. It contains 512 Kbytes of local memory and hosts a small message-based node executive called NX. The node processor with its NX is primarily responsible for coordinating message traffic into and out of the node, for scheduling and executing user processes and for controlling its companion VP. Another feature of the iPSC-VX node architecture is the dual-ported access to the memory on the VP board, which is shared between the node CPU and the VP. All user data is placed on the VP board where it is accessible to both the 80286 and the VP.

Fig. 3 also shows the steps to receive/send data into/from a node for processing by the VP. A message sent from an adjacent node is received over one of the serial communication ports and is automatically deposited in a message buffer. If a request for the message is pending (or is made some time later), NX will then transfer the data from the message buffer which resides on the node memory to the buffer on the VP memory indicated by the requesting user process. The computational results can be sent to other nodes following a similar sequence of events. Hence, node-to-node communication operations supported by NX on the node processor can be effectively *overlapped* by the mathematical operations performed on the VP board. The next section describes how this feature can be exploited to increase the performance of the parallel implementation.

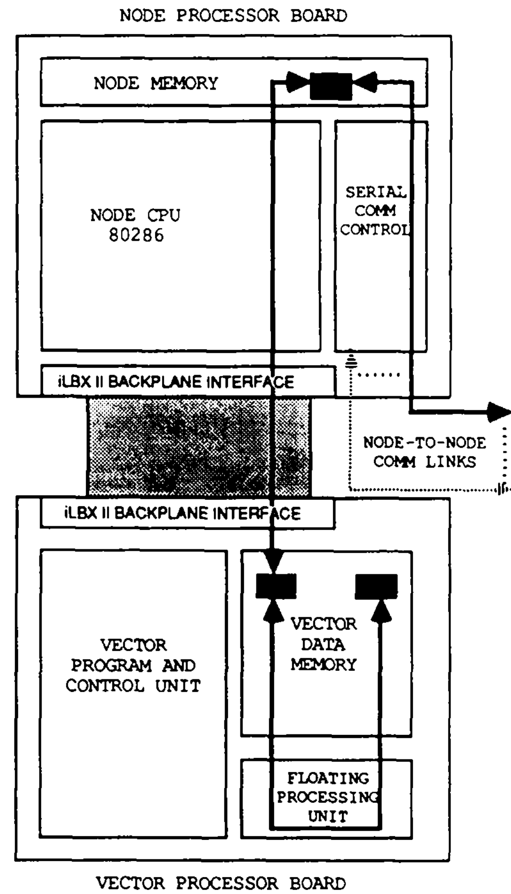


Fig. 3. iPSC-VX computational node.

3.1. *Overlapping communication and computation in the CG-SCG algorithm*

The FE nodes mapped to a processor can be

grouped as *internal* nodes and *boundary* nodes. *Internal* nodes are not connected to any FE node mapped to another processor. *Boundary* nodes are connected to at least one FE node which is mapped to another processor. For example, in *Fig. 1* (*top*), FE nodes 21–28 are the *internal* nodes, and FE nodes 17–20 and 29–32 are the *boundary* nodes mapped to processor P_1. The sparse matrix vector product computation for updating the elements of the vector q_k corresponding to the *internal* FE nodes, does not require any elements of the vector p_k which are mapped to other processors. For example, the column indices of the non-zero entries in rows 21–28 of the coefficient matrix (in *Fig. 1* (*bottom*)) corresponding to the internal FE nodes are between 17 and 32, which are also the indices of the elements of the vector p_k mapped to processor P_1. The *internal* sparse matrix-vector product computations performed on the VP can be effectively *overlapped* with the four nearest-neighbour communication steps performed by NX on the node board. Each processor can initiate the sparse matrix vector product corresponding to its *boundary FE* nodes on the VP only after its node board completes the local communication steps.

3.2. Comparison of overlapped and non-overlapped schemes

Fig. 4 shows the percentage improvement in performance, η, obtained by *overlapping* where η is defined as:

$$\eta = \frac{T_{\text{nonoverlap}} - T_{\text{overlap}}}{T_{\text{overlap}}} \cdot 100\%. \qquad (3)$$

Here, T_{overlap} is the solution time (per iteration) of the *overlapped* CG-SCG algorithm and $T_{\text{nonoverlap}}$ is the solution time of the *non-overlapped* CG-SCG algorithm on (iPSC-VX/d1-d2). In the six problems used to test the algorithms, the linear equations are those obtained in simulating deformations in metal-forming by using the finite element method. Note that, η on the iPSC-VX/d1 decreases as the size of the problem increases. This is because the computational time for the *internal* sparse matrix-vector product on each VP board is larger than the local communication time required for the *boundary* sparse matrix-vector product, even for small prob-

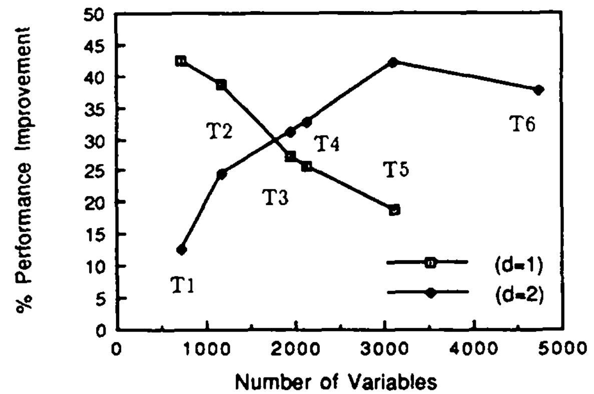


Fig. 4. Improvement in performance by overlapping computation and communication.

lems. However, the computational load of each VP is reduced by a factor of two on the iPSC-VX/d2 for problems of the same size. The number of concurrent nearest neighbour communications required for the distributed *boundary* sparse matrix-vector product computation is four in iPSC/VX/d2 compared to two in iPSC-VX/d1. The local communication time is greater than the *internal* sparse matrix vector product computation time for each test problem except for the largest one (*T6*). Hence, η on the iPSC-VX/d2, increases as the problem size increases for the first 5 test problems (*T1*-*T5*) and then decreases for the largest problem *T6*. As seen in *Fig. 4*, overlapping local communications with computation in the CG-SCG algorithm yields a substantial performance improvement of 13% to 44% on a two dimensional vector hypercube iPSC/VX/d2.

Since the NX requires the user data requested for communication to be in contiguous memory locations, each processor has to perform two vector *gather* operations to collect the most recently updated values of the right and left *boundary* elements of its p_k vector to two user communication buffers. Similarly, each processor has to perform two vector *scatter* operations to insert the elements of the p_k vector elements received from its two neighbours to the appropriate locations in its own data structures. In order to avoid the computational overhead required for communication, each processor *reorders*, in parallel, the active degrees of freedom at its *boundary* and the active degrees of freedom that it

has to receive from its two neighbours, so that they are allocated contiguous memory locations.

3.3. Architectural features of the iPSC-VX vector processor

On the VP board, a number of scalar hardware units with very low number of pipeline stages are provided to achieve functional parallelism. The independence of these functional units makes possible their parallel operation whenever their operands are functionally independent. *Fig. 5* illustrates iPSC-VX VP organization. The arithmetic unit in the VP consists of an adder and a multiplier each having two pipeline stages. *Double Precision* (DP) floating point multiplication is performed iteratively in four cycles by the 32 × 32 multiplier array. The result of a DP floating point multiplication is available at the output port, 5 cycles after initiating the multiplication and a new multiplication can be initiated only 3 cycles after the initiation of the previous one. The result of a DP floating point ALU operation is available on its output port, 3 cycles after initiating the operation and a new ALU operation can be initiated on any clockcycle after the initiation of the previous one. The output of the multiplier can be *chained* to both of the two input ports of the ALU. When the result of the multiplier unit is the operand to the ALU unit, *chaining* permits the results of the former functional unit to be transmitted directly to the ALU as they come out of the pipe.

3.4. Microprogramming techniques on iPSC-VX VP

Parallelism on the iPSC-VX vector processor can be achieved in two ways [16]: *overlapped* and *pipelined* programming. In *overlapped* programming, parallelism among the independent hardware units is exploited whenever the operands of these units are functionally independent. Sequences of dependent computations are examined to find the longest or *critical* path and then all the shorter paths are *overlapped* with the critical path. The *overlapped* method is usually applied to scalar computations and the initializations required during the *start-up* of a *pipelined* computation.

The *pipelined* method is usually applied to vector computations. In *pipelined* programming, independent computations on the sequence of vector elements are pipelined, so that different functional units operate on different vector elements, in a way similar to a hardware pipeline. In the microprogram loop that controls the vector computations, the computations in one iteration of the high level

Fig. 5. iPSC-VX vector processor (VP) architecture.

	Stage-1(S1)	Stage-2(S2)	S3	Stage-4(S4)	Stage-5(S5)
μ1	R5=R5+R6,j=MEM (Read ACOL(j))	M10=Aj (ld M10 with A(j))		A00=PROD— AjPj (chain prod to ALU) s=A00 .+D. A10 (initiate DP add)	A10=ALUR— s (feed partial sum (back to ALU)
μ2	R3=R3+R4,Aj=MEM (Read A(j)) ENFDB (enable FBRB)	M00=Pj (ld M00 with Pk(ACOL(j)) AjPj=M00 .*D. M10 (initiate DP mult.)			
μ3	R0=FBACK,Pj=MEM (Read Pk(ACOL(j))				JRS P2 (loop test)

$R3$ and $R5$ initially hold the base addresses of A and ACOL, respectively.

$R4$ and $R6$ hold the *strides* for the arrays, A and ACOL, respectively.

Fig. 6. Microcode in VP assembler for the inner loop of sparse matrix-vector product with $S = 5$.

language loop corresponding to the vector computations are overlapped with computations on preceeding and succeeding iterations. Each microinstruction operates on several vector elements in different stages of computation. Thus, for example, arithmetic operations can be carried out on one set of vector elements while another set is being fetched. Obviously this can only be done if the functional units and paths used in these different stages are independent. This process is called a *software pipeline* [16, 17] because of its similarity to the hardware pipeline concept used in conventional vector supercomputers. *Fig. 6* shows the microcode in VP assembler for a 3 microinstruction, 5-stage pipelined loop corresponding to the inner loop of matrix-vector multiplication, which will be explained in detail in the next section. In this example, each microinstruction is conceptually partitioned into 5 stages, where each stage controls operations on a different iteration.

The lower bound on the minimum number of microinstructions in the pipelined loop ($I_{\min}$), is determined by the most frequently used functional unit called the *critical* unit. The number of times around the pipelined microcode loop required to complete the entire computation on one element of a vector is the number of stages S in the *software pipeline* which determines the *start-up* overhead required to fill the *pipeline*. The following four step procedure given in [16] is used to construct the tightest *software pipeline* loop for a vector computation.

Step 1. Estimate the critical path length L by programming the computation on one element of a vector using the overlapped method.

Step 2. Find the critical functional unit or interconnect-bus to determine $I_{\min}$.

Step 3. Generate the pipelined loop microcode from the overlapped microcode by trying to fold the overlapped microcode back onto itself every $I_{\min}$ microinstructions. If unavoidable conflicts arise let $I_{\min} = I_{\min} + 1$ and repeat *Step 3*. Then, set $S \simeq \lceil L/I_{\min} \rceil$.

Step 4. Generate (S-1) prelude and (S-1) postlude sections.

Prelude and *postlude* sections are required to *fill* and *flush* the software pipeline, respectively. It should be noted that it may not always be possible to achieve the lower bound $I_{\min}$ due to reasons such as interconnect-bus conflicts, availability of a limited number of registers in the arithmetic units (especially for DP operations) and limitations introduced during DP multiplication ($I \geqslant 3$).

3.5. *Vectorization of the CG-SCG algorithm on the iPSC-VX*

As described earlier, at each iteration step of the concurrent CG-SCG algorithm, each processor of

the hypercube has to perform the following vector operations: 1 sparse matrix-vector product (Step 1), 2 inner products (Step 2) and 3 vector updates (two of them are DAXPY's) (Step 4). The implementation of each of these operations on the VP and a detailed description of how the procedure described above is applied to the sparse matrix-vector product computation will be discussed next.

3.5.1 Row-wise sparse matrix vector product
The standard column index compressed data storage scheme which loops over the nonzero column indices of the sparse coefficient matrix is used here. Hence, the row-wise sparse matrix vector product can be expressed in Fortran by the following double nested loop.

```
do i = 1,n
   start = AROW(i)
   last = AROW (i + 1) − 1
   sum = 0.
   do j = start, last
      sum = sum + A(j) × Pk(ACOL(j))
   end do
   Qk(i) = sum
end do
```

The nonzero entries of the sparse coefficient matrix are stored in completely compressed form in the *A* array. The column indices for the nonzero elements are stored in the one dimensional array ACOL and are used to index the corresponding elements of the array *Pk* for the multiplication in the inner loop. AROW is a pointer array and AROW(*i*) points to the first nonzero element of row *i* of matrix *A*, in array *A* and its column index in ACOL. This scheme can be implemented on the VP by converting the column indices in array ACOL into absolute addresses of the corresponding elements of the array *Pk*.

The following three microinstruction sequence can be used to read *Pk* (ACOL(*j*)) from the Data Memory (DM).

R5 = R5 + R6, absaddr = MEM; (read next address from DM pointed by R5)
ENFDB; (FBRB is loaded from DM-BUS, enable FBRB onto L-BUS on next cycle)
R0 = FBACK, pj = MEM; (read Pk(ACOL(j)) from DM pointed by R0)

Note that, the RALU register R5 initially holds the base address of the array ACOL, and R6 holds the stride of this array. The microoperation R0 = FBACK in the third microinstruction indicates that the RALU register R0 is loaded from the L-BUS. The four step procedure described in Section 3.4 is applied to pipeline the inner loop of the sparse matrix vector product as follows:

Step 1. Overlapped code.

1.	process address of ACOL(j) in RALU and initiate read from DM
2a.	process address of A(j) in RALU and initiate read from DM
b.	FBRB loaded with ACOL(j) from DM-BUS (default)
c.	enable FBRB onto L-BUS on next cycle
3a.	load ACOL(j) from L-BUS into a RALU register and initiate red from DM
b.	register MDR loaded with A(j) from DB-BUS (default)
4a.	load A(j) from register MDR into multiplier register M10 via A-BUS
b.	register MDR loaded with Pk(ACOL(j)) from DM-BUS (default)
5a.	load Pk(ACOL(j)) from MDR into multiplier register M00
b.	initiate DP multiplication of M00 by M10(A(j) × Pk(ACOL(j))
6-9	DP multiplication continues in the multiplier (default)
10a.	chain product (PROD) into ALU register A00
b.	initiate DP addition of A00 by A10 which holds the partial sum
11-12	DP addition continues in the ALU
13a.	feed new partial sum (ALUR) back to ALU register A10
b.	decrement PS counter, test, and branch to the beginning of loop

Therefore, the *critical* path length is $L = 13$ for the overlapped code.

Step 2. Functional unit usage is as follows; 3 RALU operations, 3 DM reads; Multiplier, ALU, L-BUS are used once and A-BUS is used twice. RALU, DM and DM-BUS that are used three times are the critical units, and therefore $I_{min} = 3$. In other words, at least 3 microinstructions are required for the inner loop.

Step 3. Success is obtained during the first folding process with $I_{min} = 3$ resulting in $S = \lceil L/I_{min} \rceil = 5$ stages. The 3 microinstruction pipelined loop written in VP assembler is given *Fig. 6*. Note that the fifth stage can be avoided in this loop by performing the first and the only microoperation (other than the loop test microoperation), $A10 = \text{ALUR} \rightarrow s$ (13.a in the overlapped code) on the first microinstruction of the previous (fourth) stage (*Fig. 7*). However, one stage (three clockcycles) early assertion of this microoperation requires the initiation of an ALU operation to initialize the partial sum to zero on the first cycle of the third (last) *prelude* section. This is achieved by loading one of the unused registers of the ALU with a DP zero during the *start-up* and then initiating an ALU bypass operation on the first microinstruction of the last *prelude* section. Note that, the sequencer operation which performs the loop test on 13.b of the overlapped code is shifted accordingly, to the last micro-intruction of the loop.

Step 4. As $S - 1 = 3$, the *prelude* and the *postlude* sections with three cycles each are developed. *Fig. 8(a)* shows the microcode structure with the prelude and postlude sections.

On the last clockcycle of the *postlude* section, the final sum value corresponding to the result for $Qk(i)$ is latched to the output register of the ALU. Hence, it takes two cycles to store the result on the DM via FIFO thus yielding a *critical* path length of 11 clockcycles for the *tail* section. Before entering the *pipelined* inner loop, one of the counters in the PS should be loaded with the length of row-i which can be computed from AROW($i+1$) − AROW(i). Most of the microinstructions in the *prelude* section use the RALU and hence the microinstructions of the *start-up* section, which require six RALU operations, cannot be overlapped efficiently with the *prelude* section. The length of the *critical* path in the overlapped code for the *start-up* section is 11 microinstructions and only the last two microinstructions can be overlapped with the first two microinstructions of the *prelude* section. This overlapped outer loop structure is depicted in *Fig. 8(b)*.

For vectors of length n, vector operations take a multiple of n clockcycles and an additional *start-up* time which can be neglected when n is sufficiently large. A period of n clockcycles is called a *chime* [18]. Thus the number of *chimes* (c), corresponds to the number of microinstructions (I) in the *pipelined* loop. Hence, the total number of clockcycles per iteration of the outer loop or the number of *chimes* is:

$$\begin{array}{llllll} c(z) \sim 9 + & 3 \times 3 & + 3 \times (z-3) + & 3 \times 3 & + 2 \\ \textit{startup} & \textit{prelude} & \textit{inner loop} & \textit{postlude} & \textit{tail} \end{array}$$

$$c(z) \sim 3 \times z + 20 \text{ clockcycles}$$

where z is the average number of non-zero entries per row of matrix A. The *start-up* and *prelude* sections of any iteration of the outer loop can be pipelined with the *postlude* and *tail* sections of the previous iteration since the operations required in the former sections do not depend on the results ob-

S1	S2	S3	S4
R5=R5+R6,j=MEM (Read ACOL(j))	M10=Aj (load M10 with A(j))		A00=PROD→ AjPj (chain product to ALU) A10=ALUR→ s (feed sum back to ALU) s=A00 .+D. A10 (initiate DP addition)
R3=R3+R4,Aj=MEM (Read A(j)) ENFDB (enable FBRB)	M00=Pj (load M00 with Pk(ACOL(j)) AjPj=M00 .*D. M10 (initiate DP mult.)		
R0=FBACK,Pj=MEM (Read Pk(ACOL(j))			JRS P2 (loop test)

Fig. 7. Microcode in VP assembler for the inner loop of sparse matrix-vector product with $S = 4$.

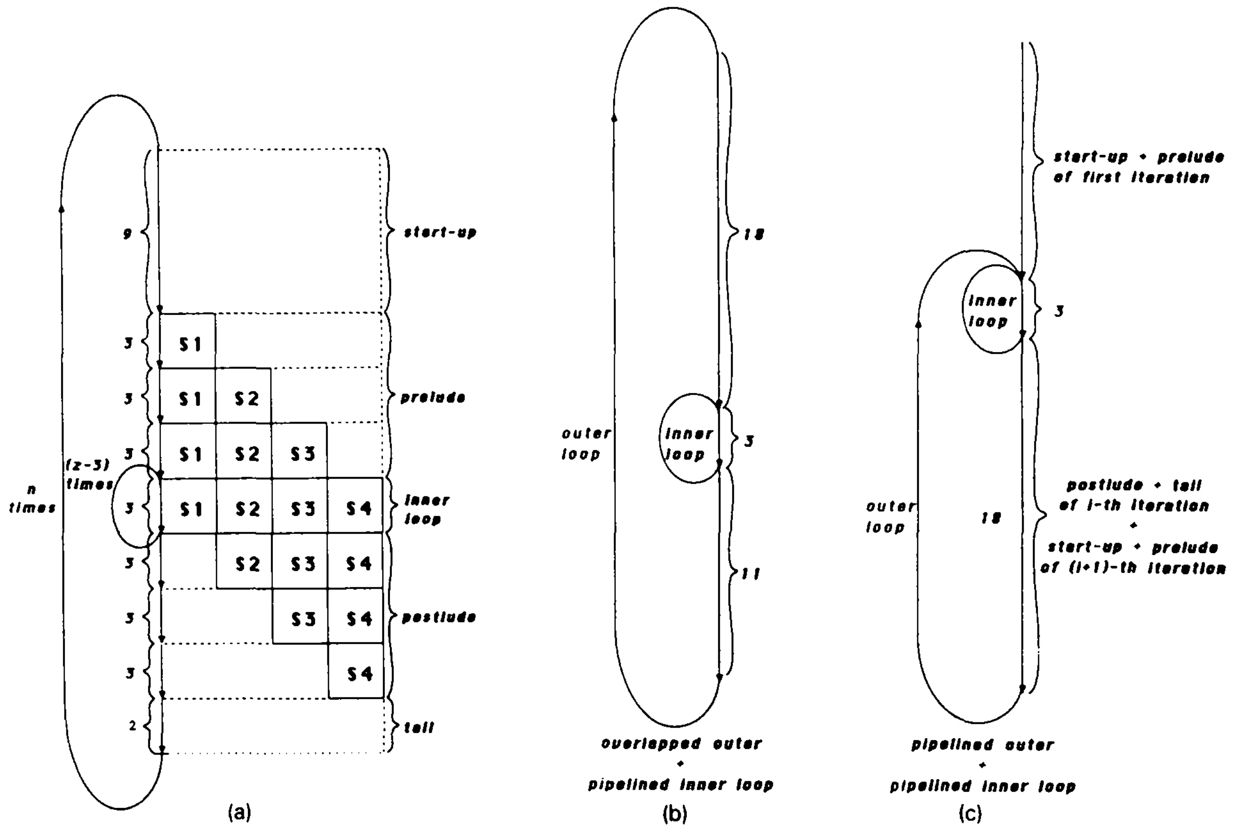


Fig. 8. (a) Microcode structure for sparse matrix-vector product, (b) Pipelined inner and overlapped outer loop, (c) Pipelined inner and pipelined outer loop.

tained at the end of the latter sections. Hence, the microoperations in the *start-up* section of the $(i+1)$-th iteration of the outer loop which make heavy use of the RALU and PS are successfully overlapped with the microoperations in the *postlude* section of the ith iteration which only uses the arithmetical units. The number of *chimes* for this *pipelined* outer loop scheme shown in *Fig. 8(c)* is:

$$c(z) = 3 \times (z - 3) + 18 = 3 \times z + 9. \qquad (4)$$

Hence, for $n = N/p$ variables mapped to a VP board, it takes $\sim (3z + 9) \times n$ clockcycles since the initial *start-up* overhead for the outer loop and all the other system overheads can be neglected for sufficiently large n. In general, $n_{1/2} \sim 30$ for the VP of the iPSC-VX, where $n_{1/2}$ indicates the number of elements in the dense column vector required to reach the half of the peak performance of the outer loop. The constant 9 in Equation (4) indicates the total number of overhead cycles per iteration of the loop.

The execution time per iteration of the outer loop ($c_t(z)$) can be calculated as follows: There are two 64-bit and one 32-bit operand DM read microoperations in the 3-cycle inner loop. For $n \leqslant 640$, the elements of the frequently referenced column array Pk can be allocated in the Static RAM (SRAM), which has an access time of 100ns as compared to the 250ns access time for DP operands for the Dynamic RAM (DRAM) (*Fig. 5*). Thus, a single iteration of the inner loop which performs 2 floating point operations (one multiply + one add), takes 550ns resulting in a peak performance of $2/0.55 = 3.64$MFLOPS. The overhead section has 7 fetches from SRAM and one 64-bit and one 32-bit operand fetch from the DRAM. Therefore the execution time for this section is 1150ns. Hence, the execution time per iteration of the outer loop as a function of z is

$$c_t(z) = 0.55 \times z + 1.15\mu s. \qquad (5)$$

Since each iteration of the outer loop involves $2z$ floating point operations to compute the inner product of each row of matrix A and the column vector Pk, the estimated performance, P_E, as a function of z can be calculated from:

$$P_E(z) = \frac{2z}{c_t(z)}\text{MFLOPS} = \frac{2z}{0.55 \times z + 1.15}\text{MFLOPS} \quad (6)$$

for Pk totally allocated in SRAM. The value of $z_{1/2}$, the average number of non-zero entries per row of A required to reach half of the peak performance of the inner loop can be calculated from:

$$\frac{1}{0.55} = \frac{2z_{1/2}}{0.55 \times z + 1.15} \quad (7)$$

as $z_{1/2} \sim 2.1$. This low value for $z_{1/2}$ is achieved by exploiting the parallelism at both levels and short pipe lengths of the functional units. For $n \geqslant 640$ the frequently used array Pk cannot be totally allocated in the SRAM due to the size limitations and the inner loop execution time increases to 700 ns, for Pk totally allocated in DRAM, resulting in a peak performance of 2.86MFLOPS. The expression for $c_t(z)$ in this case becomes $c_t(z) = 0.7 \times z + 1.15\mu s$ and the estimated performance is:

$$P_E(z) = \frac{2z}{0.70 \times z + 1.15}\text{MFLOPS} \quad (8)$$

with $z_{1/2} = 1.6$.

3.5.2 Comparison with diagonal-wise sparse matrix vector product

Sparse matrix vector products on vector supercomputers using special vector units with long pipeline lengths are carried out in a *diagonal-wise* fashion for coefficient matrices arising from finite difference or finite element discretizations [19]. However, this scheme requires a banded A matrix. For example, the coefficient matrix A in *Fig. 1* has 9 dense diagonal strips ($\sigma = 9$). The inner loop in a *diagonal-wise* sparse matrix vector product is the accumulation of the product of two dense vectors (diagonal strip vectors of A and the column vector Pk) of sizes nearly equal to n. The outer loop is iterated only σ times. Since, the inner loop is iterated almost n times, the overhead during σ iterations of the outer loop can be neglected for sufficiently large n. $I_{\min} = 4$ for the inner loop since 3 reads (one for the strip vector, one for the column vector and one for the partial sum vector) and one write operation (for partial sum vector) are required in the inner loop. Hence, $I = 5$ is found because of the limitation introduced during the DP multiplication. Thus, *diagonal-wise* sparse matrix-vector product takes $\sim (5 \times n) \times \sigma$ clockcycles with $c(\sigma) = 5 \times \sigma$. Hence, $c_t(\sigma) = 0.8\sigma\mu$secs, when the partial sum vector can be totally allocated in the SRAM. Since, $\sigma = z$ for such A matrices, equating the expression for $c_t(\sigma)$ and $c_t(z)$ for *diagonal-wise* and *row-wise* schemes, $z = 4.6$ is obtained which indicates that the *diagonal-wise* scheme gives a better rate of *sustained* performance only for $z < 4.6$. The expression $c_t(\sigma)$ given for the *diagonal-wise* scheme is a very optimistic estimate since finite element or difference discretizations of regions with appendages and holes yield a much larger number of strips and a considerable overhead is associated with finding an ordering of the A matrix to find a near minimal number of strips. Hence, the *row-wise* sparse matrix vector product scheme was chosen for microcoding, since $\sigma = z = 18$ in our sample FE problems and since physical domains arising in metalwork simulation are very irregular with appendages and holes and the row-wise scheme does not require a banded structure.

3.5.3 Innerproducts

The pipelined loop for the inner product is microcoded by following the four step procedure given in Section 3.4. The estimated *critical* path $L = 12$ is obtained from the *overlapped* microcode. The *critical* functional units are RALU, DM, and A-BUS which are used twice and thus $I_{\min} = 2$. However, this lower bound is not achieved during the first *folding* process due to the limitations encountered in DP multiplication as indicated in Section 3.4. Success is obtained during the second folding process with $I_{\min} = I_{\min} + 1 = 3$ with $S = 4$. For the innerproduct, $< p_k, q_k >$ in Step 2 of the SCG algorithm, each iteration of the pipelined loop takes $c_t = 450ns$, resulting in a rate of 4.44MFLOPS, since there are two reads from the DM and one of the vectors, p_k, is stored in SRAM. Similarly, for the innerproduct $< q_k, q_k >$, $c_t = 450ns$ resulting in 4.44MFLOPS, since there is only one read which is from the DRAM.

3.5.4 Vector updates

Two of the vector updates are of the form $y = y + \alpha x$ which are called DAXPY's, in BLAS terminology and one is of the form $y = x + \beta y$. There is no difference in their performance in terms of clockcycles. The *critical* path length is estimated as $L = 12$ from the overlapped code. The *critical* fuctional units are the RALU, DM and A-BUS which are used three times, hence $I_{\min} = 3$. Success is obtained in the first folding process with the lower bound. For the DAXPY operation $r_{k+1} = r_k - \alpha_k q_k$, $c_t = 750ns$ resulting in a rate of 2.67MFLOPS since all of the three 64-bit operands are stored in the DRAM. For the DAXPY operation $x_{k+1} = x_k + \alpha_k p_k$, $c_t = 600ns$ resulting in a rate of 3.33MFLOPS since the vector p_k is stored in the SRAM. For the vector update $p_{k+1} = r_{k+1} + \beta_k p_k$, $c_t = 450ns$ resulting in a rate of 4.44MFLOPS since the vector p_k, which is stored in the SRAM is referenced twice for read and write.

4. Experimental results

4.1. Performance of the microcoded sparse matrix vector product

Fig. 9 illustrates the estimated and measured performance of the microcoded sparse matrix vector product with respect to the number of nonzero entries per row of the sparse matrix. The sustained performance, P_s, in MFLOPS, for a particular z value is calculated from

$$P_s(z) = \frac{2zn}{T_{MVP}} \text{MFLOPS} \tag{9}$$

where T_{MVP} is the measured time for the multiplication of an $n \times n$ sparse matrix A having z nonzero entries per row with a dense column vector Pk of n elements. In these measurements n is typically chosen large enough to observe the effects of z on the performance. The measured performance is found to be within 4% of the estimated performance calculated from equations (6) and (8). It can be seen from *Fig. 9* that almost peak performance of the inner loop is achieved for very low z values. For $z \sim 17$, $\sim$ 3.15MFLOPS and $\sim$ 2.53MFLOPS, are attained when the Pk-array is totally allocated in

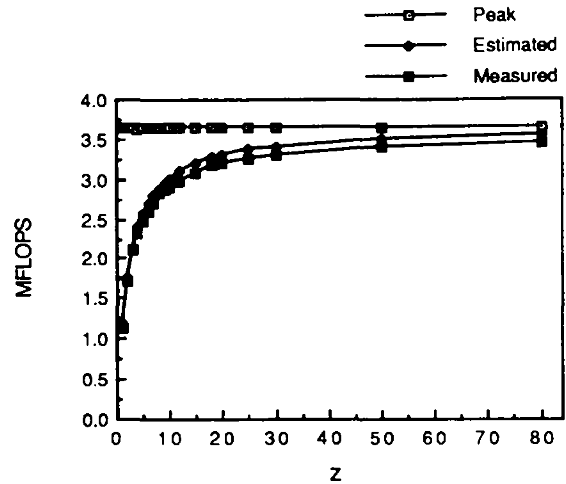


Fig. 9. Estimated and measured performance (MFLOPS) of the microcoded sparse matrix vector product as a function of z.

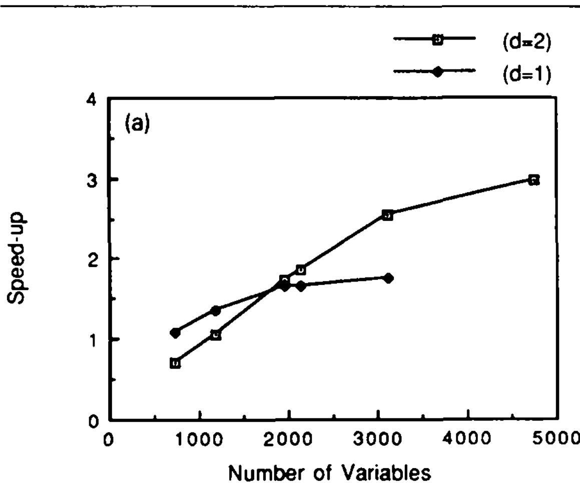
SRAM and DRAM, respectively. It should be noted here that the sustained performance of a node CPU (80286/80287) of the iPSC-VX is measured to be only $\sim$ 30KFLOPS for the sparse matrix vector product.

4.2. Overall performance

Table 1 presents solution times (per iteration) for the sequential Basic-SCG (B-SCG) algorithm on the μVAX II and iPSC-VX/d0, and the parallel and vectorized CG-SCG algorithm on the vector hypercube (iPSC-VX/d1-d2). The B-SCG algorithm is used on the μVAX II and a single node since the CG-SCG improves performance only for the parallel implementation. The six test problems *T*1-*T*6 are the linear systems of equations obtained in simulating deformations in metalforming by using the finite element method. Experimental speedup (S) and efficiency (e) obtained by parallelization are shown in *Figs 10* and *11* respectively, where $S = \frac{T_{seq}}{T_{par}}$, $e = \frac{S}{p}$; T_{seq} is the measured computation time on one processor and T_{par} is the measured parallel computation time on p processors. As expected, speedup and efficiency increase with increasing problem size. The efficiencies achieved on iPSC-VX/d1 for the

Table 1
Solution times (per iteration) and speedups for different size FE problems

Test Prob	Mesh size	Number of var.s	Number of Iters for conv.	BSCG μVAX II sol time per iter (ms)	BSCG d0 sol time per iter (ms)	CGSCG d1 sol time per iter (ms)	CGSCG d2 sol time per iter (ms)	Speedup w.r.t. μVAX II
T1	11 × 36	734	99	238.59	10.30	9.49	14.54	16.41
T2	25 × 25	1175	130	384.92	17.23	12.77	16.38	23.50
T3	32 × 32	1952	165	672.79	30.78	18.54	17.70	38.01
T4	33 × 33	2143	201	695.60	32.52	20.05	18.22	38.17
T5	40 × 40	3120	126	1021.83	-	28.73	19.84	51.50
T6	49 × 49	4752	297	1551.58	-	-	26.70	58.11



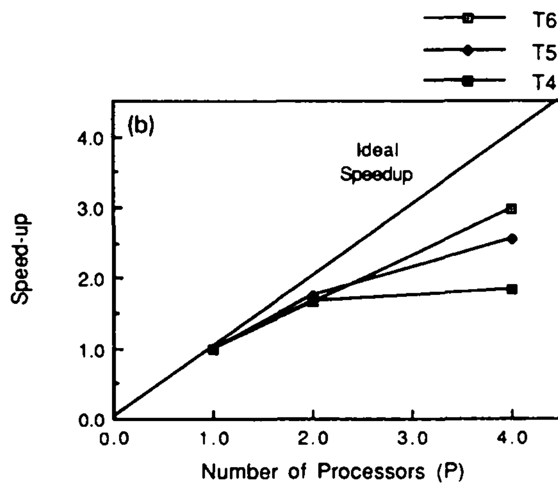


Fig. 10. Speedup by parallelization (*top*) as a function of problem size, (*bottom*) as a function of hypercube dimension.

three larger sample problems *T3*, *T4*, and *T5* are 83%, 84%, and 88%, respectively. Since the larger problems *T5* and *T6* could not be run on a single node because of memory problems, estimated values of the solution time on iPSC(VX/d0 are used for these problems to calculate the speedup and efficiency. These estimates, based on solution times of smaller problems, should be reasonable since there is no communication in the single node case. *T6* could not be run on the iPSC-VX/d1 and the solution time was not estimated since interprocessor communication time would be involved. The efficiency achieved on iPSC-VX/d2 for the largest sample problem (*T6*) is ~ 75%.

The last column in *Table 1* shows the speedup obtained on the iPSC-VX/d2 compared to the μVAX

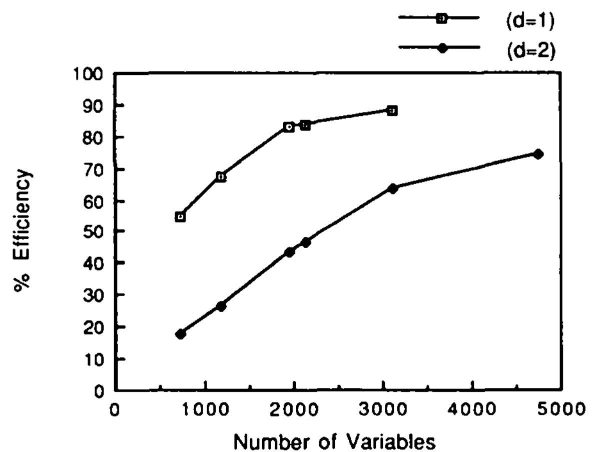


Fig. 11. Efficiency as a function of problem size.

Table 2
Performance for the B-SCG (μVAX II, iPSC-VX/d0), and the CG-SCG (iPSC-VX/d1-d2) for different size FE problems

Problem	μVAX perf. (MFLOPS)	VX/d0 perf. (MFLOPS)	VX/d1 perf. (MFLOPS)	VX/d2 perf. (MFLOPS)
T1	0.14	3.18	3.45	2.25
T2	0.14	3.11	4.24	3.30
T3	0.14	2.93	4.86	5.09
T4	0.14	2.91	4.95	5.46
T5	0.14	-	5.07	7.34
T6	0.14	-	-	8.40

II. A speed-up of 58 is obtained for $T6$. As stated before, the performance of a node CPU (80286/80287) of the iPSC-VX is ~ 30KFLOPS for the sparse matrix vector product and ~ 35KFLOPS for the CG-SCG algorithm. From *Table 2*, the performance of the μVAX II is 140KFLOPS for the CG-SCG algorithm. Therefore the speedup with respect to a single 80286/80287 processor with no vectorization is expected to be four times the speedups with respect to the μVAX II.

Table 2 presents the measured performance in MFLOPS of the sequential B-SCG algorithm (μVAX II, iPSC-VX/d0), and of the CG-SCG algorithm (iPSC-VX/d1-d2). The sustained performance, P_s, in MFLOPS, of the parallel and vectorized implementation for a particular test problem of size N is calculated from:

$$P_s(N) = \frac{2(z+5)N}{T_{\text{sol}}}\text{MFLOPS} \qquad (10)$$

where T_{sol} is the measured solution time per iteration in $\mu secs$ and $2(z+5)N$ is the total number of floating point operations in one iteration of the CG-SCG algorithm. *Fig. 12* illustrates the measured performance of the sequential B-SCG algorithm (μVAX II, iPSC-VX/d0) as a function of the problem size and of the parallel CG-SCG algorithm (iPSC-VX/d1-d2) both as a function of problem size and as a function of p, the number of processors.

The estimated peak performance is almost attained (3.18MFLOPS) on a single VP board for the smallest size sample problem $T1$. Most of the elements of the p-vector are allocated in the SRAM (640 out of 734) for this problem ($T1$). As seen from *Fig. 12*, the performance of the sequential SCG al-

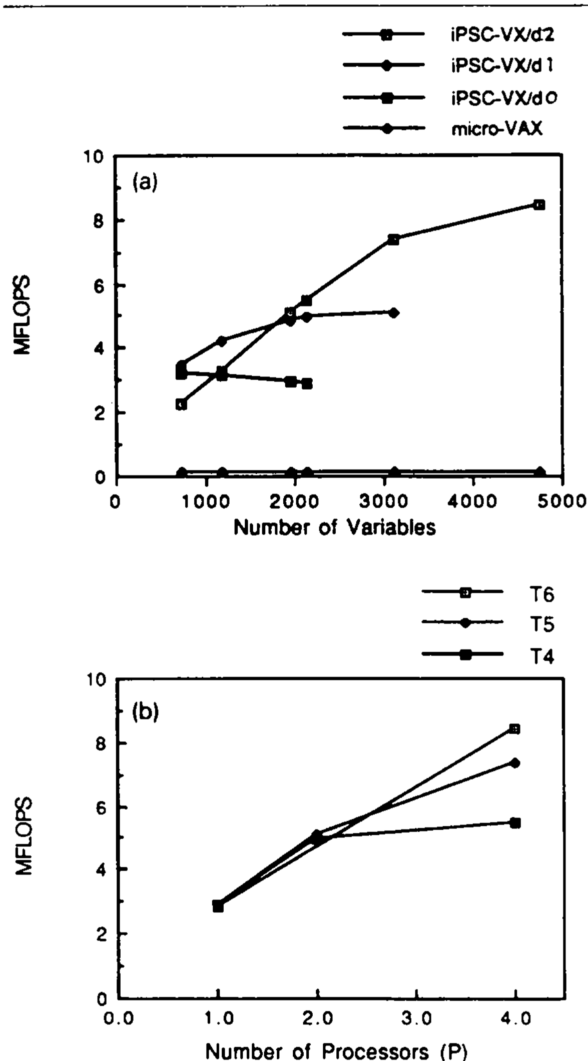


Fig. 12. Performance of the sequential B-SCG (μVAX II, iPSC-VX/d0) and the parallel CG-SCG (iPSC-VX/d1-d2) algorithms (*top*) as a function of problem size, (*bottom*) as a function of the number of processors.

gorithm on a single VP decreases slightly with increasing problem size since the portion of the p-vector allocated in SRAM decreases. A performance of 8.40MFLOPS is attained on the 4-node iPSC-VX/d2 vector hypercube for the largest sample problem $T6$.

Acknowledgements

This work was partially supported by the SBIR Program Phase II (F33615-85-C-5198) of Universal

Energy Systems Inc. with the Air Force Materials Laboratory, AFWAL/MLLM, WPAFB, Ohio 45433.

References

[1] R. Lucas, T. Blank and J. Tiemann, A parallel solution method for large sparse systems of equations, *IEEE Trans. Computer-Aided Design* CAD-6 (6) (November 1987) 981–990.

[2] M.R. Hestenes and E. Stiefel, Methods of conjugate gradients for solving linear systems, *Nat. Bur. Standards J. Res.* 49 (1952) 409–436.

[3] G.H. Golub and C.F. van Loan, *Matrix Computations* (Johns Hopkins Univerity Press, Baltimore, MD, 1983).

[4] G. Meurant, Multitasking the conjugate gradient method on the Cray X-MP/48, *Parallel Comput.* 5 (3) (1987) 267–280.

[5] C. Aykanat, F. Özgüner, P. Sadayappan and F. Ercal, Iterative algorithms for solution of large sparse systems of linear equations on hypercubes, *IEEE Trans. Comput.* c-37 (December 1988) 1554–1568.

[6] G.A. Lyzenga, A. Raefsky and G.H. Hager, Finite element and the method of conjugate gradients on a concurrent processor, in *ASME International Conference on Computers in Engineering* (1985) 393–399.

[7] C. Aykanat, F. Ozguner, S. Martin and S.M. Doraivelu, Parallelization of a finite element application program on a hypercube multiprocessor, in *Hypercube Multiprocessors 1987, SIAM, Philadelphia*, (1987) 662–673.

[8] C.L. Seitz, The cosmic cube, *Commun. Assoc. Comput. Mach.* 28 (1) (January 1985) 22-23.

[9] D.S. Kershaw, The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations, *J. Comp. Phys.* 26, (September 1978) 43–65.

[10] A. Jennings and G.M. Malik, The solution of sparse linear equations by the conjugate gradient method, *Internat. J. Numerical Methods Engineering* 12 (1978) 141–158.

[11] P. Sadayappan and F. Ercal, Nearest neighbor mapping of finite element graphs onto processor meshes, *IEEE Trans. Comput.* c-36 (December 1987) 1408–1424.

[12] C. Moler, Matrix computation on distributed multiprocessors, in *Hypercube Multiprocessors 1986, SIAM, Philadelphia* (1986) 181–195.

[13] J. Van Rosendale, Minimizing inner product data dependencies in conjugate gradient iteration, in *Proc. IEEE Internat. Conf. Parallel Processing* (August 1983) 44–46.

[14] Y. Saad, Practical use of polynomial preconditionings for the conjugate gradient method, *SIAM J. Sci. Statist. Comput.* 6 (4) (October 1985) 865–881.

[15] C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh, Basic linear algebra subprogram for fortran usage, *ACM Trans. Math. Software* 5 (3) (1979) 308–323.

[16] A. Charlesworth, An approach to scientific array processing; the architectural design of the AP-120B/FPS-164 family, *Computer* 14 (9) (September 1981) 18–27.

[17] H.C. Young, Code scheduling methods for some architectural features in PIPE, *Microprocessing Micrograming* 22 (1) (January 1988) 39–63.

[18] H.A. Van Der Vorst, The performance of Fortran implementations for preconditioned conjugate gradients on vector computers, *Parallel Comput.* 3 (1) (1986) 49–58.

[19] N.K. Madsen, G.H. Rodrigue and J.I. Karush, Matrix multiplication by diagonals on a vector/parallel processor, *Inform. Process. Lett.* 5 (2) (1976) 41–45.