11/3/78

CONF-780945--

LA-7491-C

Conference Proceedings

Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors

Held at Los Alamos Scientific Laboratory

Los Alamos, New Mexico

September 20-22, 1978

University of California

LOS ALAMOS SCIENTIFIC LABORATORY

Post Office Box 1663 Los Alamos, New Mexico 87545

This work was supported by the Director of the Applied Mathematical Sciences, US Department of Energy.

UNITED STATES
DEPARTMENT OF ENERGY
CONTRACT W-7405-ENG. 36

LA-7491-C
Conference Proceedings

Special Distribution
Issued: October 1978

Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors

Held at Los Alamos Scientific Laboratory

Los Alamos, New Mexico

September 20-22, 1978

Compiled by

B. L. Buzbee
J. F. Morrison

PREFACE

The 1978 LASL Workshop on Vector and Parallel Processors was held at the Los Alamos Scientific Laboratory, Los Alamos, New Mexico, September 20-22, 1978. Since successful utilization of vector and parallel processors presents a special set of problems, the purpose of the Workshop was to provide an exchange of information between organizations that either have acquired or plan to acquire these processors. Researchers from government laboratories, industry, and academia participated. Approximately 35 talks were presented covering operating systems, applications, algorithms, languages, and processors. Complete papers for many of the talks are contained in these Proceedings. In cases where the Proceedings contain only an abstract, you should contact the author if you want further information.

On behalf of the organizers, we thank the Workshop participants for their contributions. We also thank the Applied Mathematical Sciences Program, Office of Energy Research, U. S. Department of Energy for funding of the Workshop.

B. L. Buzbee

J. F. Morrison

AGENDA

1978 LASL WORKSHOP ON VECTOR & PARALLEL PROCESSORS

LOS ALAMOS SCIENTIFIC LABORATORY
LOS ALAMOS, NM 87545

SEPTEMBER 20-22, 1978

FULLER LODGE

HOSTS: *B. L. Buzbee, C-3 Group Leader*
J. F. Morrison, C-11 Group Leader

SEPTEMBER 20, 1978

COMMUNITY BUILDING

BANQUET

6:00-7:00 HAPPY HOUR! (NO HOST BAR)

7:00 DINNER

8:30 SPEAKER - *Neil Lincoln, CDC, "High-Speed Computing - You Take the High Road and I'll Take the Low Road and Someone Else Will Get There Before Us Both"*

SEPTEMBER 20, 1978

SESSION I - OPERATING SYSTEMS

8:15	WELCOME - *Kaye D. Lathrop, C-Division Leader*
8:20	ADVANCED HIGH LEVEL USER DIAGNOSTICS WITH THE ICL DISTRIBUTED ARRAY PROCESSOR - *Robin W. Gostick*
8:50	A HIGH PERFORMANCE GRAPHICS SYSTEM FOR THE CRAY-1 - *Robert H. Ewald, Lynn D. Maas*
9:20	CRAY-1 ERROR RECOVERY - *Alex Marusak*
9:50	THE CRAY-1 COMPUTER AND THE DEMOS OPERATING SYSTEM - *Forest Baskett*
10:20	COFFEE & ROLLS
10:35	THE ILLIAC IV SYSTEM IN 1978 - *David Stevenson*
11:05	CTSS STATUS REPORT - *Dieter Fuss*
11:35	EARLY EXPERIENCE WITH THE CRAY-1 AT NCAR - *Richard K. Sato*
12:05	LUNCH

SESSION II - APPLICATIONS

1:35	VECTORIZATION FROM A LARGE CODE POINT OF VIEW - *Margaret W. Asprey*
2:00	VECTORIZED PIC SIMULATION CODES ON THE CRAY-1 - *D. W. Forslund, C. W. Nielson*
2:30	A CRAY-1 SIMULATOR AND ITS APPLICATION TO DEVELOPMENT OF HIGH-PERFORMANCE CODES - *D. A. Orbits, D. A. Calahan*
3:00	LARC EXPERIENCE IN THE INSTALLATION OF A STAR-100 COMPUTER - *M. Rowe*
3:30	BREAK
3:45	SOME LINPACK TIMINGS ON THE CRAY-1 - *J. J. Dongarra*
4:15	STAR-100 - GOOD NEWS AND BAD NEWS - *T. Rudy*
4:45	HOW TO GET MORE OUT OF YOUR VECTOR PROCESSOR - *Brian Q. Brode*

SEPTEMBER 21, 1978

SESSION III - ALGORITHMS

8:15	POISSON SOLVERS ON A LARGE ARRAY COMPUTER - *Chester E. Grosch*
8:45	VECTORIZED FORTRAN SUBPROGRAMS FOR THE SYMMETRIC FAST FOURIER TRANSFORMS - *Paul N. Swarztrauber*
9:15	A PARALLEL ALGORITHM FOR SOLVING BONDED SYSTEMS ARISING FROM SECOND ORDER PARTIAL DIFFERENTIAL EQUATIONS IN TWO DIMENSIONS - *V. Ellel, D. Parkinson*
9:45	FINITE ELEMENT DYNAMIC ANALYSIS ON THE STAR-100 - *Jules J. Lambiotte, Jr.*
10:15	COFFEE & ROLLS
10:30	SOME VARIANTS OF METHODS FOR COMPUTING THE VARIANCE - *T. Chan, G. H. Golub, R. LeVeque*
11:00	A DIRECT POISSON SOLVER ON STAR - *M. J. Kascic*
11:30	VECTORIZATION OF BLOCK RELAXATION TECHNIQUES - SOME NUMERICAL EXPERIMENTS - *Daniel L. Boley*
12:00	LUNCH

SESSION IV - LANGUAGES

1:30	VECTORAL - A VECTOR ALGORITHMIC LANGUAGE FOR ILLIAC - *Alan Wray*
2:00	AN INTRODUCTION TO VECTRAN AND ITS USE IN SCIENTIFIC APPLICATIONS PROGRAMMING - *George Paul, M. Wayne Wilson*
2:30	REGISTER ALLOCATION IN THE SL/1 COMPILER - *Douglas D. Dunlop, John C. Knight*
3:00	ACTUS - A LANGUAGE FOR SIMD ARCHITECTURES - *R. Perrott, David Stevenson*
3:30	BREAK
3:45	THE VECTORIZER SYSTEM: CURRENT AND PROPOSED CAPABILITIES - *Mathew Myszewski*
4:15	AUTOMATIC STACKLIB FACILITIES IN STAR*FORTRAN - *Anil K. Lakhwara*
4:45	VECTORIZING FORTRAN - *Lee Higbie*

SEPTEMBER 22, 1978

SESSION V - PROCESSORS

8:15	COMPUTATIONAL FLUID DYNAMICS, ILLIAC IV, AND BEYOND - *K. G. Stevens, Jr.*
8:45	DAP - A FLEXIBLE NUMBER CRUNCHER - *S. F. Reddaway*
9:15	ARRAY PROCESSING ON THE PDP-10 - *Neil Maron, George G. Sutherland*
9:45	DISTRIBUTED NUMERIC COMPUTING ENGINES: MINICOMPUTER-BASED VECTOR AND PARALLEL COMPUTING - *W. Morven Gentleman*
10:15	COFFEE & ROLLS
10:30	DESIGN OF ARITHMETIC ELEMENTS FOR BURROUGHS SCIENTIFIC PROCESSOR - *Daniel D. Gajski, L. Rubinfield*
11:00	THE TEXAS RECONFIGURABLE ARRAY PROCESSOR - *Speaker to be announced*
11:30	OPTIMIZATION OF VECTOR OPERATIONS IN AN EXTENDED FORTRAN COMPILER - *Kenneth W. Kennedy, Jr.*

CONTENTS

PROCEEDINGS OF THE 1978 LASL WORKSHOP ON VECTOR AND PARALLEL PROCESSORS

Compiled by

B. L. Buzbee and J. F. Morrison

ABSTRACT

This is a compilation of papers and abstracts presented at the 1978 LASL Workshop on Vector and Parallel Processors held at the Los Alamos Scientific Laboratory, Los Alamos, New Mexico, September 20-22, 1978.

ADVANCED HIGH LEVEL USER DIAGNOSTICS WITH THE ICL DISTRIBUTED ARRAY PROCESSOR

by

Robin W. Gostick

International Computers Ltd.

London

ABSTRACT

The unique combination of a powerful bit organised parallel processor within the store of an advanced virtual storage serial computer allows a highly flexible and powerful diagnostic system with very low cost to the user.

The diagnostic system of the ICL 2900 series automatically provides high level post mortem information to the programmer regardless of the language or combination of languages being used and with absolutely no run-time overheads. This system has been enhanced to provide run-time diagnostics for use with the Distributed Array Processor (DAP) but with significant extensions to take advantage of the properties of the DAP and to overcome many of the problems usually associated with array processor diagnostic systems.

I. INTRODUCTION

General purpose computers have, over the last few years, provided increasingly powerful user diagnostic and debugging aids. Development of supercomputers, on the other hand, has tended to be towards production of super efficient compilers, with minimal debugging aids. The ICL DAP approach combines the facilities of the general purpose machine with the power of a supercomputer.

HARDWARE OVERVIEW

The DAP has been described in ref 1 , but the main features are summarised here. Figure (1) shows the basic construction of the DAP processing array, which consists of a 64 x 64 element matrix of simple one bit processors, each with a 4K bit storage element. The processors have both simple nearest neighbour connections and highway connections to the master control unit (MCU). The MCU performs all the instruction fetching and decoding for the system, and then broadcasts instructions to be executed simultaneously by all the processing elements, acting on their local data. One further level of control is provided by giving each processor a degree of autonomy, whereby it can select whether or not to obey certain instructions.

SECTION OF DAP ARRAY SHOWING BASIC INTERCONNECTIONS

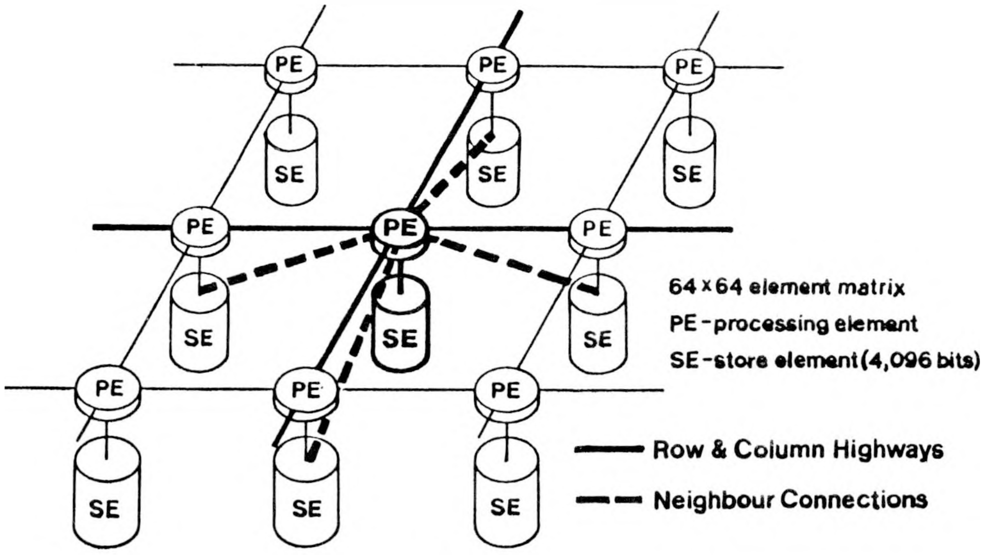


Fig. 1.

DAP AS PART OF AN ICL 2900 'HOST' COMPUTER

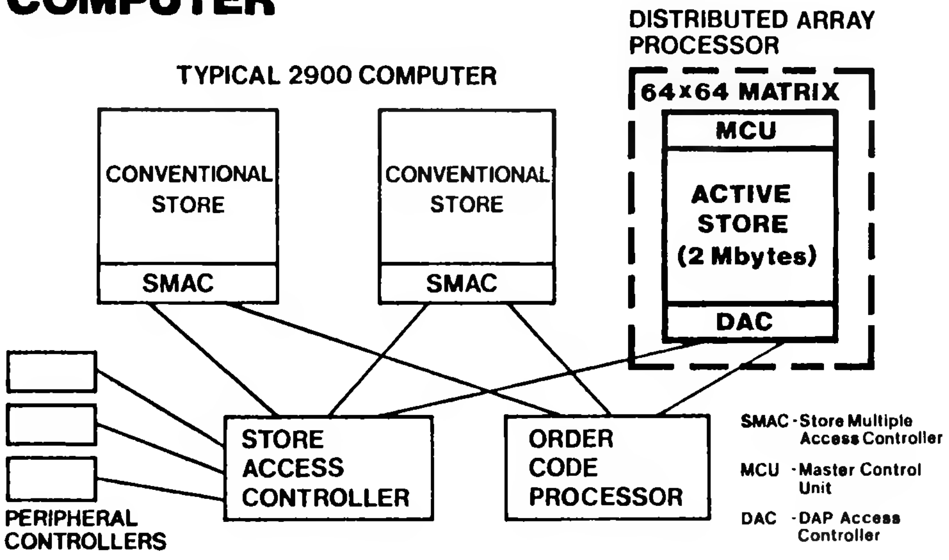


Fig. 2.

Figure (2) shows how this processing array forms part of an ICL 2900 host computer, by replacing a standard store module. The storage elements used by the processors are also seen by the host computer as standard store, and hence the data held in the storage elements may be processed by either the 2900 serial computer or by the DAP array processor.

SOFTWARE OVERVIEW

One significant advantage of the DAP/2900 approach over the traditional front end/back end configuration, is that all the software and facilities available within the 2900 can be fully utilised to operate on the data in the DAP store. This is reflected in the overall software structure of the DAP shown in figure (3). The host machine components, which typically perform serial functions such as I/O and database handling, may be written in any serial language on the 2900. The DAP routines may be written either in DAP-Fortran (a parallel dialect of Fortran) or in assembler. Data used by both sets of software exists as Fortran COMMON blocks.

USER DIAGNOSTICS ON THE ICL 2900

The virtual storage system on the 2900, VME/B, provides a full mixed language debugging and post mortem facility known as the Object Program Error Handler (OPEH). All 2900 compilers produce a diagnostic module for each compilation unit (subroutine). At run time these modules reside in the virtual storage on disc, until an error occurs. All errors, unless masked by the user, are handled by the OPEH software, which takes appropriate action depending on the options set by the user. A typical report is shown in figure (4), where the user is informed of the nature and location of the error in terms of the compilation source listing, the calling sequence taken to the failing module, and values of some or all data items used in the module. Following the post mortem the user can decide whether to terminate the run, continue at the next instruction or enter a user written error trap procedure prior to termination or continuation.

As well as providing diagnostic reports in user language, this system has the advantage that for an error-free run there is no run-time overhead caused by the diagnostic system.

INTERFACE TO DAP

The OPEH system handles all programs regardless of which high level language or combination of languages is used. Thus it is relatively simple to interface DAP to the OPEH system by merely producing the relevant diagnostic modules from the compiler, see figure (5). It is also possible to enhance the diagnostic system in line with the specialised requirements of the array processor applications, by using some of the specific properties of the DAP. These enhancements are outlined below.

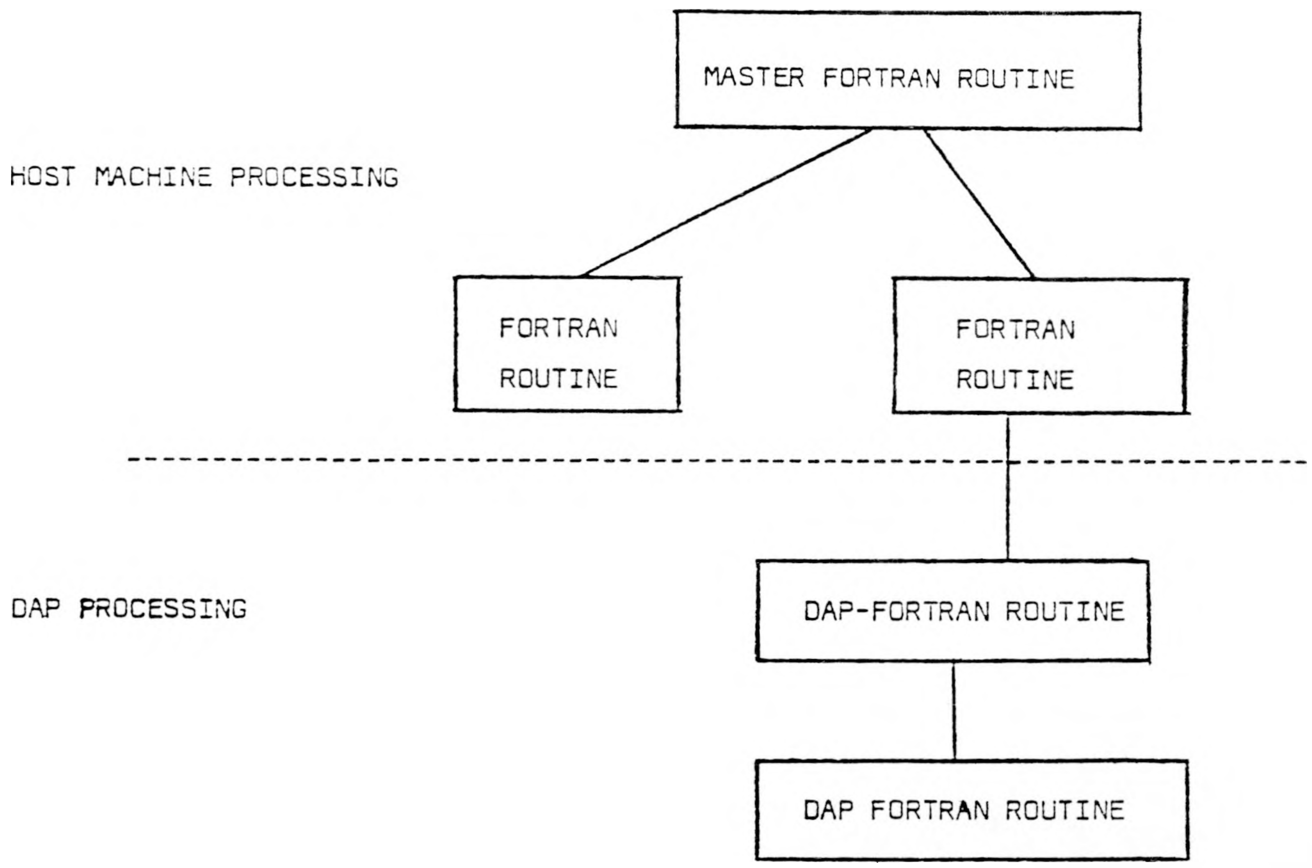
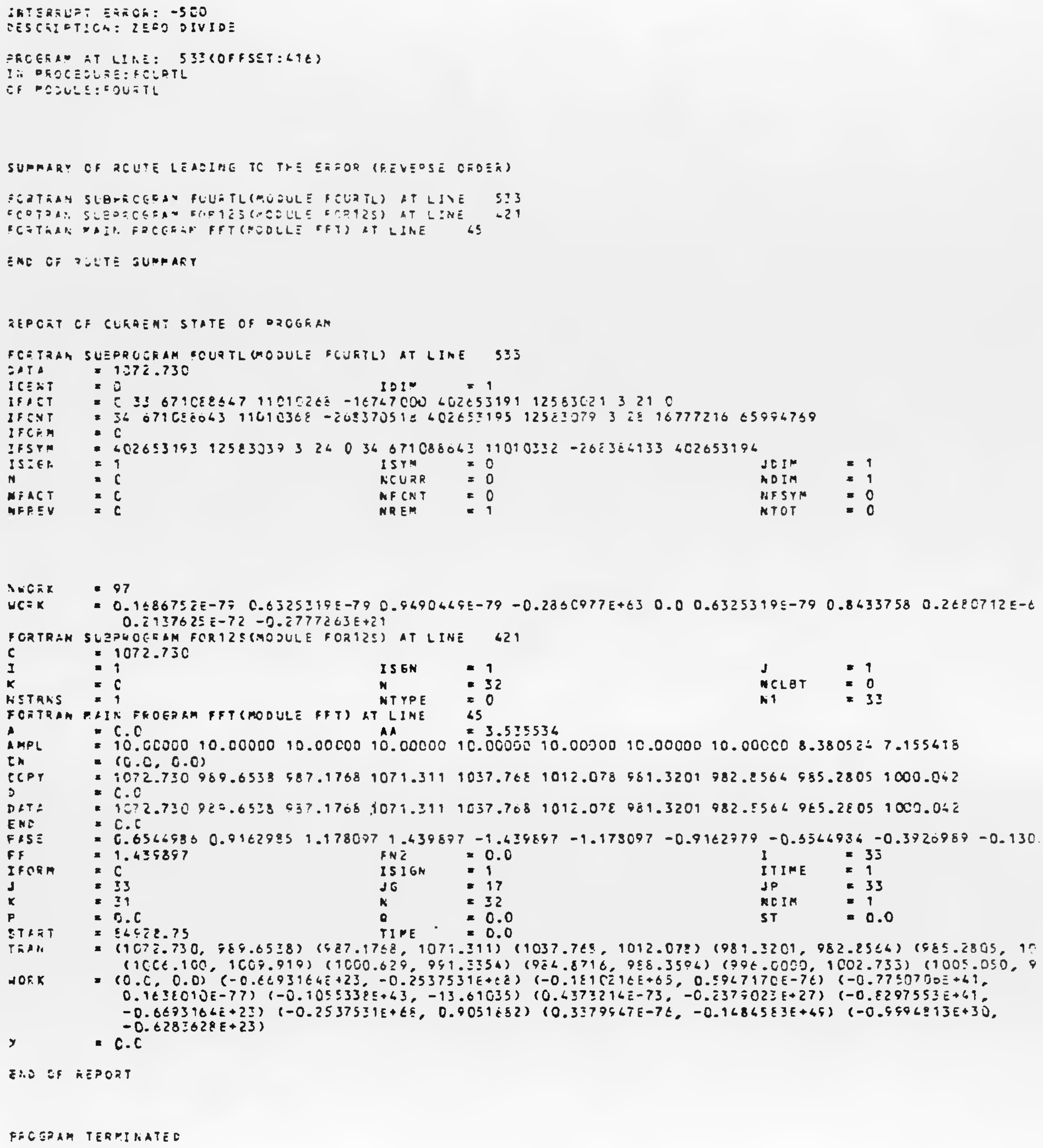


Fig. 3. Sample run-time system.



```
INTERRUPT ERROR: -500
DESCRIPTION: ZERO DIVIDE

PROGRAM AT LINE:  533(OFFSET:416)
IN PROCEDURE:FOURTL
OF MODULE:FOURTL



SUMMARY OF ROUTE LEADING TO THE ERROR (REVERSE ORDER)

FORTRAN SUBPROGRAM FOURTL(MODULE FOURTL) AT LINE   533
FORTRAN SUBPROGRAM FOR12S(MODULE FOR12S) AT LINE   421
FORTRAN MAIN PROGRAM FFT(MODULE FFT) AT LINE    45

END OF ROUTE SUMMARY



REPORT OF CURRENT STATE OF PROGRAM

FORTRAN SUBPROGRAM FOURTL(MODULE FOURTL) AT LINE   533
DATA      = 1072.730
ICEXT     = 0                               IDIM      = 1
IFACT     = 0 33 671088647 11010268 -16747000 402653191 12583021 3 21 0
IFCNT     = 34 671088643 11010368 -268370518 402653195 12583079 3 28 16777216 65994769
IFORM     = 0
IFSYM     = 402653193 12583039 3 24 0 34 671088643 11010332 -268384133 402653194
ISIGN     = 1                               ISYM      = 0                               JDIM      = 1
N         = 0                               NCURR     = 0                               NDIM      = 1
NFACT     = 0                               NFCNT     = 0                               NFSYM     = 0
NPREV     = 0                               NREM      = 1                               NTOT      = 0



NWORK     = 97
WORK      = 0.1686752E-79 0.6325319E-79 0.9490449E-79 -0.2860977E+63 0.0 0.6325319E-79 0.8433758 0.2680712E-6
              0.2137625E-72 -0.2777263E+21
FORTRAN SUBPROGRAM FOR12S(MODULE FOR12S) AT LINE   421
C         = 1072.730
I         = 1                               ISGN      = 1                               J         = 1
K         = 0                               N         = 32                              NCLBT     = 0
NSTRNS    = 1                               NTYPE     = 0                               N1        = 33
FORTRAN MAIN PROGRAM FFT(MODULE FFT) AT LINE    45
A         = 0.0                             AA        = 3.535534
AMPL      = 10.00000 10.00000 10.00000 10.00000 10.00000 10.00000 10.00000 10.00000 8.380524 7.155418
CN        = (0.0, 0.0)
COPY      = 1072.730 969.6538 987.1768 1071.311 1037.768 1012.078 981.3201 982.8564 985.2805 1000.042
D         = 0.0
DATA      = 1072.730 989.6538 987.1768 1071.311 1037.768 1012.078 981.3201 982.8564 985.2805 1000.042
END       = 0.0
FASE      = 0.6544986 0.9162985 1.178097 1.439897 -1.439897 -1.178097 -0.9162979 -0.6544934 -0.3926989 -0.130
FF        = 1.439897                        FN2       = 0.0                             I         = 33
IFORM     = 0                               ISIGN     = 1                               ITIME     = 1
J         = 33                              JG        = 17                              JP        = 33
K         = 31                              N         = 32                              NDIM      = 1
P         = 0.0                             Q         = 0.0                             ST        = 0.0
START     = 54928.75                        TIME      = 0.0
TRAN      = (1072.730, 989.6538) (987.1768, 1071.311) (1037.768, 1012.078) (981.3201, 982.8564) (985.2805, 10
              (1006.100, 1009.919) (1000.629, 991.3354) (984.8716, 988.3594) (996.0000, 1002.733) (1005.050, 9
WORK      = (0.0, 0.0) (-0.6693164E+23, -0.2537531E+68) (-0.1810216E+65, 0.5947170E-76) (-0.7750706E+41,
              0.1638010E-77) (-0.1055338E+43, -13.61035) (0.4373214E-73, -0.2379023E+27) (-0.8297553E+41,
              -0.6693164E+23) (-0.2537531E+68, 0.9051682) (0.3379947E-76, -0.1484583E+49) (-0.9994813E+30,
              -0.6283628E+23)
Y         = 0.0

END OF REPORT


PROGRAM TERMINATED
```

Fig. 4.

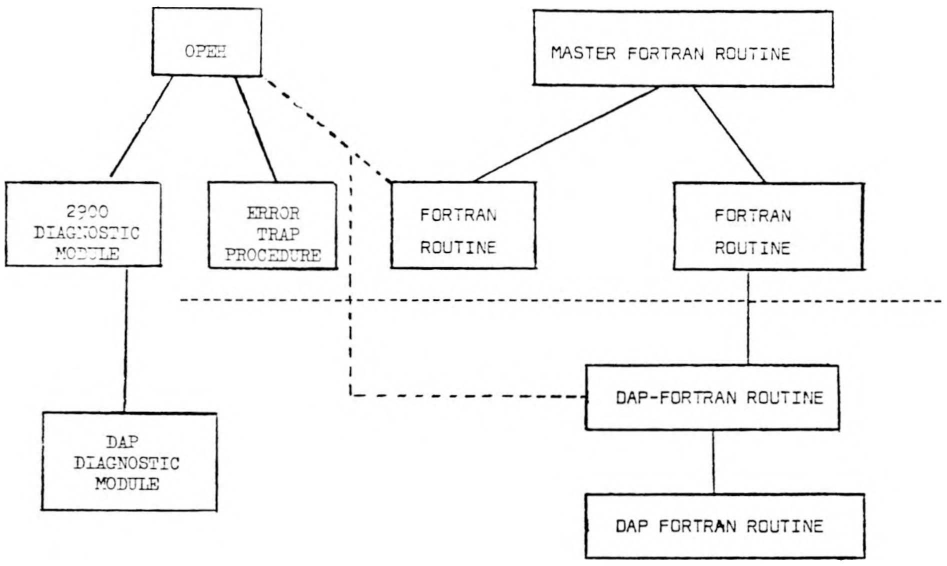


Fig. 5. DAP/2900 diagnostic system.

The basic processing element of the DAP array has a word length of one bit, and hence provides efficient manipulation and storage of logical (boolean) variables. A matrix of 64 x 64 logical variables is stored as one bit per processor, and can typically be used to control operations on a matrix of 64 x 64 arithmetic variables. Various facilities exist to provide local or global testing of the logical arrays, as well as assignment and manipulation. To control the action of the diagnostic system the user has three separate logical matrices, the Program Error Mask (PERM), the Program Error Matrix (PEM) and the Activity Mask (AM). There are also two 64 element logical vectors DUMP and GO controlling the action of the diagnostic system for up to 64 different error types, e.g. underflow, overflow, etc.

At the end of each low level arithmetic function, such as floating point multiply, the result in each processor is checked for errors. If an error such as overflow has occurred the DUMP and GO vectors are checked for that error class to determine what action should be taken. If the user has requested the system to continue regardless of error, the program error matrix is set for those processors where errors occurred. The user may then take his own action, such as re-scaling his data, after checking for errors with a statement such as

```
IF (ANY (PEM)   ) CALL   SCALE (....)
```

where the function ANY performs a logical OR of the complete PEM matrix. If the user has requested a post mortem dependent on the position of the errors, before any dump is taken the program error matrix is checked against the Program Error Mask. Only if errors have occurred in processors where the mask is true will a post mortem be taken. A further check is taken against the activity mask, if one is being used in the assignment, so that only errors occurring in positions where the results are actually going to be used will be notified. This is typically used during parts of a matrix inversion, where at various stages in the computation the pivot row and column are not used (see ref 1).

These facilities allow the programmer to make use of the full parallelism of DAP without worrying about elements of the DAP which, at any time, are not working on useful data. A further typical use would be when manipulating the 62 x 62 'inside' section of a 64 x 64 matrix.

As well as trapping errors, the diagnostic system can be invoked directly by the user program. DAP-Fortran provides an ERROR statement, which can be used to signal an error number to either the OPEH system or to the user error trap procedure.

A typical use might be

```
IF (ANY (X.LT.0.0)  ) ERROR 77

Y = SQRT (X)
```

where X and Y are matrices.

Although a wide range of facilities is provided by the system, there is only a trivial run time overhead due to the bit level nature of the DAP. The

IF (ANY (X.LT.0.0)) statement will take about one microsecond for the complete 4096 element matrix, compared with around 150 microseconds for a 4096 element floating point matrix addition.

TRACE

As well as the diagnostic system, which is designed for rapid execution until errors are discovered which require post mortem printing, the DAP also allows both high level (DAP-Fortran) and low level (assembler) trace facilities.

At the DAP-Fortran level, this is implemented by a TRACE statement, which has the status of an executable statement in Fortran. The statement has the form

```
TRACE n, (variable list)
```

and may be combined with conditional statements, e.g.

```
IF (ANY (Y.LT.0.0) ) TRACE 3, (Y)
```

At run time the value of n is compared with the run time trace level, and if n is less than the current trace level the DAP interrupts the host system, which then carries out the necessary printing.

REFERENCE

1. Flanders et al. "Efficient High Speed Computing with the Distributed Array Processor," High Speed Computer and Algorithm Organisation, Ed. Kuch, Lawrie and Sameh, Academic Press, (1977).

A HIGH PERFORMANCE GRAPHICS SYSTEM FOR THE CRAY-1

by

Robert H. Ewald
Lynn D. Maas
University of California
Los Alamos Scientific Laboratory
Los Alamos, New Mexico 87545

ABSTRACT

This paper describes the design and implementation of a state-of-the-art interactive vector graphics system connected to the CRAY-1 supercomputer. The primary design goal for this graphics system is that it support large hydrodynamic computer programs used in weapons design calculations. The interactive use of these programs requires displays consisting of up to 20 000 vectors, extensive interaction tools, and high-bandwidth communication rates. The major system components selected for this project were an Evans and Sutherland Picture System 2 and a Digital Equipment Corporation (DEC) PDP-11/70 and PDP 11/34 running the UNIX operating system.

This paper presents the system design goals and performance criteria. The hardware/software systems chosen for this project are reviewed, and the integration of this system into the Los Alamos Scientific Laboratory's (LASL) Integrated Computer Network (ICN) is described. This implementation involved most areas of applied computing, including computer graphics, communications, distributed processing, and computer security. The level of effort required for this implementation is described, and the results and benefits are presented. Future plans for this system are also briefly described.

*Work for this project was conducted under the United States Energy Research and Development Administration (ERDA) Contract W-7405-ENG. 36.

CRAY-1 ERROR RECOVERY

by

Alex Marusak
University of California
Los Alamos Scientific Laboratory
Los Alamos, New Mexico 87545

ABSTRACT

A package of FORTRAN-callable error recovery routines has been written for the CRAY-1 operating under the DEMOS System. Programs may regain control after the occurrence of the normally fatal errors: Error Exit, Program Range, Operand Range, Floating Point Overflow and Time Limit. On recovery a program may branch either to a user-specified error-handling procedure or to a user-specified statement label within a previously entered procedure. The recovery package will optionally print the type of error which occurred and a trace-back from the point of error through the calling path back to the main program. Repeated recoveries are permitted.

THE CRAY-1 COMPUTER AND THE DEMOS OPERATING SYSTEM

by

Forest Baskett
University of California
Los Alamos Scientific Laboratory
Los Alamos, New Mexico 87545

ABSTRACT

The unusual features of the CRAY-1 computer are described with emphasis on those relevant to the operating system being implemented at Los Alamos. The machine is a high-speed computer with a 12.5-nanosecond cycle time, vector registers and vector operations. The memory addressing structure is very simple but the memory bandwidth is 5 gigabits per second. This results in an operating system design that is task oriented and depends on copying information from one address space to another.

The mechanism that supports this communication in DEMOS is based on ideas from capability-based operating system design. The implementation is unusual and is especially designed to support a high-performance file system. The file system design borrows ideas from UNIX with important elaborations in allocation and buffering to ensure high performance.

THE ILLIAC IV SYSTEM IN 1978

by

David Stevenson

Institute for Advanced Computation

NASA-Ames Research Center

Moffett Field, California

ABSTRACT

This paper describes the current system configuration of the ILLIAC IV at NASA-Ames Research Center. The status of the present hardware is briefly described and I/O performance characteristics of representative ILLIAC codes is used to illustrate the current modes of use of the memory system. The paper concludes with a review of plans to increase the processing capabilities and memory capacity of the ILLIAC IV.

CTSS STATUS REPORT

by

Dieter Fuss
Lawrence Livermore Laboratory
Livermore, California 94550

ABSTRACT

The National Magnetic Fusion Energy Computer Center (MFECC) provides large-scale computational support to the Magnetic Fusion research community throughout the country. The MFECC serves over forty groups of researchers in national laboratories, universities, and industry, thus facilitating the sharing of information, codes, data, manpower, and computer power.

The primary worker computer of the MFECC is a CRAY-1. In order to provide the most human-productive service to the over 1000 users of the MFECC, a timesharing operating system, called CTSS, was developed for the CRAY-1. The status of this operating system, its capabilities, and some of the motivations for developing it will be discussed.

EARLY EXPERIENCE WITH THE CRAY-1 AT NCAR

by

Richard K. Sato
National Center for Atmospheric Research
Boulder, Colorado

ABSTRACT

The CRAY-1 at the National Center for Atmospheric Research was delivered in July 1977. Although general users have access to the CRAY-1, large numerical models consume a major portion of the time. The CDC 7600 currently serves as the front-end to the CRAY-1 system and the Ampex TBM serves as the mass storage medium for "permanent" datasets.

A brief summary of the first year of the CRAY-1 at NCAR is presented. The work involved in the conversion of a large code (the NCAR general circulation model) from the CDC 7600 to the CRAY-1 is described and timing comparisons given. The current NCAR configuration is described and the system--hardware and software--is discussed from a users standpoint.

---

*The National Center for Atmospheric Research is sponsored by the National Science Foundation.

VECTORIZATION FROM A LARGE CODE POINT OF VIEW

by

Margaret W. Asprey

Energy Division
Los Alamos Scientific Laboratory
University of California
Los Alamos, NM 87545 USA

ABSTRACT

While very large codes make the most efficient use of a vector machine like the CRAY, they also are usually the most difficult to vectorize. Since they were usually written to optimize scalar usage as well as to solve a number of different problems, an understanding of the special techniques and storage structures used is important. To prevent wasted effort, a decision should be made at the start as to the format in which the code is to be preserved for future maintenance and further modification.

The MCA VECTORIZER is a useful tool for the purpose but must be used in conjunction with restructuring the existing FORTRAN. For efficiency, only heavily used sections of the code should be vectorized, so that these have to be identified initially. The basic restructuring techniques are: 1) separating vectorizable code from nonvectorizable, and 2) rearranging DO-loops and transfers.

From my experience, it seems more efficient to rewrite the pertinent parts of the code from the basic algorithms to avoid interference from scalar-type thinking. In any event, an overall knowledge of the code is important to do an efficient job.

I. INTRODUCTION

Having spent a year struggling with one of the new third-generation computers, the CRAY-1, I am quite convinced that a new generation of programs and/or programmers is needed to cope! Apparently Seymour Cray feels similarly. He was quoted in Business Week as saying, "Nobody, and I mean nobody knows how to program large parallel machines."

While it is true that simply moving scalar code onto the CRAY brought a speed increase of better than a factor of two, the work involved in the transfer, including the difficulties of a new system and different word lengths, are likely to more than offset the gain unless additional speed can be obtained from vectorization. Here the potentials are large, up to factors of twenty, but obtaining them has not turned out to be so easy. Since, as I'm sure everyone is aware, the really expensive part of computing is the software, in this report I will discuss my experience with two attempts to solve this problem. First, the vector primitives developed at LASL[1] and second the VECTORIZER developed at MCA (Massachusetts Computer Associates).[2,3,4] At the outset, it must be admitted that more problems remain than have been solved. While there is talk of a vectorizing compiler, so far, talk is all we have.

II. VECTOR PRIMITIVES

The vector primitives are a set of callable macros that perform a variety of vector operations. The general format available follows, with the first being new, but is now recommended:[1]

CALL name (3Hjki, P_j, P_k, P_i)

or

CALL name j k i (P_j, P_k, P_i)

where:

name	specifies the macro, and always consists of four alphabetic characters.
j,k,i	specify the vector registers or small core arrays to be used in the operation.

NOTE: If j,k, or i are digits from 0 through 7, the corresponding vector register is to be used. If any are replaced by the letter X, the corresponding argument in the calling sequence is to be used.

P_j, P_k, P_i	may be array names, elements of arrays, scalars or zero.

NOTE: The indices correspond to the register specification above where required. If an argument is zero (or omitted), for example, P_i, the corresponding register, i, must have been specified. If an array name (or element of an array) is given, the vector operation will begin with the first element (or the given element). The length of the operation will be determined by the contents of the vector length register that is set by the macro SETL as shown in Table I. In this case the compiler will assign the registers.

TABLE I

VECTOR TO VECTOR REAL ADD AND VECTOR TO SCALAR REAL PRODUCT

<u>FORTRAN:</u>

```
      DO 10 I = 1,LEN
 10   A(I) = B(I) + C(I)*Q
```

<u>VECTOR:</u>

CALL SETLX(LEN)	Set desired vector length
CALL VRAVXX1(A,B,O)	Add A(I) to B(I) and store in register V1
CALL SRPVX1X(Q,O,C)	Take real product of Q and contents of V1 and store in C(I)

If the programmer wishes to control register usage, which can be an advantage since the maximum speeds are obtainable through what is known as "chaining," i.e., having registers ready for operation before needed so that operations can continue without waiting, this could be done as shown in Table II.

Obviously, this last requires more from the programmer. Codes already large become still larger, much more difficult to follow and, thus, more difficult to debug, maintain, and modify. A partial solution to this problem is the VECTORIZER developed by MCA.[2,3,4]

TABLE II

CONTROL OF REGISTER OPERATED BY PROGRAMMER

<u>VECTOR</u>:

CALL SETLX	Set vector length
CALL ILDV XXO(A,1)	Load A(I) in register VO, incrementing by 1*
CALL ILDV XX1(B,1)	Load B(I) in register V1, incrementing by 1*
CALL VRAV 012	Add contents of vectors VO and V1 and place in V2
CALL SRPV X23(Q)	Take the product of scalar Q with the elements of vector V2 and place in V3
CALL ISTV 3XX(0,1,C)	Store contents of V3 in C, incrementing by 1*

*Default increments are always 1

III. THE VECTORIZER

The VECTORIZER is a FORTRAN program developed by MCA (Massachusetts Computer Associates) for the purpose of performing automatically the conversion of scalar FORTRAN to vector FORTRAN in the format of the vector primitives such as described above for the CRAY-1 and has been designed so that code for different systems can be obtained by modification of the codegenerator, CVPGEN, only. Since all my experience is with CRAY-1,[5,6] only this type of output will be considered here. A simplified diagram of the structure of the VECTORIZER is given in Fig. 1 with sample input in Fig. 2. Since for the average user, the internal structure of the VECTORIZER is of no concern, reference is made to MCA, the manual[2], and their memoranda.[3,4] The VECTORIZER will be treated from here on as a "black box" with FORTRAN input going in on one side and vector FORTRAN suitable for the CRAY-1 coming out the other side. Note carefully, it is truly FORTRAN, even though at a casual glance it almost looks like assembly language.

IV. RESTRUCTURING

Before doing any vectorization, the code should be analyzed for the utility of vectorization. It should be recognized that a certain amount, possibly even a fairly large amount, of time and effort will be required for recoding and debugging. Also, the vectorized code is more difficult to read, maintain, and modify, so that a decision must be made as to whether one, two, or three versions will be maintained in the future. Will the original scalar FORTRAN be retained or discarded? Can the people who will be working on the code. such as making changes and adding features, read the vector FORTRAN well enough to make changes there or should the restructured code be maintained as well and changes made in it? One of the big advantages of the VECTORIZER is that changes can be made in the restructured code. which is then passed through the VECTORIZER, rather than having to make the changes in the vectorized code itself. However, it should be recognized that the restructured code is usually quite different from the original scalar code and it is quite possible that all three versions will be needed and when changes are made, the restructuring will have to be done again as well as passing the code through the VECTORIZER. In my opinion, this decision should be made almost at the beginning, because it affects how much and where changes are made in restructuring. I have seen restructuring thrown away because by the time it was debugged, the scalar code had been rewritten so that the restructuring had to be done again.

Once the decision to vectorize is made, the next decision is "how much" and "which parts" of the code will be modified. If routines for determining the amount of time spent in various parts of a program such as STAT[7] are available, they should be used to identify the most heavily used parts of the code for initial vectorization. The type of code should be considered carefully because certain types are difficult, if not impossible, to vectorize. It cannot be emphasized too much that time and effort are necessary to obtain results of any magnitude. The more complex the code, the larger these costs will be. On the surface this may not be obvious, but for so many years the optimization emphasis for the large "cell" type computer code has included skipping unnecessary calculations for inactive or empty cells, that it means a major change in thinking that is not easy. This sort of programming can only be vectorized by major reprogramming with attendant debugging.

If the code exhibits many transfers forward and/or backward, into or out of loops (including function and subroutine calls), or if there are many optional paths through the code, restructuring will be extremely difficult. Probably, in fact, a code of this type should not be vectorized at all unless it is <u>very</u> heavily used. In particular, the algorithm should receive a long hard look for possibilities of simplification. In my opinion, vectorizing in general would be more efficient if one started from the original algorithm rather than from the scalar code. The scalar methods are different enough that they tend to get in the way of "thinking" vector.

In considering the material that is to be vectorized, there are a number of basic principles to keep in mind (some of these are due to John Levesque of Research Development Associates (RDA):[8]

1. Only vectorize the heavily used parts of the code.

2. Only DO loops can be vectorized, but small loops are inefficient because they don't allow chaining.

3. Avoid transfers as much as possible.

4. Separate vectorizable and non-vectorizable code.

5. If possible, rethink the algorithm.

6. Fetches and stores interrupt chaining and should be minimized.

7. Code that cannot be vectorized include

   DO loops with transfers in or out, including function and subroutine calls

   DO loops with internal transfers

   DO loops with indirect or offset indexing

   Code without DO loops.

Once a routine is selected for restructuring, the first step is to consider the DO loop structure. If there are no major DO loops in a routine, it is usually either called from a loop in another routine or it calls a routine containing a loop. As much of the calculation as possible should be done within a loop, inside the same routine. Either the calculations should be pulled into the main routine or the DO loop pulled into the called routine. Examples of both techniques are given in Figs. 3,4, and 5.

The next step is to get rid of as many transfers as possible. How this is done is a function of the original problem. One example is given in Fig. 6 with the vectorized result as Fig. 7. It should also be noted that very short loops are not efficient since they interrupt the chaining. Where possible for best efficiency, all fetches into vector registers should be done in advance so that as soon as one operation is complete the next can be started without waiting for a fetch from memory. This is not being done in Fig. 6. My understanding is that the VECTORIZER does better on this as of now.

In conjunction with removing the transfers, the nonvectorizable code should be separated from the vectorizable. This can often be done by pre- and/or postsetting special and boundary conditions. In the example shown in Fig. 8, in the original code, immediately after the DO-1000 K-line, there are complicated tests and transfers involving the setting of inactive and/or empty cell variables to zero, followed by a skip to the end of the loop. This prevents any vectorizing in the entire routine. By letting all values be calculated, even though there is "garbage" in the values for the empty cells, then postsetting them as shown in Fig. 9, the bulk of the calculations are vectorized (Fig. 10) and a respectable speedup is achieved. However, this is also an example of how rethinking the algorithm would have helped. Tests similar to these are made in other places in other routines as well. Making the determination in one place, setting a few bits on a flag word of which there is at least one for each cell, would prevent this repetition. However, this involves knowing the overall code and data structure rather than just the routines being vectorized.

Another fairly common difficulty is the use of indirect indexing. which also can effectively prevent an entire loop from being vectorized. Figs. 11, 12, and 13 are examples of this problem with some suggestions of what can be done about it. Figure 13 uses INDEXED, which is a routine written by John Levesque of RDA, in CAL, the assembly language for the CRAY-1. Some further tests have shown that a single call is efficient and will give a gain in speed while if more than one block is to be used, it is less efficient than a single loop. The tests were made by Don Willerton (C-3). Also included in this problem is that of offset indices when a cell above, below, or to one side is being considered in a differencing problem. The VECTORIZER can handle this situation unless it is an iterative procedure such as:

```
X(I) = A(I) * X(I+1) + B(I) * X(I-1).
```

Here the FORTRAN code requires old values for I+1 and new values for I-1. But in vector mode only all of the old values or all of the new values for I, I-1, and I+1 are available at one time. The easiest solution to this problem is the use of temporaries to store the old values and/or to save the new values.

Two examples of ways to handle the postsetting of special conditions are given in Figs. 14 and 15. In Fig. 14 a logical variable is used for making the decision rather than the original transfer and in Fig. 15 the test is pulled into a DO loop rather than using a transfer. One other minor problem is that the VECTORIZER cannot convert functions such as SIN, COS, EXP, nor the FORTRAN exponential to vector operation (see Figs. 16 and 17). Again, these calls will prevent vectorization until vector functions are available.

An example of one way to replace forward transfers so that the loop can be vectorized is given in Fig. 18. There are, of course, many different ways to achieve this, for example, use of logical variables to make the tests. The example would be even more useful if more than two values were to be summed. Another old trick to gain speed in scalar coding and one that no longer helps in vector mode is that of precomputing indices. At one time, since CRAY has no vector integer multiply unit, the precomputing was being done incorrectly. This has been fixed, but is still less efficient than letting the system do it for you.

Finally, we come to the interface difficulties. Both of these result from precompilation insertion of fixed length dimensions. The results are hardly the fault of the VECTORIZER since what is being done is quite reasonable from what the VECTORIZER "sees".

The amount of storage available to run one of these big codes has been a continuing problem. Some techniques that have been used include: preprocessors; use of the PARAMETER statement (the latter only now becoming ANSI standard FORTRAN) so that the compiler inserts the appropriate integer dimensions; and one large array with all variables packed contiguously with pointer words precomputed and passed to subroutines through the calling sequences. Possibly, this will become less of a problem with the new larger memories. However, in general, experience shows that programmers simply expand to fill the available space and call for more. The solution used here was essentially that of the preprocessor. The MOLL code, which is an editor as well as pre- processor, uses a PARAMETER-type statement to obtain and compute values to be inserted into dimension statements in "CLICHE's" or "COMDECK's". The CLICHE's are then inserted into routines calling for them and the source code is then sent to the compiler.

In order to run routines through the VECTORIZER, the dimensions must have been inserted by the editor so that the VECTORIZER "thinks" they are fixed and unchanging. If now it is necessary to pick a vector out of a multiply (2 or 3) dimensioned array, based on the second or third index, a "stride" is computed based on these fixed dimensions and inserted into the vector primitives in the body of the routine. If these modified routines are then inserted in the production version and in the future the parameter dimensions are changed in the CLICHE's, the strides would <u>not</u> be changed and wrong values would be indexed by the vector routines. See Figs. 19 and 20.

It is clear from Fig. 19 that when a parameter value is changed, a wrong correspondence would exist between the contents of AAA0(L) and that of A(K,L). The only solution to this problem that we have come up with is to allow the option to insist that variables always be used for stride values. However, this is apparently a fairly major recoding job and has not. to my knowledge. been done as yet.

Another difficulty is also caused by the space problem and pre-insertion of fixed dimensions. In this case, many different types of problems can be run with the same program using different options. However, again to save space, it is desirable to use only the storage needed for the type being currently run and not waste space on variables that will not be used. Again using the PARAMETER statement, the variables not to be used are set to dimensions of 1 by 1. An example of this is given in Figs. 21, 22, and 23. Now in scalar code, when this variable is to be used or set, some parameter is tested before the usage. If the parameter is true, the variable is dimensioned and will be used. If it is false, the variable is not dimensioned and should not be set. When the VECTORIZER is modifying a loop containing this test, it looks at all dimensioned variables used in the loop, determines the smallest, and concludes that the extent of the DO loop need not be any larger than the smallest such dimension. This is quite reasonable. In Fig. 22, the VECTORIZER generated variable AAA3 while only dimensioned one, will be stored into as if it were dimensioned KMAX. thus overflowing into other storage. The VECTORIZER doesn't know anything about our clever trick of saving storage space. The direct solution is to avoid the problem by removing the variable from the DO loop, which can then be vectorized correctly, and move the test into a loop that cannot be vectorized anyway. In Fig. 23, the DO-950 loop cannot be vectorized anyway and now the DO-1000 loop will be handled correctly. The ultimate and ideal solution to this problem is not obvious to me, but potential users should be warned of its existence.

While the VECTORIZER still has some deficiencies, it is a very useful and valuable tool but should be used with some caution. Above all however, it should be recognized that time and effort will be required to use it. The newly revised user's manual[2] of April 12, 1978, has some good suggestions as to restructuring, and MCA has been very cooperative about correcting deficiencies and adding features. Its big advantage, of course, is that once the restructuring is done correctly, by a simple pass through the VECTORIZER, speed increases of better than two times are obtained without the need of making changes in the highly unreadable vector code or learning to write in the vector primitive mode.

To sum up, vectorizing requires a new look at some old problems. The VECTORIZER, while not a panacea, can be of assistance in rapidly vectorizing code and does the job reasonably efficiently and usually correctly compared with the time needed to learn how to vectorize and then to do the corresponding thing manually. There are pitfalls, some of which have been pointed out here. The main point is that vectorizing requires a basic change in thinking that is not ever easy. My own belief is that one should go back and rethink the algorithm (probably a useful exercise anyway!) because I find that efficient vector code and efficient scalar code are enough different that the scalar code has a tendency to get in the way of thinking along vector lines. Vector thinking is a necessity to effectively use the CRAY.

REFERENCES

1. LASL Guide to the CRAY-1 Computer, Software Documentation, Group C-2, LASL, 7/77.
2. Mathew Myszewski, David Loveman, Vectorizer System User's Manual, 4/12/78.
3. Mathew Myszewski, COMPASS Vectorizer Information Memo, Memorandum from Massachusetts Computer Associates, Inc., Wakefield, Mass., 3/13/77.
4. Mat Myszewski, CVPGEN, The CVP Generator Control Routine, Memorandum from Massachusetts Computer Associates, Inc., 9/7/77.
5. CRAY-1, Computer System, CRAY OS Version 1.0, Manual #2240011, Cray Research, Inc., 7/77.

6. CRAY-1 Computer System, CAL ASSEMBLER Version 1, Manual #2240000, Cray Research, Inc., 7/76.

7. STAT, Program Activity Statistics Package, (For CDC7600, CROS System), Internal Documentation, L. Rudsinski, C-3, and J. Melendez, C-2, 1975.

8. John Levesque, How to Write Vectorizable FORTRAN, Unpublished, Internal Memo, RDA, 1977.

```
                FORTRAN input
                      !
                      !
                      v
                    PARSER
               (reads FORTRAN)
                      !
                      !
                      v
                   ANALYZER
 (recognizes implicit vector operations)
                      !
                      !
                      v
                  GENERATOR
 (converts to explicit vector operations)
                      !
                      !
                      v
                 TRANSCRIBER
               (writes FORTRAN)
                      !
                      !
                      v
               FORTRAN output
```

Fig. 1.
Simplified flow chart of vectorizer from Ref. 8.

```
              DO FOR ALL body

WRL(I) = (ALR(I,L) + ALR(I+1,L))**2

              transformed body

CALL VRAV X X 1 (ALR(1,L),ALR(2,L)
CALL VRPV 1 1 X ( Ø,  Ø,   WLR(1))
```

Fig. 2.
Example of input and output for generator.

```
      SUBROUTINE YY
      DIMENSION A(100),B(100)
      LMAX = 100
      DO 100 L = 1,LMAX
                 .
                 .        (CALCULATION SET A)
      .          .
      CALL XX(A(L),B(L))
                 .
                 .        (CALCULATION SET B)
                 .
  100 CONTINUE
                 .
                 .
                 .
      END

      SUBROUTINE XX(A,B)
      DATA Z/10/
                 .
                 .       (CALCULATION SET C)
                 .
      B = A * Z + X
                 .
                 .        (CALCULATION SET D)
                 .
      END
```

Fig. 3.
Overall subroutine restructuring. Call XX prevents vectorization in YY and nothing can be vectorized in XX.

```
      SUBROUTINE YY


      DATA Z/10/
      DIMENSION A(100), B(100)
      LMAX = 100
      DO 100 L = 1,LMAX
                 .
                 .          (CALCULATION SET A)
                 .
                 .
                 .          (CALCULATION SET C)
                 .
      B(L) = A(L) * Z + X
                 .
                 .          (CALCULATION SET D)
                 .
                 .          (CALCULATION SET B)
                 .
  100 CONTINUE
                 .
                 .
                 .
      END
```

Fig. 4.
Restructure by pulling called routine XX into calling routine YY. Returning variables to YY requires addition of indices.

```
      SUBROUTINE YY
      DIMENSIONS A(100), B(100)
                        .
                        .
      CALL XX (A,B)
                        .
                        .
      END


      SUBROUTINE XX(A,B)
      DATA Z/10/
      LMAX = 100
      DO 10 L=1,LMAX
                        .
                        .         (CALCULATION SET A)
                        .
                        .         (CALCULATION SET C)
                        .
      B(L) = A(L) * Z + X
                        .
                        .
                        .
                        .         (CALCULATION SET D)
                        .
                        .         (CALCULATION SET B)
                        .
   10 CONTINUE
                        .
                        .
      END
```

Fig. 5.
Restructure by pulling DO-loop into called routine XX. Indices must be addes and calling routine simplified.

Original FORTRAN (cannot be vectorized)

```
      DO 135 L=1,LMAX
      DO 135 K=2,KMAX
      IF ((FIT(K,L).AND. IFMASK) .EQ. 0) GO TO 135
      IM1=FIT(K,L).AND. IMMASK
      IM2=FIT(K,L+1).AND. IMMASK
      IF (IM1.NE. 0 .AND. IM2.NE. 0) GO TO 135
      IF (IM1.EQ. 0 .AND. IM2.EQ. 0) ALOJK(K,L)=0.0
      IF (IM1.NE. 0) ALOJK(K,L)=ALOJ1(K,L)
      IF (IM2.NE. 0) ALOJK(K,L)=ALOJ3(K,L+1)
  135 CONTINUE
C
```

Restructured FORTRAN (will vectorize)

```
C
      DO 135 L=1,LMAX
      DO 135 K=2,KMAX
C
C     TEST CASE FOR VECTORIZATION
      ITM3(K)=FIT(K,L).AND.IFMASK
      ITM1(K)=FIT(K,L).AND. IMMASK
      ITM2(K)=FIT(K,L+1).AND. IMMASK
      TMO(K)=0.
      IF(ITM1(K).NE.0)TMO(K)=ALOJ1(K,L)
      IF(ITM2(K).NE.0)TMO(K)=ALOJ3(K,L+1)
      IF(ITM1(K).NE.0.AND.ITM2(K).NE.0)ITM3(K)=0
      IF(ITM3(K).EQ.0)TMO(K)=ALOJK(K,L)
      ALOJK(K,L)=TMO(K)
  135 CONTINUE
C
```

Fig. 6.
Getting rid of transfers

```
      DO 90049 L=1,LMAX
      NITER=NMAX-1
      CALL LOPCXXX(NITER,OUTRES,OUTRNX)
      CALL SETLX(OUTRES)
      OUTRND=OUTRES
      ISTRIP=2
      DO 90050 OUTRNX=1,OUTRNX
C
C     TEST CASE FOR VECTORIZATION
      CALL ILNVX03(FIT(ISTRIP,L))
      CALL SANVX01(IFMASK)
      CALL ISTV1C1(0,0,ITM3(ISTRIP))
      CALL SANVX02(IMMASK)
      CALL ISTV2C1(0,0,ITM1(ISTRIP))
      CALL ILNVX03(FIT(ISTRIP,L+1))
      CALL SANVX30(IMMASK)
      CALL ISTV40X(0,0,ITM2(ISTRIP))
      CALL SANV005
      CALL ISTV50X(0,0,TMO(ISTRIP))
      CALL ILNVX04(ITM1(ISTRIP))
      CALL SVMX6
      CALL ILNVX07(ALOJ1(ISTRIP,L))
      CALL CVMX75
      CALL ISTV1CX(0,0,TMO(ISTRIP))
      CALL ILNVX02(ITM2(ISTRIP))
      CALL SVMX2
      CALL ILNVX06(ALOJ3(ISTRIP,L+1))
      CALL CVMX410
      CALL ISTV60X(0,0,TMO(ISTRIP))
      CALL ILNVX05(ITM1(ISTRIP))
      CALL SVMX5
      CALL ILNVX07(ITM2(ISTRIP))
      CALL VMSTX(SCPOOL(1))
      CALL SVMX7
      CALL ILNVX02(ITM3(ISTRIP))
      CALL SETMX(SCPOOL(1).AND.SCPOOL(2))
      CALL CSMX21(0)
      CALL ISTV10X(0,0,ITM3(ISTRIP))
      CALL SVMX1
      CALL ILNVX34(ALOJK(ISTRIP,L))
      CALL ILNVX0A(TMO(ISTRIP))
      CALL CVMX455
      CALL ISTV53X(0,0,TMO(ISTRIP))
      CALL ISTV55X(0,0,ALOJK(ISTRIP,L))
      ISTRIP=NSTRWD+ISTRIP
      OUTRNC=4
      CALL SETLX(64)
90050 CONTINUE
  135 CONTINUE
90049 CONTINUE
```

Fig. 7.
Vectorized version of loop from Fig. 6.

```
      DO 990 L=1,LMAX
      DO 1000 K=1,KMAX
      IM=FIT(K,L).AND.IMMASK
      IF(IM.EQ.0)GO TO 15
      MM=SHIFT(IM,-IMS)
      IF(MM.EQ.MVOID)GO TO 15
      ISK=FIT2(K,L).AND.ISKMASK
      IF(ISK.NE.0)GO TO 25
      K1=K-1
      K2=K+1
      L1=L-1
      L2=L+1
      DO 10 I=K1,K2
      DO 10 J=L1,L2
      ISK=FIT2(I,J).AND.ISKMASK
      IF(ISK.NE.0)GO TO 25
   10 CONTINUE
   15 CONTINUE
      ALR(K,L)=0.0
      ALZ(K,L)=0.0
      AKR(K,L)=0.0
      AKZ(K,L)=0.0
      AJT(K,L)=0.0
      RHAT(K,L)=0.0
      UR13(K,L)=0.0
      URZ4(K,L)=0.0
      GO TO 1000
   25 CONTINUE
      TS0=0.5*(RT(K,L)-RT(K-1,L-1))
      TS1=0.5*(RT(K-1,L)-RT(K,L-1))
      ALR(K,L)=TS0+TS1
      AKR(K,L)=TS0-TS1
C
C
C     MANY MORE CALCULATIONS HERE WITH ALL OF THE
C     ABOVE EIGHT VARIABLES BEING SET, ALR,ALZ,ETC.
C
C
C
 1000 CONTINUE
  990 CONTINUE
```

Fig. 8.
Original FORTRAN for example of restructuring by postsetting of variables

```
      DO 990 L=1,LMAX
      DO 1000 K=1,KMAX
      TS0=0.5*(RT(K,L)-RT(K-1,L-1))
      TS1=0.5*(RT(K-1,L)-RT(K,L-1))
      ALR(K,L)=TS0+TS1
      AKR(K,L)=TS0-TS1
C
C
C     ALL OF THE PREVIOUS CALCULATIONS HERE WITH ALL OF THE
C     VARIABLES BEING SET AS BEFORE, ALR,ALZ,ETC.
C
C
C
 1000 CONTINUE
C
      LA=MAX0(L-1,1)
      LB=MIN0(L+1,LMAXP1)
      DO 800 K=1,KMAXP1
      IM2=FIT2(K,LA).AND.ISKMASK
      IM3=FIT2(K,L ).AND.ISKMASK
      IM4=FIT2(K,LB).AND.ISKMASK
      ITM4(K)=IM2+IM3+IM4
  800 CONTINUE
      K1=2
      K2=KMAX
      DO 80 K=K1,K2
      ITM2(K)=FIT(K,L).AND.IMMASK
      MM=SHIFT(ITM2(K),-IMR)
      IF(MM.NE.0)MM=MMOUT(MM)
      ITM3(K)=MM
   80 CONTINUE
      DO 90 K=K1,K2
      IM1=ITM4(K-1)+ITM4(K)+ITM4(K+1)
      IF(ITM3(K).EQ.MVOID)IM1=0
      IF(ITM2(K).EQ.0)IM1=0
      LS0=.TRUE.
      IF(IM1.NE.0)LS0=.FALSE.
      ITM1(K)=LS0
   90 CONTINUE
      DO 100 K=1,KMAXP1
      LS0=ITM1(K)
      IF(LS0)ALR(K,L)=0.
      IF(LS0)ALZ(K,L)=0.
      IF(LS0)AKR(K,L)=0.
      IF(LS0)AKZ(K,L)=0.
      IF(LS0)RHAT(K,L)=0.
      IF(LS0)AJT(K,L)=0.
      IF(LS0)JR13(K,L)=0.
      IF(LS0)JR24(K,L)=0.
  100 CONTINUE
  990 CONTINUE
```

Fig. 9.
Restructured FORTRAN with all variables set and 'garbage' overwritten at end of loop where needed.

```
      INTEGER AAA0,OUTRXS,OUTRMX,LAST,OUTRWD,ISTRIP,OUTRNX,NITER,
     1 INNRMX
      DIMENSION AAA0(81)
      DO 990 L=1,LMAX
      CALL LOPEXXX(KMAX,OUTRXS,OUTRMX)
      CALL SETLX(OUTRXS)
      LAST=OUTRXS
      OUTRWD=OUTRXS
      ISTRIP=1
      DO 90000 OUTRNX=1,OUTRMX
      CALL ILDVX00(RT(ISTRIP,L))
      CALL ILDVX01(RT(ISTRIP-1,L-1))

          .
          .
          .

      CALL VRSV376
      CALL ISTV60X(0,0,AKR(ISTRIP,L))  *
      ISTRIP=OUTRWD+ISTRIP
      OUTRWD=64
      CALL SETLX(64)
      LAST=64
90000 CONTINUE
 1000 CONTINUE

      LA=MAX0(L-1,1)
      LB=MIN0(L+1,LMAXP1)
      CALL LOPEXXX(KMAXP1,OUTRXS,OUTRMX)
      CALL SETLX(OUTRXS)
      LAST=OUTRXS
      OUTRWD=OUTRXS
      ISTRIP=1
      DO 90001 OUTRNX=1,OUTRMX
      CALL ILDVX00(FIT2(ISTRIP,LA))
      CALL SAVVX01(ISKMASK)

          .
          .
          .
      CALL ISTV70X(0,0,ITM4(ISTRIP))
      ISTRIP=OUTRWD+ISTRIP
      OUTRWD=64
      CALL SETLX(64)
      LAST=64
90001 CONTINUE
  800 CONTINUE
```

* Note: The bulk of the calculations were done at this point.

Fig. 10.
Vectorized version of FORTRAN in Fig. 9.

```
      K1=2
      K2=KMAX
      NITER=K2*1-K1
      CALL LOPEXXX(NITER,OUTRXS,OUTRMX)
      CALL SETLX(OUTRXS)
      OUTRWD=OUTRXS
      ISTRIP=K1
      DO 90002 OUTRNX=1,OUTRMX
      INNRMX=OUTRWD*ISTRIP-1
      CALL ILDVX00(FIT(ISTRIP,L))
        .
        .
        .
      CALL ISTV40X(0,0,AAA0(ISTRIP))
      DO 90005 K=ISTRIP,INNRMX
      IF(AAA0(K).NE.0) AAA0(K)=MMOUT(AAA0(K)) ← *
      CONTINUE
90005 CONTINUE
      CALL ILDVX00(AAA0(ISTRIP))
      CALL ISTV00X(0,0,ITM3(ISTRIP)
      ISTRIP=OUTRWD*ISTRIP
      OUTRWD=32
      CALL SETLX(64)
90002 CONTINUE
   80 CONTINUE
       NITER=K2*1-K1
       CALL LOPEXXX(NITER,OUTRXS,OUTRMX)
       CALL SETLX(OUTRXS)
       LAST=OUTRXS
       OUTRWD=OUTRXS
       ISTRIP=K1
       DO 90002 OUTRNX=1,OUTRMX
       CALL ILDVX00(ITM4(ISTRIP-1))
       CALL ILDVX01(ITM4(ISTRIP))
         .
         .
         .
   90  CONTINUE
       CALL LOPEXXX(KMAXP1,OUTRXS,OUTRMX)
       CALL SETLX(OUTRXS)
       LAST=OUTRXS
       OUTRWD=OUTRXS
       ISTRIP=1
       DO 90007 OUTRNX=1,OUTRMX
       CALL ILDVX00(ITM3(ISTRIP))
       CALL ILDVX01(ALR(ISTRIP,L))
         .
         .
         .
       CALL ISTV60X(0,0,UR24(ISTRIP,L))
       ISTRIP=OUTRWD*ISTRIP
       OUTRWD=54
       CALL SETLX(64)
       LAST=64
90007  CONTINUE
  100  CONTINUE
  990  CONTINUE
```

* Note: While almost all of the postsetting is done using vector operations, this one indirect indexing could not be.

Fig. 10 (cont).

ORIGINAL FORTRAN

```
C
      DO 60 L=2,LMAX
      DO 60 K=2,KMAX
      IMN=MNKL(K,L)
      AJTH(K,L)=0.0
      IF(MNKL(K,L).NE.0)AJTH(K,L)=1.372*FLT(K,L)
     1      /(FJTH(K,L)*CKK(IMN))
   60 CONTINUE
C
```

RESTRUCTURED FORTRAN

```
C
      DO 60 L=2,LMAX
      DO 58 K=2,KMAX
      IMN=MNKL(K,L)
      TM10(K)=CKK(IMN)
      IF(IMN.LE.0)TM10(K)=1.0
   58 CONTINUE
C
      DO 60 K=2,KMAX
      TM11=FJTH(K,L)*TM10(K)
      IF(TM11.EQ.0)TM11=1.0
      AJTH(K,L)=1.372*FLT(K,L)/TM11
      IF(MNKL(K,L).EQ.0)AJTH(K,L)=0.0
   60 CONTINUE
C
```

Fig. 11.
Offset indexing and suggested 'fix.' DO-60 K-Loop will vectorize even though DO-58 K-Loop will not.

```
C
      DO 90024 L=2,LMAX
      DO 58 K=2,KMAX
      IMN=MNKL(K,L)
      TM10(K)=CKK(IMN)
      IF(IMN.LE.0) TM10(K)=1.0
      CONTINUE
   58 CONTINUE
C
      NITER=KMAX-1
      CALL LOPC1AA(NITER,OUTRAS,OUTRMX)
      CALL SETLX(OUTRAS)
      OUTRMS=OUTRAS
      ISTRIP=2
      DO 90025 OUTRNX=1,OUTRMX
      CALL TLOVX00(FJTH(ISTRIP,L))
      CALL TLOVX01(TM10(ISTRIP))
      CALL VMPVE12
      CALL TSTV20X(0.0,TM11(ISTRIP))
      CALL SVM26
      CALL CSMUX23(1.0)
      CALL TSTV20X(0.0,TM11(ISTRIP))
      CALL TLOVX04(FLT(ISTRIP,L))
      CALL SMPVX45(1.372)
      CALL VMAV35
      CALL VMPVS07
      CALL VMAVE31
      CALL VMPV712
      CALL TSTV20X(0.0,AJTH(ISTRIP,L))
      CALL TLOVX05(MNKL(ISTRIP,L))
      CALL SVM26


      CALL CSMUX27(0.0)
      CALL TSTV70X(0.0,AJTH(ISTRIP,L))
      ISTRIP=OUTRMS+ISTRIP
      OUTRMS=64
      CALL SETLX(64)
90025 CONTINUE
   60 CONTINUE
90024 CONTINUE
C
```

Fig. 12.
Vectorized version of FORTRAN in Fig. 11 .

Original FORTRAN

```
   DO 10 K=1, KMAXR
   IM = FIT(K,L).AND.IMMASK
   RHO1(k) = RHO(IM)
      .
      .       Calculations here using RHO1(K)
      .
10 CONTINUE
```

Restructured FORTRAN

```
    DO 10 K=1,KMAXR
    IMN(K)=FIT(K,L).AND.IMMASK
 10 CONTINUE
    LEN=KMAXR
    CALL INDEXED(RHO,IMN,RHO1,LEN)
    DO 100 K=1,KMAXR
       .
       .      Rest of calculations from original
       .      loop 10 using RHO1(K)
100 CONTINUE
```

Fig. 13.
Indirect indexing using CAL routine INDEXED.

Original FORTRAN

```
      DO 5 L=1,LHAX
      DO 5 K=1,KMAX
      MNKL(K,L)=FIT(K,L).AND.IMMASK
      ICHKL(K,L)=FIT2(K,L).AND.IPFMASK
      IF(MNKL(K,L).NE.0.AND. ICHKL(K,L).EQ.0)GO TO 5
      CHI(K,L)=0.0
      ALK(K,L)=0.0
      ALL(K,L)=0.0
      BTL(K,L)=0.0
    5 CONTINUE
```

Restructured FORTRAN

```
      DO 5 L=1,LMAX
      DO 5 K=1,KMAX
      MNKL(K,L)=FIT(K,L).AND.IMMASK
      ICHKL(K,L)=FIT2(K,L).AND.IPFMASK
      L50=.FALSE.
      IF(MNKL(K,L).EQ.0)L50=.TRUE.
      IF(ICHKL(K,L).NE.0)L50=.TRUE.
      IF(L50)CHI(K,L)=0.0
      IF(L50)ALK(K,L)=0.0
      IF(L50)ALL(K,L)=0.0
      IF(L50)BTL(K,L)=0.0
    5 CONTINUE
```

Vectorized FORTRAN

```
      INTEGER OUTRXS,OUTRMX,LAST,OUTRWD,ISTRIP,OUTRNX
      DO 5 K=1,KMAX
      CALL LOPLXXX(LMAX,OUTRXS,OUTRMX)
      CALL SETLX(OUTRXS)
      LAST=OUTRXS
      OUTRWD=OUTRXS
      ISTRIP=1
      DO 90000 OUTRNX=1,OUTRMX
      CALL ILDYXX0(FIT(K,ISTRIP),8T)
      CALL SANYX01(IMMASK)
           •
           •
           •
      CALL CSMQX52(0,0)
      CALL ISTV2XX(0,B1,BTL(K,ISTRIP))
      ISTRIP=OUTRWD+ISTRIP
      OUTRWD=24
      CALL SETLX(64)
      LAST=64
90000 CONTINUE
90001 CONTINUE
    5 CONTINUE
```

Fig. 14.
Use of logicals in place of transfers.

Original FORTRAN

```
     IF(NSQ.NE.0)GO TO 990
     IF(RT(K,L).GE.0)GO TO 990
     RT(K,L)=0.0
    .UT(K,L)=ABS(UT(K,L))
     RTC(K,L)=0.5*ROLD
990  CONTINUE
```

Restructured FORTRAN

```
     IF(NSQ.NE.0)GO TO 987
     DO 967 K=1,KMAX
     IF(RT(K,L).LT.0.0)RT(K,L)=0.0
     IF(RT(K,L).LT.0.0)UT(K,L)=ABS(UT(K,L))
     IF(RT(K,L).LT.0.0)RTC(K,L)=0.5*RTEMP(K)
967  CONTINUE
987  CONTINUE
990  CONTINUE
```

Fig. 15.
'IF' test pulled into DO-Loop in place of transfer.

Original FORTRAN

```
      DO 115 L=2,LMAX
      DO 115 K=2,KMAX
      THPL(K)=THCT(K,L+1)**4
      THMI(K)=THCT(K,L)**4
      THU(K,L)=AMAX1(THPL(K),THMI(K))
      TH3(K)=SQRT(DLHRK(K,L))
      UNOM(K,L)=THU(K,L)*.977*ABS(THPL(K)-THMI(K))*TH3(K)*
     1 ALOJK(K,L)*FLUXN
115   CONTINUE
```

Vectorized FORTRAN (chaining broken for fourth power)

```
      DO 90055 L=2,LMAX
      NITER=KMAX-1
      CALL LOPCXXX(NITER,OUTRXS,OUTRMX)
      CALL SETLX(OUTRXS)
      LAST=OUTRXS
      OUTPWD=OUTRXS
      ISTRIP=2
      DO 90057 OUTRNX=1,OUTRMX
      INNRMX=OUTRWD+ISTRIP-1
      DO 90058 K=ISTRIP,INNRMX
      THPL(K)=THCT(K,L+1)**4
      THMI(K)=THCT(K,L)**4
90058 CONTINUE
      CALL ILDVX00(THPL(ISTRIP))
      CALL ILDVX01(THMI(ISTRIP))
      CALL VRSV012
      CALL SVMP2
      CALL CVMG013
      CALL ISTV30X(0.0,THU(ISTRIP,L))
      CALL ILDVX01(DLHRK(ISTRIP,L))
      CALL VSUMT
```

Fig. 16.
No vectorization of exponents other than two.

Restructured FORTRAN

```
      DO 105 L=1,LMAX-1
      DO 105 K=1,KMAXP1

      THPL,THMI MODIFIED FOR VECTORIZER
      TS0=THCT(K,L+1)
      TS0=TS0*TS0
      THPL(K)=TS0*TS0
      TS1=THCT(K,L)
      TS1=TS1*TS1
      THMI(K)=TS1*TS1
      THU(K,L)=AMAX1(THPL(K),THMI(K))
      TS2=SQRT(DLHRL(K,L))
      UNOM(K,L)=THU(K,L)*.973*ABS(THPL(K)-THMI(K))*TS2*
     1 ALOJL(K,L)*FLUXN
105   CONTINUE
```

Vectorized FORTRAN

```
      DO 90047 L=1,LMAXP1
      CALL LOPCXXX(KMAXP1,OUTRXS,OUTRMX)
      CALL SETLX(OUTRXS)
      LAST=OUTRXS
      OUTPWD=OUTRXS
      ISTRIP=1
      DO 90048 OUTRNX=1,OUTRMX
      CALL ILDVX00(THCT(ISTRIP+1,L))
      CALL VRPV001
      CALL VRPV112
      CALL ISTV20X(0.0,THPL(ISTRIP))
      CALL ILDVX03(THCT(ISTRIP,L))
      CALL VRPV334
      CALL VRPV445
      CALL ISTV50X(0.0,THMI(ISTRIP))
      CALL ILDVX00(THPL(ISTRIP))
      CALL VRSV057
      CALL SVMP7
      CALL CVMG051
      CALL ISTV10X(0.0,THU(ISTRIP,L))
      CALL ILDVX01(DLHRL(ISTRIP,L))
      CALL VSUMT
```

Fig. 17.
Vectorization works for second power.

ORIGINAL FORTRAN

```
      DIMENSION A(50), B(50), C(50)
      ...
      DO 20 I = 1, N
      IF (A(I).LT.X) GO TO 20
      IF (B(I).LT.X) TO TO 10
      C(I) = C(I) + B(I)
10    C(I) = C(I) + A(I)
20    CONTINUE
```

RESTRUCTURED FORTRAN

```
      DO 20 I = 1,N
      Y(I) = 0.0
      IF (A(I).GE.X) Y(I) = A(I)
      IF (B(I).GE.X) Y(I) = Y(I) + B(I)
      C(I) = C(I) + Y(I)
20    CONTINUE
```

Fig. 18.
One method of replacing forward transfers.

```
*PARAMETER'
      KMAX=50
      LMAX=100
      END
CLICHE BLANK
      DIMENSION A(KMAX,LMAX), AAAO(LMAX)
      END
```

Pre-Computing Dimensions

```
       .
       .
       .
      SUBROUTINE X
      USE BLANK
       .
       .
       .
      DO 10 K=1, KMAX
      DO10 L=1, LMAX
      AAAO(L)=0.5*A(K,L)
10    CONTINUE
       .
       .
       :
      END
```

Original FORTRAN

```
      DO 10 K=1, KMAX
      CALL ILDVXXO (A(K,1), 50)
      CALL SRPVX01 (0.5)
      CALL ISTVIXX (0,0,AAAO)
10    CONTINUE
```

Vector Output (50 is the stride)

```
      /
     .SUBROUTINE X
      DIMENSION A(60,100), AAAO(100)
       .
       .
       .
      DO 10 K=1, KMAX
      CALL ILDVXXO (A(K,1),50)
      CALL SRPVX01 (0.5)
      CALL ISTVIXX (0,0,AAAO)
10    CONTINUE
       .
       .
       .
      END
```

Routine with KMAX and LMAX changed.

Fig. 19.
Stride problem from editor-VECTORIZER interface.

```
C
       DIMENSION  RARR(81,121),DLRR(81,121),DLRRL(81,121),ARDRL(81,121),D
      *LRRK(81,121),ARDRK(81,121),THL(4,81,121),MNKL(81,121),TH4(81,121),
      *OPA(4,81,121),AJTH(81,121),ALOJ1(81,121),ALOJ2(81,121),ALOJ3(81,12
      *1),ALOJ4(81,121),HLP(81,121),SGT(81,121),GTT(81,121),ALOJL(81,121)
```

Original FORTRAN

```
C
       DO 65 L=2,LMAX
       DO 65 K=2,KMAX
       HLP(K,L)=FLT(K,L)*OPA(4,K,L)
       FLT(K,L)=FLT(K,L)*OPA(2,K,L)
       FLT(K,L)=(FLT(K,L).AND..NOT. MASK) .OR.
      1 (SHIFT(HLP(K,L),-THALF).AND. MASK)
    65 CONTINUE
C
```

Vectorizer Output

```
       DO 65 K=1,KMAXP1
       CALL LOPCXXX(LMAXP1,OUTRXS,OUTRMX)
       CALL SETLX(OUTRXS)
       OUTPWD=OUTRXS
       ISTRIP=1
       DO 90033 OUTRNX=1,OUTRMX
       INNRMX=OUTRWD+ISTRIP-1
       CALL ILDVXX0(FLT(K,ISTRIP),81)
       CALL ILDVXX1(OPA(4,K,ISTRIP),324)
       CALL VRPV012
       CALL ISTV2XX(0,81,HLP(K,ISTRIP))
       CALL ILDVXX3(OPA(2,K,ISTRIP),324)
       CALL VRPV034
       CALL ISTV4XX(0,81,FLT(K,ISTRIP))
       DO 90034 L=ISTRIP,INNRMX
       FLT(K,L)=(FLT(K,L).AND.(.NOT.MASK)).OR.(SHIFT(HLP(K,L),(-IHALF))
        .AND.MASK)
 90034 CONTINUE
       ISTRIP=OUTRWD+ISTRIP
       OUTRWD=64
       CALL SETLX(64)
 90033 CONTINUE
 90035 CONTINUE
    65 CONTINUE
C
```

Fig. 20.
Typical example of the Stride problem.

Dimensioning for Special Options (Note 1TH)

```
      COMMON/LCM1/  RT(81,121),ZT(81,121),AMT(81,121),UT(81,121),VT(81,1
     *21),TT(81,121),ET(81,121),THAT(81,121),THCT(81,121),PT(81,121),CHI
     *(81,121),FT(81,121),FJT(81,121)
C
      COMMON/LCMA/  DET(81,121),DYHT(81,121),CFT(81,121),VIM(81,121),FJO
     *IT(81,121),FLT(81,121),GT(81,121),Q1T(81,121),Q2T(81,121),Q3T(81,1
     *21),Q4T(81,121)
C
      COMMON/LCMB/  FIT2(81,121),YMASS(4,81,121)
C
C             ARRAYS FOR 3-T PHYSICS
C
      COMMON/B3T/  TTH(1,1),TTO(1,1),TRO(1,1),TI(1,1),TIH(1,1),TRH(1,1),
     *ERI(1,1),CVH(1,1),SGAMEI(1,1),SGAMER(1,1),SDE(1,1)
C
                 .
                 .
                 .
```

Original FORTRAN

```
700   DO 1000 K=KK1,KK2
C
C                       CALCULATE NEW DENSITY
C
      TS1=AMT(K,L)
      IF(TS1.EQ.0)ITM1(K)=0
      IF (TS1.EQ.0) TS1=1.
       TT(K,L)=TM2(K)/TS1
      TC=0.5*(TM1(K)+TT(K,L))
       FLT(K,L)=TC
      IF (THREET) TTH(K,L)=TC
      FJTH(K,L)=0.5*(FJT(K,L)+TMO(K))
 1000 CONTINUE
                 .
                 .
                 .
       DO 950 K=KK1,KK2
C                CHECK IF EVERYTHING OK
C       IF(ITM2(K).EQ.0)GO TO 930
       IF(ITM1(K).EQ.0)GO TO 930
       IF(FJT(K,L) .GT. 0 .AND. TT(K,L) .GT. 0) GO TO 120
C
                 .
                 .
                 .
```

Fig. 21.
Example of values being set for special options.

```
      INTEGER NITER,OUTRXS,OUTRMX,OUTRWD,ISTRIP,OUTRNX
      REAL AAA0,AAA1,AAA2,AAA3
      DIMENSION AAA0(1),AAA1(1),AAA2(1),AAA3(1)
      EQUIVALENCE (AAA3,AAA2)
                •
                •
                •
  700 NITER=KK2+1-KK1
      CALL SETLX(NITER)
C
C                      CALCULATE NEW DENSITY
C
      CALL ILDVX00(AMT(KK1,L))
      CALL SVMZ0
      CALL ILDVX01(ITM1(KK1))
      CALL CSMGX12(0)
      CALL ISTV20X(0.0,ITM1(KK1))
      CALL SVMZ0
      CALL CSMGX03(1.0)
      CALL ILDVX04(TM2(KK1))
      CALL VRXV35
      CALL VRPV456
      CALL VRIV537
      CALL VRPV671
      CALL ISTV10X(0.0,TT(KK1,L))
      CALL ILDVX00(TM1(KK1))
      CALL VRAV015
      CALL SRPVX56(0.5)
      CALL ISTV60X(0.0,AAA3(KK1))
      CALL ISTV60X(0.0,FIT(KK1,L))
      IF(.NOT.(THREET))GO TO 90001
      CALL ILDVX07(AAA3(KK1))
      CALL ISTV70X(0.0,TTH(KK1,L))
90001 CONTINUE
      CALL ILDVX00(FJT(KK1,L))
      CALL ILDVX01(TM0(KK1))
      CALL VRAV012
      CALL SRPVX23(0.5)
      CALL ISTV30X(0.0,FJTH(KK1,L))
 1000 CONTINUE
```

Fig. 22.
Incorrect vectorization of code in Fig. 21.

```
700     DO 1000 K=KK1,KK2
C
C                         CALCULATE NEW DENSITY OF ZONE
C
        TS1=AV1(K,L)
        IF(TS1.EQ.0)ITM1(K)=0
        IF (TS1.EQ.0) TS1=1.
         TT(K,L)=TM2(K)/TS1
        TC=0.5*(TM1(K)+TT(K,L))
         FLT(K,L)=TC
        FJTH(K,L)=0.5*(FJT(K,L)+TM0(K))
 1000   CONTINUE
        DO 949 K=KK1,KK2
        TM2(K)=TM0(K)-FJT(K,L)
        IF(TM2(K).LE.0)TM2(K)=-1.
        TM1(K)=.25*(FJT(K,L)/TM2(K))*DTNUP
949     CONTINUE
C            .
C            .
             .
        DO 950 K=KK1,KK2
C
        IF(THREE1)ITH(K,L)=FLT(K,L)
C                 CHECK IF EVERYTHING OK
         IF(ITM2(K).EQ.0)GO TO 930
        IF(ITM1(K).EQ.0)GO TO 930
        IF(FJT(K,L) .GT. 0 .AND. TT(K,L) .GT. 0) GO TO 120
               .
               .
               .
```

Fig. 23.
Restructuring to correct special option problem of Figs. 21 and 22.

VECTORIZED PIC SIMULATION CODES ON THE CRAY-1

by

D. W. Forslund
C. W. Nielson
University of California
Los Alamos Scientific Laboratory
Los Alamos, New Mexico 87545

ABSTRACT

The PIC simulation code WAVE has been almost completely vectorized for the CRAY-1 and is being routinely used on a production basis. We discuss here the vectorizing techniques for the particle mover and the field solver as well as the I/O routines which result in the code being nearly CPU-bound. The procedure used to vectorize the particle mover is to rewrite it in a series of small loops which then are readily converted by a vectorizer program into special vector macros recognized by the FTN compiler at LASL. This allows selective vectorization of different sections of code to determine the optimal vectorization strategy. As is well known the interpolation technique used in PIC simulation for the fields, charges and currents is not vectorizable. We find that the optimal strategy for the FTN compiler is to keep this interpolation process completely in scalar mode. If we separate the scalar fetch and then vectorize the interpolation, we find a degradation of 30% in speed. On the CRAY-1 we obtain speeds of 5.5 s/particle for a 2-D electrostatic mover, 11.5 s/particle for a 2 1/2-D (with all field quantities) nonrelativistic electromagnetic mover and 12.4 s/particle for a 2 1/2-D relativistic electromagnetic mover. We also use a fully vectorized Poisson solve algorithm which uses FFT (Berglund real form) in one direction and tridiagonal solve in the other. Vectorization is achieved in the direction normal to the transform or tridiagonal solve. A 256 x 256 Poisson solve takes 100 ms and a 64 x 64 Poisson solve takes 5.1 ms. The FFT takes 2/3 of the time and the tridiagonal solve takes 1/3 of the time. In order to sustain these speeds for large problems, an efficient I/O algorithm is needed on the CRAY-1. The algorithm we have implemented in production is triple-buffering with two disk channels. Sustained transfer rates of 375 000 words/sec/channel are obtained which allow for nearly complete overlap with the relativistic mover. Exploratory tests have shown that it will be possible to obtain sustained rates of 450 000 words/sec on each of four channels driven simultaneously and overlapped with computation. The size of the required buffers to achieve this is dependent on the details of the operating system.

A CRAY-1 SIMULATOR AND ITS APPLICATION TO DEVELOPMENT OF HIGH-PERFORMANCE CODES

by

D. A. Orbits
D. A. Calahan
Department of Electrical and Computer Engineering
University of Michigan
Ann Arbor, Michigan

ABSTRACT

A logical/timing simulator for the CRAY-1 is described. Used in conjunction with a companion cross-assembler, the simulator has provided an invaluable non-resident programming aid for study of high performance vector linear algebra algorithms and for the development, from a Fortran model, of large assembly-coded benchmarks for aerodynamic simulation.

I. INTRODUCTION

It has become increasingly clear to algorithm developers for vector processors that management of data flow and myriad vector/scalar resources for critical computation kernels can often be significantly improved by assembly coding. Speedups of 2-5:1 are not uncommon, especially for linear algebra codes.

Because timing considerations are far more important to vector than to scalar processing, such assembly coding becomes an extraordinary task even for the skilled programmer. Indeed, analyzing the data flow associated with a proposed algorithm is so demanding that either the algorithm developer despairs of optimizing code performance, or else becomes so involved in a timing analysis that the larger issues of algorithm performance are put aside.

These problems arose in the study of vectorized linear algebra algorithms and in the conversion of aerodynamic simulation ("numerical wind tunnel") codes for the CRAY-1. We believe that our approach - a logical/timing simulator - provides a timely solution and suggests an opportunity for closer coupling in the future between algorithm and architectural designers.

II. THE U OF M CRAY-1 SIMULATOR

A. Introduction

This portion of the paper will discuss in general terms the features of the U of M Cray-1 Simulator. We will address each of the following three questions:

1. Why we chose to build a Cray-1 simulator?
2. What features does the simulator provide?

3. How well does the simulator meet its objectives in terms of cost, accuracy and usefulness?

Lack of space prevents a detailed discussion of the simulator's features. Reference <1> provides an in depth discussion of the material presented here.

B. Why Build a Cray-1 Simulator

The decision to build a Cray-1 simulator was motivated by the following four considerations:

1. Most important is the need to analyze the performance of Cray-1 algorithms. The simulator is superior to the Cray-1 in this respect because software instrumentation in the simulator permits a detailed study of Cray-1 resource usage and conflict.
2. For those computational kernels which must be carefully designed and coded, the programmer can use the simulator to analyze instruction delays and re-organize instructions as necessary to minimize resource conflicts.
3. When debugging programs, it is highly desirable to have interactive control of program execution. A Cray-1 batch environment is not very conducive to the diagnosis of program bugs. The simulator should provide a symbolic debugging environment with a rich set of debugging commands. A companion Cray-1 cross assembler <2> was developed which provides symbol information to the simulator.
4. Finally, with the simulator it is a simple matter to architect Cray-1 modifications in software so as to study their impact on algorithm performance. As we study algorithm behavior, we plan to experiment with variations of the Cray-1 architecture.

We feel that the Cray-1 simulator meets all of these objectives and will next discuss the specific features of the simulator.

C. Cray-1 Simulator Features

The four major features provided by the simulator are:

1. Performance reporting
2. The command language debugging facility
3. The subroutine interface to the simulator
4. The ability to implement Cray-1 architectural modifications

1. Performance Reporting. The principal value of a simulator is its' ability to convey to the algorithm designer and programmer the behavior of algorithms and kernels in the highly concurrent environment of the Cray-1 computer <3>. The Cray-1 simulator currently provides three kinds of activity reporting. In increasing level of reporting detail, the three reports are:

1. Operation counts
2. Data flow summary
3. Clock period activity report.

The operation counts report is useful for measuring Cray-1 functional unit utilization and overall algorithm performance.

The data flow summary is useful for studying aggregate data rates in various segments of the Cray-1. This also permits the study of the data flow across the memory hierarchy boundaries.

The clock period activity (CPACT) report provides a detailed record of the state of the various Cray-1 resources. This can be used to study instruction delays in algorithm kernels to optimize their performance.

Each of these reports will now be discussed in more detail.

1.1 Operation Counts. The operation count report is divided into two sections: (1) a floating point result counts section, and (2) a vector usage counts section.

The floating point result counts section reports the program's use of both vector and scalar floating point operations. For each entry in the table (Figure 1), both the number of results and their percentage are printed.

```
                         FLOATING  POINT  RESULT  COUNTS

              ADDITION      MULTIPLICATION     RECIPROCAL         TOTAL

VECTOR (%)    5530 ( 43.4)    7119 ( 55.9)      63 (  0.5)   12712 ( 99.9)
SCALAR (%)       5 (  0.0)       5 (  0.0)       8 (  0.1)      18 (  0.1)

TOTAL  (%)    5535 ( 43.5)    7124 ( 56.0)      71 (  0.6)   12730 (100.0)
```

Figure 1
Floating Point Result Counts Table

Floating point additions (and subtractions) and reciprocals are counted directly from the instructions that perform them. The multiplication count is adjusted due to the Cray-1 reciprocal approximation.

The vector usage counts section reports the program's use of the Cray-1 vector unit resources. Figure 2 shows the vector usage counts table.

```
                U OF M  CRAY-1  SIMULATOR   (UM138) THU JUN 15/78
                           VECTOR  USAGE  COUNTS

CUM. TIMING        FP ADD  FP MUL  FP DIV    LOG.    SHIFT  I. ADD  V-LOAD  V-STOR

TIME BUSY (CP)      5882    7705     67       67     1277       0   10322    2501
% TIME BUSY       36.20%  47.42%   0.41%    0.41%    7.86%   0.0 %  63.52%  15.39%
NO. RESULTS         5530    7245     63       63     1201       0    9706    2316
NO. VECTORS           68     115      1        1       19       0     154      37
AVERAGE VL         62.84   63.00   63.00    63.00    63.21     0.0   63.03   62.59

RUN TIME (CP)  :  16250
MFLOPS         :  62.67
COMPOSITE AVL  :  62.95
CONCURRENCY    :   1.71
MIPS           :   5.02
```

Figure 2
Vector Usage Counts Table

Each column of the table represents a different vector functional unit. Left to right the units are:

1. Floating point adder
2. Floating point multiplier
3. Floating point reciprocal approximation
4. Vector logical
5. Vector shift
6. Vector integer adder
7. Vector memory path (split between loads and stores).

The rows of the table represent: unit busy time, percent unit busy time of total run time, the number of vector results produced by the unit, the number of vector instructions issued to the unit, and the average vector length processed by the unit.

Five other statistics are printed beneath the table: the run time, the MFLOPS (million floating point operations per second) for the program, the composite average vector length over all vector units, the vector unit concurrency, and the MIPS (million instructions per second) rate.

<u>1.2 Data flow summary</u>. This report (Figure 3) provides a summary of the data traffic over the major Cray-1 data trunks. To aid in identifying the various data paths for which traffic information is provided, the simulator prints a block diagram of the Cray-1 central processor and attaches path labels to each of the data paths. These path labels are referenced on the left hand side of the report preceeding a number which represents the number of operands shipped over that path. The data paths with arrows are uni-directional whereas the paths shown dotted are bi-directional.

The left most column of the figure represents the Cray-1 computational units divided into three groups; vector, scalar and address. The floating point functional units are assumed to be shared between the vector and scalar groups.

The center column of the figure represents the Cray-1 register storage. Top to bottom, four register groups are portrayed; the vector registers, the scalar registers, the T and B registers, and the address registers. The vertical bi-directional data paths (shown dotted) between the four register groups are used for inter-group data transfers.

The right hand column of the figure represents Cray-1 main memory. Memory is shown in four sections only for the purpose of the figure. Any register group may reference any location in Cray-1 main memory.

The labeling scheme is defined as follows:

1. 'A', 'S' and 'V' are for address, scalar and vector.
2. 'O' is for operand and 'R' is for result.
3. 'X' means a bi-directional data path.
4. 'M' means the path is a memory path used by the three register groups tied both to memory and a computational unit. The T and B registers communicate only with memory and other register groups.

For example, 'SMO' is the operand data path to the scalar registers from memory, while 'SO' is the operand data path to the scalar computation units from the scalar registers

Below the data path portion of the report, four other statistics (including branches, fetches, and instruction issues) are printed.

This data may be used to compute many useful percentages and ratios, some of which are discussed in <1>.

<u>1.3 Clock period activity (CPACT)report</u>. The CPACT report provides a detailed description of Cray-1 resource activity at each clock period of the simulation. Because of the substantial amount of output produced by this report, it must be invoked judiciously to avoid producing several hundred pages of output.

Figure 4 shows the format of the report. Across the top of the report, the various column headings are devoted to the Cray-1 resources that may be called into play by an instruction. Time flows down the page with each clock period of simulation time producing an output record. When an instruction issues, a single character tag is assigned to the instruction for the purpose of tracking the instruction's resource use throughout its' execution. Some of the tags are shown underscored, indicating the next instruction to issue requires the resource

```
VO       :     24861
VMO      :      9706

VR       :     14102
VMR      :      2316

SXV      :         7


SO       :        50
SMO      :        15

SR       :        34
SMR      :         0

SXT      :        10

BTO      :        17

SXA      :         5

BTR      :        17

AXB      :        66


AO       :       414
AMO      :        16

AR       :       207
AMR      :         0


MISC.     :        237
BRANCHES  :          4
FETCHES   :         20
ISSUES    :       1020
```

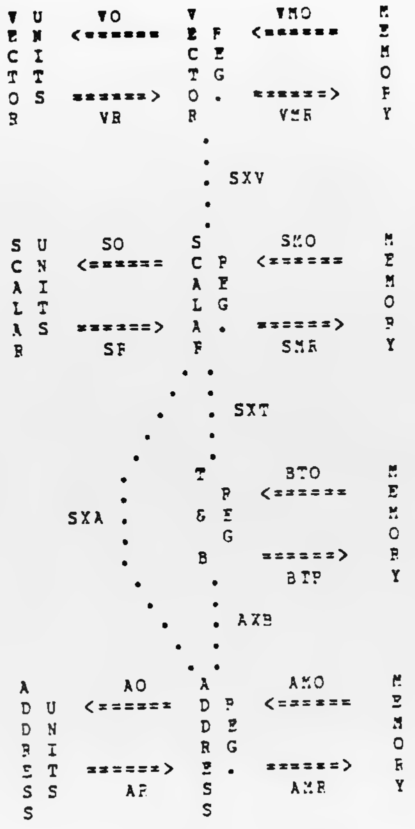


Figure 3
Data Flow Summary

occupied by this tag. As a consequence of this conflict, the next instruction must hold issue until its' resource conflicts vanish.

Below we define the various headings of the CPACT report:

1. ST. - The machine state.
   IS - instruction issue
   blank - instruction hold issue
   FE - instruction fetch sequence
2. TAG - An instruction's resource tracing tag.
3. INSTR. - The Cray-1 instruction mnemonic.
   '<BLANK>' is a pass instruction.
4. P-ADDR - The parcel address of the issuing instruction.
5. CP - The machine clock period.
6. +*/&>+ - The Cray-1 vector functional units.
   (FP add, FP mult, FP recip., Logical, Shift Integer add)
7. V. REG - The Cray-1 vector registers.
8. MEMORY - The memory access network and memory banks for tracking scalar memory references.
9. ARA - The A-register access path.
10. A. REG - The address registers.

```
     T                                          FFF       V. REG      S R R R   MEMORY BANKS   A  A. REG  S  S. REG   V AS             S O BF BB
ST.  A     INSTRUCTION         P-ADDR     CP    PPPVVV                C K K K                  P          R           M 30             T 7 CP ST
     G                                          +*/&>+   01234567     1 A B C 0123456789ABCDEF A 01234567 A 01234567    BB             M 3 GA FX

                                           612|         |            |                        | |        | |         | |              |    4
                                           613|         |            |                        | |        | |         | |              |    4
IS       <BLANK>                           614|         |            |                        | |        | |         | |              |
IS       <BLANK>                           615|         |            |                        | |        | |         | |              |
IS   6   B10  A7                  121A     616|         |            |                        | |        | |         | |              |
IS   7   A7   A4-A0               121B     617|         |            |                        | |       7| |         | |              |
IS   8   B11  A5                  121C     618|         |            |                        |7|       7| |         | |              |
IS   9   B01  A4                  121D     619|         |            |                        | |        | |         | |              |
IS   A   S1   A7                  122A     620|         |            |                        | |        | | A       | |              |
IS   B   B02  A2                  122B     621|         |            |                        | |        |A| A       | |              |
IS   C   S2   #>72                122C     622|         |            |                        | |        |C|         | |              |
IS   D   S1   S2&S1               122D     623|         |            |                        | |        |D|         | |              |
IS   E   A7   S1                  123A     624|         |            |                        |E|        | |         | |              |
IS   F   A7   A7+A0               123B     625|         |            |                        | |       F| |         | |              |
IS   G   B03  A6                  123C     626|         |            |                        |F|       F| |         | |              |
IS   H   VL   A7                  123D     627|         |            |                        | |        | |         | |              |
IS   I   B04  A7                  124A     628|         |            |                        | |        | |         | |              |
IS   J   B05  A3                  124B     629|         |            |                        | |        | |         | |              |
IS   K   B06  A6                  124C     630|         |            |                        | |        | |         | |              |
IS   L   A5   B11                 124D     631|         |            |                        |L|        | |         | |              |
IS   M   A6   A2+A5               125A     632|         |            |                        | |      M | |         | |              |
IS   N   B12  A1                  125B     633|         |            |                        |M|      M | |         | |              |
IS   O   S1         0,A2          125C     634|         |            |O                       | |        | | O       | |              |
IS       <BLANK>                           635|         |            |  O                     | |        | | O       | |              |
IS   P   A2   A2+A0               126A     636|         |            |    O                   | |  P     | | O       | |              |
                                           637|         |            |      O                 |P|  P     | | O       | |              |
IS   Q   A7   B10                 126B     638|         |            |                      O |Q|        | | O       | |              |
IS   R   A5   A7-A5               126C     639|         |            |                      O | |     R  | | O       | |              |
IS   S   A0   A0+A3               126D     640|         |            |                      O |R|S    R  | | O       | |S             |
                                           641|         |            |                      O |S|S       | | O       | |S             |
IS   T   V3   ,A0,A0              127A     642|         |   T        |                        | |        | | O       | |S             |       T
IS   U   A0   A0+A1               127B     643|         |   T        |                        | |U       | | O       | |U             |       T
                                           644|         |   T        |                        |U|U       |O| O       | |U             |       T
                                           645|         |   T        |                        | |        | |         | |U             |       T
                                           646|         |   T        |                        | |        | |         | |U             |       T
                                           647|         |   T        |                        | |        | |         | |              |       T
                                           648|         |   T        |                        | |        | |         | |              |       T
                                           649|         |   T        |                        | |        | |         | |              |       T
                                           650|         |   T        |                        | |        | |         | |              |       T
IS   V   S2         0,A2          127C     651|         |   *        |V                       | |        | |  V      | |              |
IS       <BLANK>                           652|         |   T        |  V                     | |        | |  V      | |              |
IS   W   S0   V0,A0               130A     653|         |   T        |    V                   | |        | |W V      | | W            |
                                           654|         |   T        |      V                 | |        | |W V      | | W            |
IS   X   V0   ,A0,A0              130B     655|         |X  T        |                       V| |        | |W V      | | W            |       X
                                           656|         |X           |                       V| |        | |W V      | | W            |       X
                                           657|         |X           |                       V| |        |W|W V      | | W            |       X
                                           658|         |X           |                       V| |        | |  V      | | W            |       X
                                           659|         |X           |                        | |        | |  V      | | W            |       X
                                           660|         |X           |                        | |        | |  V      | |              |       X
                                           661|         |X           |                        | |        |V|  V      | |              |       X
                                           662|         |X           |                        | |        | |         | |              |       X
```

Figure 4
CPACT Report

11.	SRA	- The S-register access path.
12.	S. REG	- The scalar registers.
13.	VM	- The vector mask register flag.
14.	AOB	- Register A0 busy flag (Branch data invalid).
15.	SOB	- Register S0 busy flag (Branch data invalid).
16.	STH	- The storage hold flag.
17.	073	- The vector mask read inhibit flag.
18.	BCG	- The parcel buffer change flag.
19.	FPA	- The fetch sequence pause flag.
20.	BSF	- The block sequence flag for vector memory accesses.
21.	BTX	- The B and T register block transfer flag.

2. Cray-1 simulator command language. In developing algorithms for the Cray-1, we needed a command language that provided good facilities for debugging assembly language programs. Consequently, about one half of the simulator commands are provided for debugging purposes. Space limitations prevent more than a brief mention of the command language features.

The debugging commands allow:

1. The setting of breakpoints with optionally specified automatic commands.
2. The display and alteration of storage locations.
3. Symbolic reference to storage locations using the symbol information provided by a companion Cray-1 cross assembler <2>.
4. Single stepping through programs.
5. The monitoring of bad memory references, floating point overflow and divide check.

The simulation control commands provide:

1. The loading of absolute and relocatable modules produced by the cross assembler.
2. Starting and halting the simulation.
3. Linkage control when the simulator is itself called as a subroutine from another program.
4. Simulator Input/Output control, allowing input command files and output diversion.
5. Cost control commands that allow disabling instruction timing, thereby reducing costs by a factor of ten.

3. Cray-1 Simulator subroutine interface. There are two subroutine interfaces in the Cray-1 simulator. First, there is a command language interface, allowing a higher level program to call the simulator as a subroutine. Second, there is an EXIT interface that allows a Cray-1 program to call a user defined service program via the Cray-1 EXIT instruction. Figure 5 illustrates the simulator environment.

3.1 Simulator command language interface. Calling the simulator as a subroutine has many advantages:

1. The ability to convert only a portion of a Fortran program to Cray-1 assembly language, allows the user to simulate only the converted portion while leaving the remaining program in Fortran to run more efficiently on the host machine.
2. Being able to share the simulated Cray-1 memory between the calling program and the simulator allows efficient data communication across the interface.
3. When studying a given algorithm for application to the Cray-1, it is often convenient to perform any housekeeping and initialization

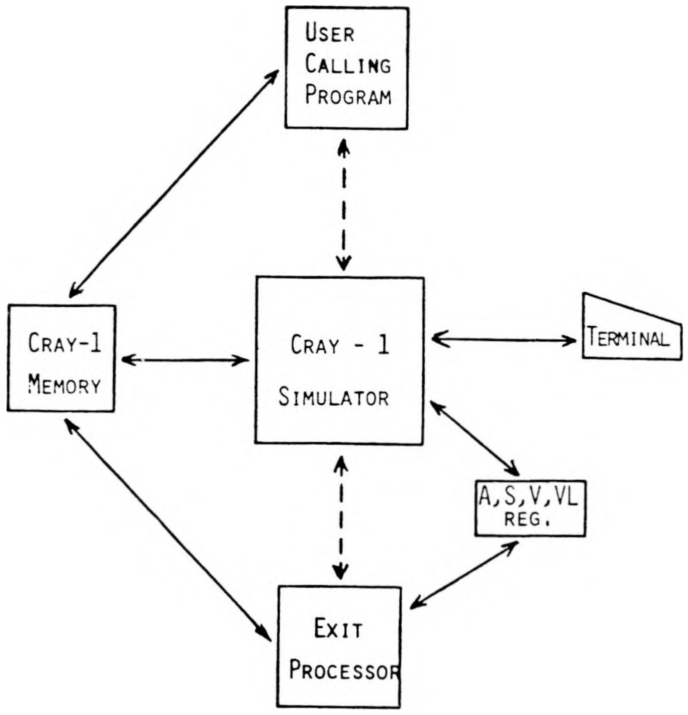


Figure 5
The Cray-1 Simulator Environment

functions in the user's Fortran program. Therefore, only the algorithm need be coded in Cray-1 assembly language.

This interface allows all command language commands to be passed as text strings to the simulator.

4. Exit processor interface. As it is useful to call the Cray-1 simulator as a subroutine, it is also useful to be able to call another program from within the simulated Cray-1 program. This transfer of control from the Cray-1 program to a target program is accomplished through the use of the Cray-1 EXIT instruction.

The Cray-1 assembly language mnemonic for the exit instruction is shown below:

```
EX   ijk
```

The exit code field (ijk) is a nine bit field within the exit instruction. Exit codes may range from zero to 511 decimal. Upon encountering an exit instruction, the simulator checks the exit code field (ijk) for a non-zero value. If ijk is zero, a normal Cray-1 program exit is performed. If ijk is non-zero, the simulator will call the subroutine CRAYEX. Through this interface the simulator passes the A,S,V and VL registers as arguments as well as the exit code for dispatching. As mentioned above, simulated Cray-1 memory sharing may be used to pass data back and forth. This interface is very useful for writing some primitive functions like square root, trigonometric, etc. and running them at full speed on the host machine.

5. Modeling Cray-1 architectural changes. The Cray-1 simulator provides a flexible medium in which one may conveniently experiment with architectural changes to the Cray-1.

As an example, one such change we have explored is the effect of increasing the Cray-1 memory bandwidth to the limit of its' memory system. The 16 memory banks on the Cray-1 have a bank cycle time of four clock periods, so it would be possible to move data at a maximum rate of four words per clock period. This data rate is reduced for unfavorable skip increments, (k) as shown in Table I. The simulation timing was adjusted to move memory data at the rates shown below, but with the same vector start-up time. Since data could be loaded into a register at four words per clock period, chaining from a memory load was disallowed for all data rates. This is no hardship since it is usually possible to reorder vector loads within the code.

k mod 16	Data rate (Words/cp)	k mod 16	Data rate (Words/cp)
0	.25	8	.5
1	4	9	4
2	2	10	2
3	4	11	4
4	1	12	1
5	4	13	4
6	2	14	2
7	4	15	4

Table I
Maximum data rates

This alteration has a very favorable impact on memory bound algorithms.

D. Conclusion

In developing Cray-1 algorithms in assembly language, we have found the debugging features and the subroutine interface to be extremely useful. In one conversion project three numerically complex Fortran programs were converted to assembly language. Each program was converted by a different programmer. Table 2 shows the man hours required.

Routine	Number Fortran Lines	Number Assembly Lines	F:A Ratio	Man Hours
1	98	1500	15:1	50
2	97	239	2.5:1	25
3	40	280	7:1	35

Table 2
Conversion effort

Once developed on the simulator, very little effort was required to get the assembly codes running on the Cray-1.

The timing accuracy of the simulator has been very good. We have observed a timing error on the order of 1/2% on some codes and less on others. We have also timed 66 short instruction test segments to calibrate the simulator timing. In one program the timing error was one clock period out of 318.

In some ways simulating Cray-1 programs is far superior to running them directly on the Cray-1. With simulation, the user has complete freedom to instrument the simulator and generate reports of important events under investigation.

Certainly the major disadvantage of simulation is the time cost. As an illustration, we at the University of Michigan ran the Cray -1 simulator on an Amdahl 470V/6 (IBM System 360/370 compatible), and the cpu time ratio between the Amdahl simulation time and the Cray-1 execution time for the same Cray-1 program is shown below.

Mode 1.	3,000 to 1
Mode 2.	30,000 to 1
Mode 3.	90,000 to 1

where:

Mode 1 simulates the numeric computation using host machine arithmetic but provides no instruction timing. A program executing in one millisecond on the Cray-1 would require three seconds of simulation time on the Amdahl 470.
Mode 2 combines both instruction timing and numeric computation.
Mode 3 produces the detailed activity report for each clock period of the simulation in addition to the instruction timing and numeric computation.

III. DEVELOPMENT OF HIGH PERFORMANCE CODES

A. Introduction

The designer of algorithms for memory-hierarchical, multi-resource machines of the Cray-1 class quickly discovers that algorithm performance is profoundly influenced by language and coding considerations. For example, one finds that in general the better the short-vector performance, the more the advantages of serial algorithms (e.g., faster convergence) can be exploited. (Note that parallel algorithms are a performance subclass of serial algorithms.) This performance thus becomes an excellent candidate for simulator study.

Such optimized kernels can often be developed with the aid of the simulator in a "top-down" manner. On the assumption that vector floating point operations form a minimal set which characterize a code's function, the following is proposed as a two-step high performance assembly code synthesis procedure.

1. Code only the vector instructions, to achieve optimum MFLOP rate. Without addressing and other instructions to impede vector instruction issue, a maximum rate is achieved for any particular choice of vector instruction sequence. Simulated conflicts and remedial programming are the most obvious with such a skeletal program.

2. Inspect the simulator output (in the "steady state" for an instruction loop) for a variety of vector lengths to show any consistent gaps; insert non-vector instructions in these gaps.

B. An Example: The Vector Inner Product *

In the processing of a single dense system on the Cray-1 a vector accumulation of the form

*The reader not familiar with Cray-1 architectural terminology is referred to the results of Table 3.

$$V_j = \sum_{i=1}^{n} \alpha_{ji} V_i \qquad n \le 64$$

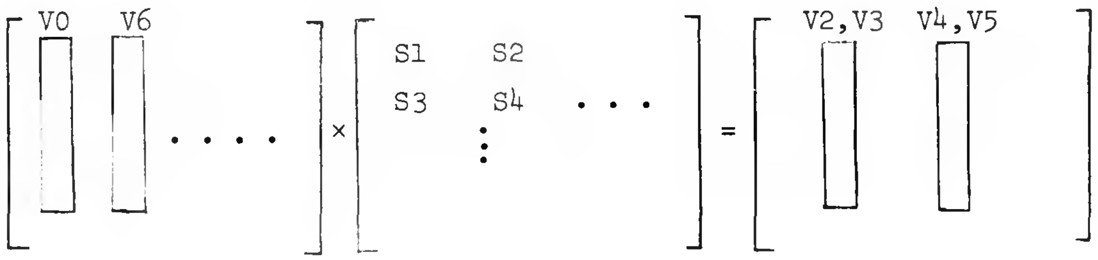
is common, where V_ℓ is a vector and α_{ji} a scalar. For example, this kernel would occur in the multiplication of two matrices or in the column-wise solution of simultaneous equations.

The conventional CAL kernel for this accumulation would have the model form

```
Start       V0 ← memory       vector fetch   )
            V1 ← S1 * V0      scalar*vector  >   "ping"
            V3 ← V2 + V1      vector add     )
            V0 ← memory       vector fetch   )
            V5 ← S3 * V0      scalar*vector  >   "pong"
            V2 ← V3 + V5      vector add     )
            Jump   Start
```

The symmetry of this kernel between the first three and the next three instructions results from the inability of a vector register to function as both a source and destination register in an accumulation. The simulated clock-level report is given in Figure 6(a) for vectors of length 4. In this case, chaining forces long gaps between instruction issue, leaving the floating point paths free more than 50% of the time, and an execution rate of 24 MFLOPS (Table 3).

Inspection of this report rather soon suggests that these gaps could be filled by interleaving chained register operations. We tentatively assign {V2, V3} to store one set of "ping-pong" partial products, and {V4, V5} the other set. For a matrix multiply, each set would accumulate a different column of the result matrix, as illustrated below.



Note that two registers, V0 and V6, are used to prefetch vectors from memory; registers V1 and V7 are intended to chain the interlaced multiplier-add chains. The assignment of two registers for each task reduces the chance of becoming register bound.

Because each memory fetch yields two scalar-vector multiplies (e.g., S1 * V0 and S2 * V0) the memory traffic is 1/2 that of the standard algorithm.

Figure 6(b) depicts the resulting steady-state loop behavior for a vector length of 4 (the assembly code can be deduced from the simulator output). Rather remarkably, the chains for this vector length arrange themselves so that the vector floating point units are never free; clearly this is the maximum

(a) Standard algorithm (43 cl/loop)

```
                              CRAY-1 SIMULATOR

S T                              FFF      V. REG BSRRR
T A INSTRUCTION     P-ADDR  CP   PPPVVV          SCKKK
  G                              +*/&>+ 01234567 F1ABC

I I V0  ,A0,A1         24A    60|G     |I G     |I     |
                              61|G     |I G     |I     |
                              62|      |I *     |I     |
                              63|      |I G     |I     |
                              64|      |I G     |I     |
                              65|      |I G     |I     |
                              66|      |I G     |I     |
                              67|      |I       |I     |
                              68|      |I       |I     |
I J V1  S1*FV0         24B    69| J    |JJ      |      |
                              70| J    |JJ      |      |
                              71| J    |JJ      |      |
                              72| J    |JJ      |      |
                              73| J    |JJ      |      |
                              74| J    | J      |      |
                              75| J    | J      |      |
                              76| J    | J      |      |
                              77|      | J      |      |
I K V3  V2+FV1         24C    78|K     | KKK    |      |
I L V0  ,A0,A1         24D    79|K     |LKKK    |L     |
                              80|K     |LKKK    |L     |
                              81|K     |LKKK    |L     |
                              82|K     |LKKK    |L     |
                              83|K     |L  K    |L     |
                              84|K     |L  K    |L     |
                              85|K     |L  K    |L     |
                              86|      |L  *    |L     |
                              87|      |L  K    |L     |
I M V5  S3*FV0         25A    88| M    |M  K M  |      |
                              89| M    |M  K M  |      |
                              90| M    |M  K M  |      |
                              91| M    |M    M  |      |
                              92| M    |M    M  |      |
                              93| M    |     M  |      |
                              94| M    |     M  |      |
                              95| M    |     M  |      |
                              96|      |     M  |      |
I N V2  V3+FV5         25B    97|N     |  NN N  |      |
I O J         24A      25C    98|N     |  NN N  |      |
                              99|N     |  NN N  |      |
                             100|N     |  NN N  |      |
                             101|N     |  NN N  |      |
                             102|N     |  N     |      |
I P V0  ,A0,A1         24A   103|N     |P N     |P     |
                             104|N     |P N     |P     |
                             105|      |P *     |P     |
                             106|      |P N     |P     |
                             107|      |P N     |P     |
```

(b) HP algorithm (32 cl/loop)

```
                              CRAY-I SIMULATOR

S T                              FFF      V. REG BSRRR
T A INSTRUCTION     P-ADDR  CP   PPPVVV          SCKKK
  G                              +*/&>+ 01234567 F1ABC

I 0 V7  S2*RV0         24D   107|Y0    |0X   YW0|      |
I 1 V2  V3+FV1         25A   108|10    |0111 *W0|      |
                             109|10    |0111 YW0|      |
                             110|10    |0111 Y 0|      |
                             111|10    |0111 Y 0|      |
I 2 V0  ,A0,A1         25B   112|10    |2111 Y 0|2     |
                             113|10    |2 1    0|2     |
                             114|10    |2 1    0|2     |
I 3 V1  S1*RV6         25C   115|13    |231   30|2     |
I 4 V4  V5+FV7         25D   116|43    |23* 4434|2     |
                             117|43    |231 4434|2     |
                             118|43    |231 4434|2     |
                             119|43    |231 4434|2     |
                             120|43    |231 44 4|2     |
                             121|43    |*3  4   |      |
                             122|43    |23  4   |      |
I 5 V7  S2*RV6         26A   123|45    |23  4 55|      |
I 6 V3  V1+FV2         26B   124|65    |2666* 55|      |
                             125|65    |26664 55|      |
                             126|65    | 6664 55|      |
                             127|65    | 6664 55|      |
I 7 V6  ,A0,A1         26C   128|65    | 6664 75|7     |
                             129|65    |   6  75|7     |
                             130|65    |   6  75|7     |
I 8 V1  S1*RV0         26D   131|68    |88 6  75|7     |
I 9 V5  V7+FV4         27A   132|98    |88 *9979|7     |
I A J         24D      27B   133|98    |88 69979|7     |
                             134|98    |88 69979|7     |
                             135|98    |88 69979|7     |
                             136|98    | 8 69979|7     |
                             137|98    | 8   9* |      |
                             138|98    | 8   97 |      |
I B V7  S2*RV0         24D   139|9B    |B8   97B|      |
I C V2  V3+FV1         25A   140|CB    |BCCC *7B|      |
                             141|CB    |BCCC 97B|      |
                             142|CB    |BCCC 9 B|      |
                             143|CB    |BCCC 9 B|      |
```

Figure 6.

Clock period reports of accumulation loops for VL = 4. Note busy time of floating point units under columns FP+ and FP*.

possible execution rate for a vector length of 4. Since the floating point pipelines are reserved for (VL + 4) clocks, this rate is

$$\begin{aligned} \text{MFLOPS} &= 160\left(\frac{\text{VL}}{\text{VL} + 4}\right) \\ &= 80 \end{aligned} \tag{1}$$

When the vector length is increased beyond 4, it turns out that the interlaced chains are not maintained. However, as Figure 7 shows, these chains are not essential to maintenance of a high state of floating point pipeline occupancy! Rather, the fetches, multiplies, and additions now arrange themselves around the loop so that the appropriate instruction is ready to issue (with four exceptions) whenever a pipeline becomes free. Whereas (1) indicates a maximum rate of 107 MFLOPS, the report shows a rate of 102 MFLOPS. This mode of unchained operation continues through VL = 64, clearly a counter-intuitive result.

A series of simulations for $\text{VL} = 2^k$, k = 1, 2, ...6, reveals a number of gaps in the instruction issue suitable for insertion of addressing and loop control instructions; however, the four scalar fetches per loop add 8 clocks for instruction issues (being two-parcel instructions).

Comparative execution rates for the standard code, for the vector-instruction-only HP version, and for the complete HP code are given in Table 3. The maximum rate from equation (1) is also given. The HP version outperforms the standard code by 2.5:1 for VL = 2,4 and by 1.8:1 for VL = 8. Further it is always an improvement over the standard code.

		High Performance		
Vector Length	Standard	Vector-Only	Complete	Maximum
2	12	44	31	53
4	24	80	61	80
8	48	102	87	107
16	87	125	116	128
32	125	140	135	142
64	140	149	146	151

Table 3

Execution rates of accumulation loops (MFLOPS)

The complete HP code has been timed on the CRAY-1; simulated timings were found in error by 1 in 318 clocks.

A matrix multiply code constructed from this CAL kernel shows a 35% decrease in computation time versus the standard CAL algorithm for VL = 4 and 8, and always shows an improvement for $2 \leq \text{VL} \leq 64$.

Other high performance linear algebra codes are being developed in this "top-down" manner and will be reported in reference <4>.

```
                                    CRAY-I SIMULATOR

S T                               FFF     V. REG  BSRRR
T A INSTRUCTION     P-ADDR    CP  PPPVVV          SCKKK
  G                               +*/&>+ 01234567 F1ABC


I W V7  S2*RV0         24D    112|UW    |WT   * W|      |
                              113|UW    |WT   U W|      |
                              114|UW    |WT   U W|      |
                              115|UW    |WT   U W|      |
                              116| W    |WT   U W|      |
I X V2  V3+FV1         25A    117|XW    |WXXX U W|      |
                              118|XW    |WXXX U W|      |
                              119|XW    |WXXX U W|      |
I Y V0  ,A0,A1         25B    120|XW    |YXXX   W|Y     |
                              121|XW    |YXXX   *|Y     |
                              122|XW    |YXXX   W|Y     |
                              123|XW    |YXXX   W|Y     |
                              124|X     |YXXX   W|Y     |
I Z V1  S1*RV6         25C    125|XZ    |YZ*   ZW|Y     |
                              126|XZ    |YZX   ZW|Y     |
                              127|XZ    |YZX   ZW|Y     |
                              128|XZ    |YZX   ZW|Y     |
I 0 V4  V5+FV7         25D    129|0Z    |*ZX 00Z0|Y     |
                              130|0Z    |YZX 00Z0|Y     |
                              131|0Z    |YZX 00Z0|Y     |
                              132|0Z    |YZX 00Z0|      |
                              133|0Z    |YZ  00 0|      |
                              134|0Z    |Y*  00 0|      |
                              135|0Z    |YZ  00 0|      |
                              136|0Z    |YZ  00 0|      |
I 1 V7  S2*RV6         26A    137|01    | Z  * 11|      |
                              138|01    | Z  0 11|      |
                              139|01    | Z  0 11|      |
                              140|01    | Z  0 11|      |
                              141| 1    | Z  0 11|      |
I 2 V3  V1+FV2         26B    142|21    | 2220 11|      |
                              143|21    | 2220 11|      |
                              144|21    | 2220 11|      |
I 3 V6  ,A0,A1         26C    145|21    | 222  31|3     |
                              146|21    | 222  3*|3     |
                              147|21    | 222  31|3     |
                              148|21    | 222  31|3     |
                              149|2     | 222  31|3     |
I 4 V1  S1*RV0         26D    150|24    |44 *  31|3     |
                              151|24    |44 2  31|3     |
                              152|24    |44 2  31|3     |
                              153|24    |44 2  31|3     |
I 5 V5  V7+FV4         27A    154|54    |44 255*5|3     |
I 6 J        24D       27B    155|54    |44 25535|3     |
                              156|54    |44 25535|3     |
                              157|54    |44 25535|      |
                              158|54    | 4  5535|      |
                              159|54    | *  5535|      |
                              160|54    | 4  5535|      |
                              161|54    | 4  5535|      |
I 7 V7  S2*RV0         24D    162|57    |74   * 7|      |
                              163|57    |74   5 7|      |
                              164|57    |74   5 7|      |
                              165|57    |74   5 7|      |
                                                 |      |
                                                 |      |
```

Figure 7.
HP accumulation loop with broken chains for VL = 8 .

IV. CONCLUSION

Simulation has been employed for some time as an aid in the design of computing systems. It now appears useful to algorithm designers as well, and may even become essential when multi-resource scientific machines such as the Flow Model Processor <5> become available in the 1980's. Indeed, it may be that, for the management of such machines, a simulator will become an integral part of innovative code preparation procedures that exploit a high degree of user interaction and intuition concerning computer and problem structure.

Acknowledgements

This work was jointly supported by the Directorate of Mathematical and Information Sciences, Air Force Office of Scientific Research, and by the Air Force Flight Dynamics Laboratory, Wright Patterson Air Force Base, under Grant AFSOR 75-28]2.

The authors acknowledge the assistance of William Ames, who prepared the cross assembler; of Cray Research, Inc., who provided timing information; and the National Center for Atmospheric Research, who provided Cray-1 processor time.

References

1. Orbits, D. A., "A Cray-1 Simulator," SEL Report No. 118, System Engineering Laboratory, the University of Michigan, Sept. 1978.

2. Ames, W. G., "A Cray-1 Cross Assembler," SEL Report No. 120, System Engineering Laboratory, The University of Michigan, Sept. 1978.

3. Russel, R. M., "The Cray-1 Computer System," CACM, Cray Research Inc., January 1978, pp. 63-72.

4. Calahan, D. A., "Performance of Linear Algebra Codes on the Cray-1," Proc. 5th Symposium on Reservoir Simulation, Denver, January 31 - February 2, 1979.

5. "Final Report, Numerical Aerodynamic Simulation Facility, Preliminary Study Extension," two reports prepared by Burroughs Corporation and Control Data Corporation for NASA/Ames Research Center, Moffett Field, CA.

LaRC EXPERIENCE IN THE INSTALLATION OF A STAR-100 COMPUTER

by

M. G. Rowe
NASA/Langley Research Center
Hampton, Virginia 23665

ABSTRACT

A Control Data Corporation (CDC) STAR-100 computer system was installed by CDC at the NASA/Langley Research Center (LRC) in December, 1975. Since that time the use of the STAR-100 as a vector processor to solve research problems at LRC has steadily increased. The evaluation of the STAR-100 system as a major research tool at LRC will be discussed, as well as the performance and reliability of the system.

Both hardware and software system configurations for the STAR-100 system installed at LRC have undergone significant modification and growth since the initial installation. The hardware and software configurations will be discussed. There have been numerous problems to overcome, with some failures.

As a result of LRC's experience in installing and using a vector processor such as the STAR-100, a summary of items that may be of interest to other installations considering such actions will be discussed.

SOME LINPACK TIMINGS ON THE CRAY-1*

by

J. J. Dongarra**
University of California
Los Alamos Scientific Laboratory
Los Alamos, New Mexico

ABSTRACT

This report compares the Los Alamos Scientific Laboratory (LASL) compilers and FORTRAN tools used in running programs on the CRAY-1 computer. A sample of linear equation routines from the LINPACK collection were tried using these compilers and tools to determine what aids give the fastest execution speed for FORTRAN codes run on the CRAY-1.

INTRODUCTION

This report gives timing data obtained from experiments performed on the CDC 7600 and CRAY-1 computers at the Los Alamos Scientific Laboratory (LASL). The timing studies were done using linear equation solvers from the LINPACK package.[1] The CDC 7600 does only scalar processing, whereas the CRAY-1 does both scalar and vector processing. Extremely high execution speeds can be achieved by utilizing the vector capabilities of the CRAY-1.[2] Algorithms for solving linear equations are amenable to a high degree of vectorization. This, together with their wide use at LASL and other Department of Energy (DOE) laboratories, motivated this study.

One of the ground rules for these experiments was that the algorithms would not be changed; no modifications were made to the structure of the algorithms. The LINPACK algorithms are designed to access matrix elements by column whenever possible. The results of other studies suggest that column orientation is beneficial for general computers.[3,4] This study finds that it is beneficial for vector computers as well.

*Also released as LA-7389-MS, August 1978.
**Current Address: Argonne National Laboratory, Argonne, IL 60439

All LINPACK routines used the Basic Linear Algebra Subprograms (BLAS)[5] in carrying out basic computations. Since the BLAS do most of the arithmetic work in the package, it is critical that they run efficiently. A decision to retain the BLAS in LINPACK has not been finalized at this point. If we decide the BLAS are a burden to the package, thereby making the execution time of the routines substandard, we are prepared to remove them automatically using a system designed by Boyle and Dritz called TAMPR.[6] TAMPR can remove calls to the BLAS and replace them with the corresponding inline code. This is done by describing the replacement transformations to TAMPR along with the code that is to be transformed. TAMPR detects the calls and replaces them with appropriate "optimum" inline FORTRAN code (see Appendix).

LASL currently has two FORTRAN compilers and a vectorization package available for the CRAY-1. First, there is the FORTRAN FTN cross-compiler (FTNX). This compiler is a modified Control Data Corporation (CDC) FTN compiler that is used on the CDC 7600 and generates CRAY Assembly Language (CAL) code. This CAL code can then be sent to the CRAY-1 and executed. The FTNX compiler performs some instruction scheduling and optimization of the generated CAL code. It has limitations in that it only generates scalar code.

LASL uses the Massachusetts Computer Associates' (MCA) Vectorizer[7] in conjunction with the FTNX compiler to utilize vector instructions on the CRAY-1. The vectorizer analyzes the FORTRAN source code for vectorization possibilities. If vectorization can be performed on a loop, the Vectorizer transforms the serial FORTRAN code into vector primitives, in the form of subroutine calls, which perform the same function as the serial code. When the code that has been passed through the Vectorizer is given to the FTNX compiler, the compiler replaces the calls to vector primitives with inline vector instructions.[7] This enables the user to access vector features of the CRAY-1 while keeping his source code in standard FORTRAN.

There is also a set of BLAS written in CAL code that takes advantage of the CRAY-1 vector hardware.[8] These routines have the same subroutine linkage conventions that were adopted in the FTNX compiler and therefore can be utilized by FORTRAN code that references the BLAS compiled by the FTNX compiler.

The second compiler, which runs on the CRAY-1, was developed by Cray Research, Inc. (CRI).[9] This compiler does no instruction scheduling, but does perform vectorization of the FORTRAN source code. The CRI compiler will examine FORTRAN loops for possible vectorization and, if it can, generate the appropriate CAL code to utilize vector hardware instructions. The user may also turn vectorization off, that is, the compiler generates only scalar instructions. At the present time the CRI compiler does very limited code optimization, such as common subscript elimination. Work is currently going on at CRI to include instruction scheduling and global optimization. Unfortunately, the subroutine linkage conventions of the FTNX compiler and the CRI compiler are not compatible. Therefore, the CAL BLAS used with the FTNX compiler cannot be utilized by the CRI compiler.

TIMING STUDIES

Timings were carried out on the CRAY-1 for four routines in LINPACK. The four routines deal with real general square matrices and perform the following operations: decompose a matrix into its LU factors and estimate the condition number (SGECO), decompose a matrix into its LU factors (SGEFA), solve a system given the factorized matrix (SGESL), and compute the determinate and inverse given the factorized matrix (SGEDI). The amount of work required for routines SGECO, SGEFA, and SGEDI is $O(N^3)$ and for SGESL is $O(N^2)$, where N is the order of the matrix.

Seven different implementations of these four routines were timed on the CRAY-1. Two different versions were used: one with calls to the BLAS and one with the BLAS replaced by inline code. These were compiled on both the FTNX and CRI compilers. In addition, on the FTNX compiler two versions of the BLAS code were used, one with a FORTRAN unrolled structure (see Appendix) and another with the BLAS coded in CAL. The inline version for the FTNX was also passed through MCA's Vectorizer. In the CRI compiler, runs were made with vectorization turned on and off. Table I summarizes the environments in which the codes were executed.

Figures 1 through 4 and Tables II and III summarize the results of the timings for routines SGECO, SGEFA, SGESL, and SGEDI. Timings were done on matrices of order 50 through 350 in steps of 50. The X-axis displays on a linear scale the order of the matrix timed, and the Y-axis displays on a log scale the time, T, divided by N^3 or N^2, depending on the work done by the routine.

TABLE I

CODE TIMING ENVIRONMENTS

Compiler	Graph Legend	Characteristics
FTNX	+	FORTRAN BLAS*
FTNX	∇	CAL BLAS
FTNX	□	inline code replaces calls to the BLAS
FTNX	X	inline code vectorized
CRI	Δ	FORTRAN BLAS*
CRI	○	inline code replaces calls to the BLAS, vectorization turned off
CRI	◇	inline code replaces calls to the BLAS, vectorization turned on

*FORTRAN BLAS have been unrolled.

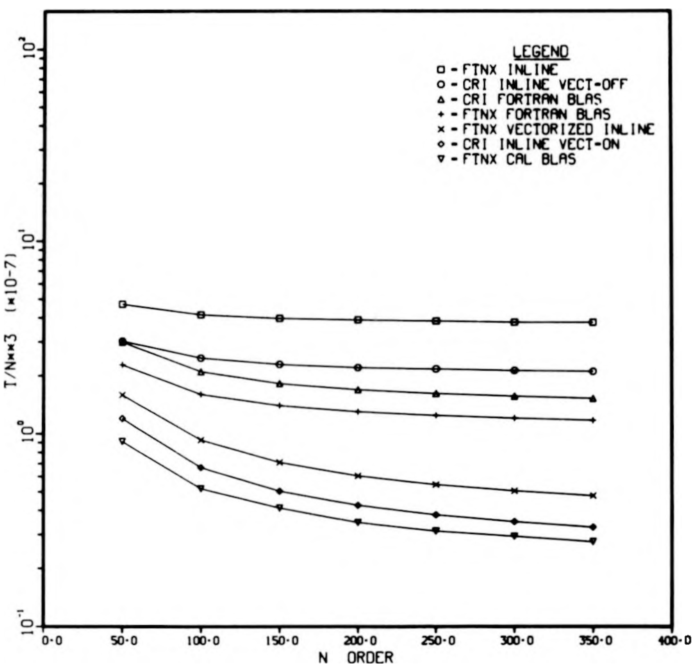


Fig. 1. SGECO timing.

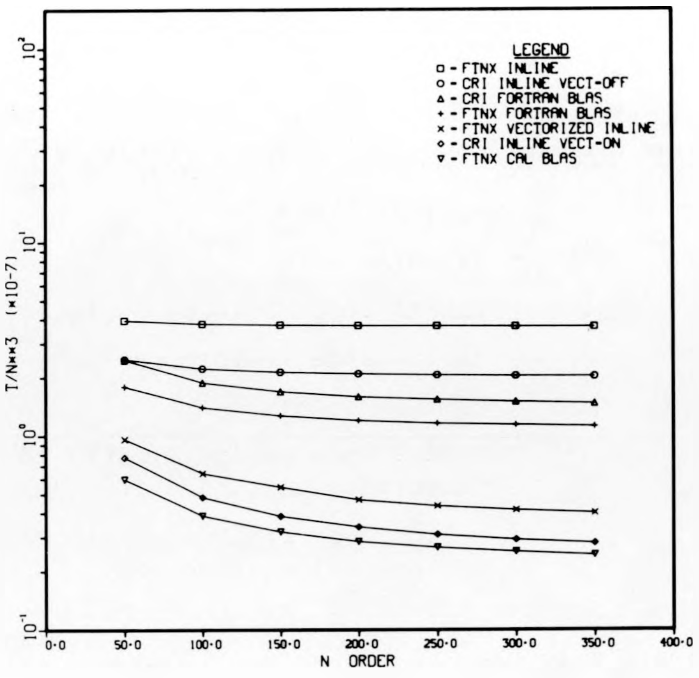


Fig. 2. SGEFA timing.

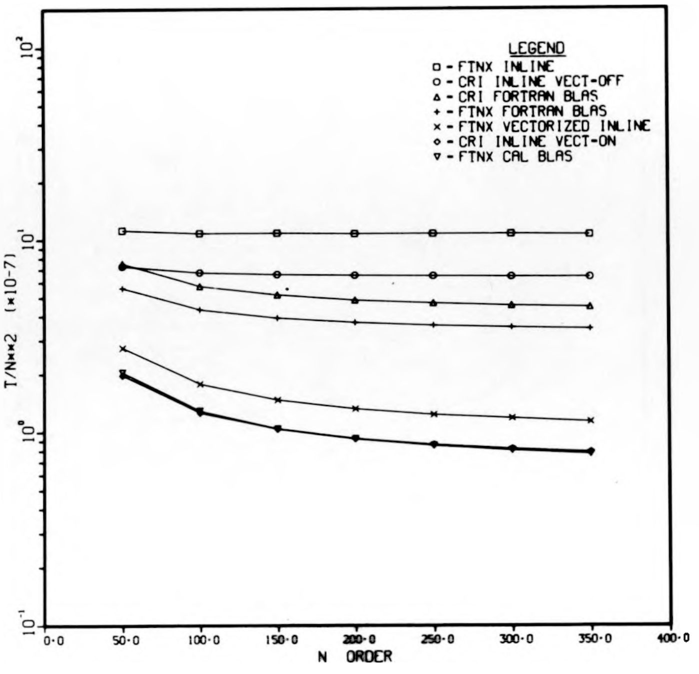


Fig. 3. SGESL timing.

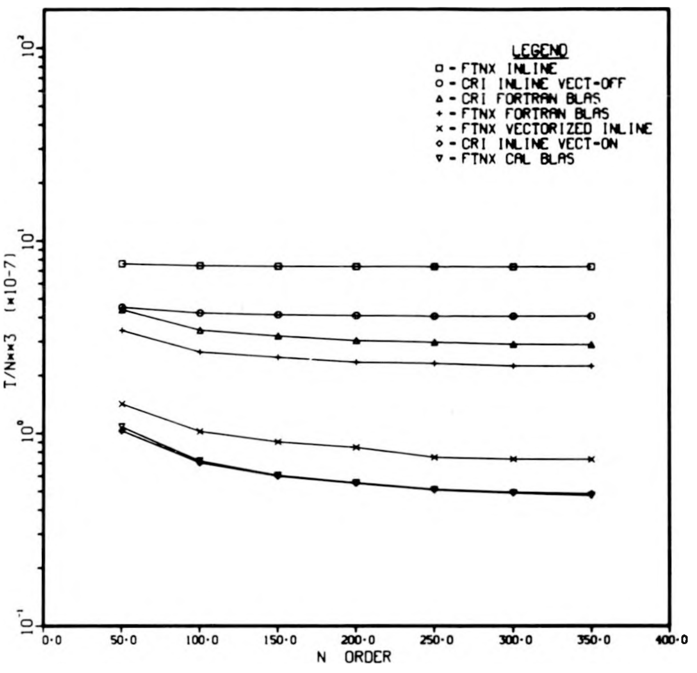


Fig. 4. SGED1 timing.

TABLE II

RATIOS OF EXECUTION TIME FOR SGEFA AT ORDER 350
RUN UNDER DIFFERENT CONDITIONS

	FTNX Inline	CRI Inline VECT=OFF	CRI FORTRAN BLAS	FTNX FORTRAN BLAS	FTNX Vector-ized Inline	CRI Inline VECT=ON
CRI Inline VECT=OFF	1.2					
CRI FORTRAN BLAS	1.7	1.4				
FTNX FORTRAN BLAS	3.3	2.7	1.9			
FTNX Vectorized Inline	9.1	7.5	5.4	2.8		
CRI Inline VECT=ON	13.1	10.8	7.8	4.0	1.4	
FTNX CAL BLAS	15.1	12.4	8.9	4.6	1.7	1.2

(Row entries are so many times faster than column entries.)

TABLE III

MILLIONS OF FLOATING POINT OPERATIONS PER SECOND (MFLOPS) ACHIEVED IN DIFFERENT ENVIRONMENTS

	MFLOPS			
	SGECO	SGEFA	SGESL	SGEDI
FTNX Inline	1.8	1.8	1.9	2.0
CRI Inline VECT=OFF	3.2	3.3	3.1	3.7
CRI FORTRAN BLAS	4.4	4.5	4.5	5.2
FTNX FORTRAN BLAS	5.7	6.0	5.8	6.8
FTNX Vectorized Inline	14.0	16.6	17.7	20.6
CRI Inline VECT=ON	20.4	23.8	25.4	31.1
FTNX CAL BLAS	24.2	27.4	26.0	31.7

As can be seen from Figs. 1-4 and Tables II and III, there is a factor of 15 between the slowest and fastest execution speeds when run under different conditions. This wide range in execution rates is due to execution at scalar speeds compared to vector speeds. It should be pointed out that there is a version of Gaussian elimination, coded entirely in CAL, that executes around the 120-MFLOP range.[10] This version uses an algorithm designed to perform well with the CRAY-1's architecture.

We would expect the FTNX inline coding to be faster than the CRI inline VECT=OFF. They both deal exclusively with scalar code, but the FTNX compiler performs scheduling and optimization while the CRI compiler does not. The timings indicate the CRI inline VECT=OFF is faster by a factor of 1.2 than the FTNX inline. After studying the generated assembly language code for both compilers it became apparent why the timings were contrary to our expectations. The FTNX compiler does not keep the loop index in a register. Consequently, it must load and store the loop index each time the loop iterates. The CRI compiler, on the other hand, maintains the loop index in a register during the loop execution. The inline code in this comparison is typified by one-operation loops, which are executed $O(N^2)$ times. The FTNX compiler for the inline case is spending a large amount of time fetching and storing the loop index; this loop overhead can take as long to be performed as the operation itself. Because the CRI compiler keeps the index in a register, it can fetch and store in two cycles, thus giving it the advantage.

In the case where FORTRAN BLAS are used, the FTNX compiler is faster than the CRI compiler. The FORTRAN BLAS used are an unrolled version (see Appendix). The unrolled BLAS are perfect for concurrent operations and segmentation, since many results are defined per pass through the loop. Because the FTNX compiler does optimization and instruction scheduling and the CRI compiler does not, we would expect the FTNX execution to be faster even though the FTNX compiler forces a fetch and store of the loop index from memory each time through the loop. The loop-indexing operation in the unrolled case is a small part of the loop execution time. There is close to a factor of 2 in execution speeds between the FTNX and CRI compilers using the FORTRAN BLAS.

For the case of the vectorized FTNX inline compared with the inline CRI VECT=ON, the CRI code executes faster by a factor of 1.5. It appears that the vectorized FTNX inline code pays a substantial overhead for using FORTRAN code to manage vector segments (that is, vectors on the CRAY-1 must be partitioned into segments of length ≤ 64; see Appendix). The CRI compiler, of course, has the same structure but at an assembly level. The vectorized FTNX code appears to have another defect: because of the way calls to vector primitives are set up under FTNX, chain slots may be missed during execution. This is not the case in the CRI compiler. (The people at LASL have corrected the FTNX compiler, and it now catches chain slots.)

The CAL BLAS under the FTNX compiler provide the fastest execution speed. These BLAS are painstakingly and cleverly coded implementations. The CRI compiler VECT=ON comes very close to the execution speed of the CAL BLAS.

Timing of the routines was also carried out on a CDC 7600 using three different codings. The inline versions were compiled and run on the CDC 7600 as well as the versions involving calls to the BLAS. Two different implementations of the BLAS were used on the CDC 7600; one was the FORTRAN BLAS unrolled and the other was a COMPASS (CDC 7600 Assembly Language) version of the BLAS, which were tuned to run optimally on the CDC 7600. Table IV shows the ratios of execution time at order 100 for SGEFA, which was run in seven different environments on the CRAY-1 and three different environments on the CDC 7600.

The timings indicate that at order 100 the CDC 7600 FTN-compiled inline run is faster than the CRAY-1 FTNX-compiled inline run. Again, this is because the FTNX compiler stores and fetches the loop index every pass through the loop, while the CDC 7600 FTN compiler keeps the loop indexes in a register during loop execution.

TABLE IV

COMPARISON OF RELATIVE EXECUTION TIMES FOR SGEFA AT ORDER 100
ON THE CDC 7600 AND CRAY-1 USING THE FTN, FTNX, AND CRI COMPILERS

	FTNX Inline	7600 Inline	CRI Inline VECT=OFF	CDC 7600 FORTRAN BLAS	CRI FORTRAN BLAS	CDC 7600 COMPASS BLAS	FTNX FORTRAN BLAS	CRI Inline VECT=ON	FTNX Vector-ized Inline
CDC 7600 Inline	1.4								
CRI Inline VECT=OFF	1.7	1.3							
CDC 7600 FORTRAN BLAS	1.9	1.4	1.1						
CRI FORTRAN BLAS	2.0	1.5	1.2	1.1					
CDC 7600 Compass BLAS	2.6	1.9	1.5	1.4	1.3				
FTNX FORTRAN BLAS	2.7	2.0	1.6	1.5	1.3	1.0			
CRI Inline VECT=ON	5.9	4.2	3.3	3.0	2.8	2.2	2.1		
FTNX Vectorized Inline	5.9	4.3	3.5	3.2	2.9	2.3	2.2	1.0	
FTNX CAL BLAS	7.2	5.3	4.3	3.9	3.6	2.8	2.7	1.3	1.2

(Row entries are so many times faster than column entries.)

The CRAY-1 FTNX run with FORTRAN BLAS was 1.5 times faster than the CDC 7600 FTN run with FORTRAN BLAS. This is more typical of the results expected in comparing the CDC 7600 to the CRAY-1 run on scalar code.

The CDC 7600 COMPASS BLAS run about the same speed as the CRAY-1 FTNX FORTRAN BLAS run. The COMPASS BLAS on the CDC 7600 have been carefully coded so the loops are unrolled, and the instructions are scheduled so full overlap of arithmetic operations can occur. The results here are not surprising; the COMPASS BLAS can be thought of as running at "vector" speeds on the CDC 7600.

SUMMARY

This report describes what can be expected from the CRAY-1 for solving linear systems in a FORTRAN environment at the present time. There are things that can be done to the compilers to make the code generated more efficient. For example, in the FTNX compiler the loop index should be held in a register, and in the CRI compiler the performance could be improved by local and global optimization and instruction scheduling. This report is not intended to describe how the hardware performs. Readers who would like that information are referred to a report by T. Keller.[11] This report shows that the way in which an algorithm is implemented on the CRAY-1 affects the way it will perform.

After the timings were completed, an improvement was realized in the CAL BLA SAXPY that made the four codes that used the CAL BLAS run between 15-20% faster.

ACKNOWLEDGMENTS

I would like to thank Bill Buzbee, Thomas Jordan, and all the people at LASL who made my stay interesting and educational.

LINPACK ANECDOTE #1

One of the interesting things uncovered by the timing was a hardware problem in the LASL CRAY-1. During the timing runs, a check was made of the answers produced by the various routines. It was discovered that codes run in two different implementations produced the wrong answers. After some investigation by T. Jordan, the CRAY-1 engineers, and me, the problem was traced to a hardware board in the arithmetic unit that adjusts the exponents of operands before vector addition. The exponent adjustment was not being performed correctly in certain instances when operating in vector mode. Incorrect results were produced when the exponent to be adjusted had a certain bit pattern, making the errors in the answers appear somewhat mysterious. When it was finally tracked down and the defective board replaced, the correct results were obtained. The interesting thing is that the CRAY-1 had been in operation at LASL for a little over two months and no one seemed to notice any problems. The machine passed its diagnostic tests every morning and many hours of production work had been completed before the problem was uncovered.

REFERENCES

1. J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, "LINPACK Working Note #9, Preliminary LINPACK User's Guide," Argonne National Laboratory report ANL TM-313 (August 1977).

2. CRAY-1 Computer System Reference Manual, 2240004, Cray Research, Inc., (February 1977).

3. C. B. Moler, "Matrix Computation with Fortran and Paging," Comm. ACM 15 (April 1972), pp. 268-270.

4. J. J. Dongarra, "LINPACK Working Note #3, Fortran BLAS Timing," Argonne National Laboratory (December 1976).

5. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprogram for Use with Fortran," Submitted to Trans. Math. Software, July 1976.

6. J. M. Boyle and M. Matz, "Automating Multiple Program Relations," Proc. MRI Symposium XXIV; Computer Software Engineering, (Poly. Tech. Press, New York, 1977), pp. 421-457.

7. "LASL Guide to the CRAY-1 Computer, PIM-7, Los Alamos Scientific Laboratory report LA-5525-M, Vol. 7, (November 1977).

8. T. L. Jordan, private communications, Los Alamos Scientific Laboratory, (November 1977).

9. CRAY-1 Fortran Reference Manual, 2240009, Cray Research, Inc., (November 1977).

10. K. Fong and T. L. Jordan, "Some Linear Algebraic Algorithms and their Performance on CRAY-1," Los Alamos Scientific Laboratory report LA-6774 (June 1977).

11. T. W. Keller, "CRAY-1 Evaluation," Los Alamos Scientific Laboratory report LA-6456-MS (December 1976).

APPENDIX

SUBROUTINE SGEFA1 - EXAMPLE WITH CALLS TO THE BLAS

```
      SUBROUTINE SGEFA1(A,LDA,N,IPVT,INFO)
      INTEGER LDA,N,IPVT(1),INFO
      REAL A(LDA,1)
C
C     SGEFA1 FACTORS A REAL MATRIX BY GAUSSIAN ELIMINATION.
C
C     SGEFA1 IS USUALLY CALLED BY SGECO, BUT IT CAN BE CALLED
C     DIRECTLY WITH A SAVING IN TIME IF  RCOND  IS NOT NEEDED.
C     (TIME FOR SGECO) = (1 + 9/N)*(TIME FOR SGEFA1) .
C
C     ON ENTRY
C
C        A       REAL(LDA, N)
C                THE MATRIX TO BE FACTORED.
C
C        LDA     INTEGER
C                THE LEADING DIMENSION OF THE ARRAY  A .
C
C        N       INTEGER
C                THE ORDER OF THE MATRIX  A .
C
C     ON RETURN
C
C        A       AN UPPER TRIANGULAR MATRIX AND THE MULTIPLIERS
C                WHICH WERE USED TO OBTAIN IT.
C                THE FACTORIZATION CAN BE WRITTEN  A = L*U  WHERE
C                L  IS A PRODUCT OF PERMUTATION AND UNIT LOWER
C                TRIANGULAR MATRICES AND  U  IS UPPER TRIANGULAR.
C
C        IPVT    INTEGER(N)
C                AN INTEGER VECTOR OF PIVOT INDICES.
```

```
C        INFO    INTEGER
C                = 0  NORMAL VALUE.
C                = K  IF  U(K,K) .EQ. 0.0 .  THIS IS NOT AN ERROR
C                     CONDITION FOR THIS SUBROUTINE, BUT IT DOES
C                     INDICATE THAT SGESL1 OR SGEDI1 WILL DIVIDE BY ZERO

C                     IF CALLED.  USE  RCOND  IN SGECO FOR A RELIABLE
C                     INDICATION OF SINGULARITY.
C
C     LINPACK. THIS VERSION DATED 07/14/77 .
C     CLEVE MOLER, UNIVERSITY OF NEW MEXICO, ARGONNE NATIONAL LABS.
C
C     SUBROUTINES AND FUNCTIONS
C
C     BLAS SAXPY,SSCAL,ISAMAX
C
C     INTERNAL VARIABLES
C
      REAL T
      INTEGER ISAMAX,J,K,KP1,L,NM1
C
C
C     GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING
C
      INFO = 0
      NM1 = N - 1
      IF (NM1 .LT. 1) GO TO 70
      DO 60 K = 1, NM1
         KP1 = K + 1
C
C        FIND L = PIVOT INDEX
C
         L = ISAMAX(N-K+1,A(K,K),1) + K - 1
         IPVT(K) = L
C
C        ZERO PIVOT IMPLIES THIS COLUMN ALREADY TRIANGULARIZED
C
         IF (A(L,K) .EQ. 0.0E0) GO TO 40
C
C           INTERCHANGE IF NECESSARY
C
            IF (L .EQ. K) GO TO 10
               T = A(L,K)
               A(L,K) = A(K,K)
               A(K,K) = T
   10       CONTINUE
C
C           COMPUTE MULTIPLIERS
C
            T = -1.0E0/A(K,K)
            CALL SSCAL(N-K,T,A(K+1,K),1)
C
C           ROW ELIMINATION WITH COLUMN INDEXING
C
            DO 30 J = KP1, N
               T = A(L,J)
               IF (L .EQ. K) GO TO 20
                  A(L,J) = A(K,J)
                  A(K,J) = T
   20          CONTINUE
               CALL SAXPY(N-K,T,A(K+1,K),1,A(K+1,J),1)
   30       CONTINUE
         GO TO 50
   40    CONTINUE
            INFO = K
```

```
 50    CONTINUE
 60 CONTINUE
 70 CONTINUE
    IPVT(N) = N
    IF (A(N,N) .EQ. 0.0E0) INFO = N
    RETURN
    END
```

SUBROUTINE SGEFA2 - EXAMPLE WITH BLAS REPLACED WITH INLINE CODE

```
      SUBROUTINE SGEFA2(A,LDA,N,IPVT,INFO)
      INTEGER LDA,N,IPVT(1),INFO
      REAL A(LDA,1)
C
C     SGEFA2 FACTORS A REAL MATRIX BY GAUSSIAN ELIMINATION.
C
C     SGEFA2 IS USUALLY CALLED BY SGECO, BUT IT CAN BE CALLED
C     DIRECTLY WITH A SAVING IN TIME IF  RCOND  IS NOT NEEDED.
C     (TIME FOR SGECO) = (1 + 9/N)*(TIME FOR SGEFA2) .
C
C     ON ENTRY
C
C        A       REAL(LDA, N)
C                THE MATRIX TO BE FACTORED.
C
C        LDA     INTEGER
C                THE LEADING DIMENSION OF THE ARRAY  A .
C
C        N       INTEGER
C                THE ORDER OF THE MATRIX  A .
C
C     ON RETURN
C
C        A       AN UPPER TRIANGULAR MATRIX AND THE MULTIPLIERS
C                WHICH WERE USED TO OBTAIN IT.
C                THE FACTORIZATION CAN BE WRITTEN  A = L*U  WHERE
C                L  IS A PRODUCT OF PERMUTATION AND UNIT LOWER
C                TRIANGULAR MATRICES AND  U  IS UPPER TRIANGULAR.
C
C        IPVT    INTEGER(N)
C                AN INTEGER VECTOR OF PIVOT INDICES.
C
C        INFO    INTEGER
C                = 0  NORMAL VALUE.
C                = K  IF  U(K,K) .EQ. 0.0 .  THIS IS NOT AN ERROR
C                     CONDITION FOR THIS SUBROUTINE, BUT IT DOES
C                     INDICATE THAT SGESL2 OR SGEDI2 WILL DIVIDE BY ZERO

C                     IF CALLED.  USE  RCOND  IN SGECO FOR A RELIABLE
C                     INDICATION OF SINGULARITY.
C
C     LINPACK. THIS VERSION DATED 07/14/77 .
C     CLEVE MOLER, UNIVERSITY OF NEW MEXICO, ARGONNE NATIONAL LABS.
C
C     SUBROUTINES AND FUNCTIONS
C
C     BLAS SAXPY,SSCAL,ISAMAX
C
C     INTERNAL VARIABLES
C
      REAL T
      INTEGER ISAMAX,J,K,KP1,L,NM1
C
C
C     GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING
C
      INFO = 0
      NM1 = N - 1
      IF (NM1 .LT. 1) GO TO 130
      DO 120 K = 1, NM1
         KP1 = K + 1
         ISAMAX = K
         IF (N .LT. K) GO TO 20
         DO 10 IAMAX = K, N
            IF (ABS(A(IAMAX,K)) .GT. ABS(A(ISAMAX,K))) ISAMAX = IAMAX
   10    CONTINUE
   20    CONTINUE
C
C        FIND L = PIVOT INDEX
C
         L = ISAMAX
         IPVT(K) = L
C
C        ZERO PIVOT IMPLIES THIS COLUMN ALREADY TRIANGULARIZED
C
         IF (A(L,K) .EQ. 0.0E0) GO TO 100
C
C           INTERCHANGE IF NECESSARY
C
            IF (L .EQ. K) GO TO 30
               T = A(L,K)
               A(L,K) = A(K,K)
               A(K,K) = T
   30       CONTINUE
C
C           COMPUTE MULTIPLIERS
C
            T = -1.0E0/A(K,K)
            KSCAL = K + 1
            IF (N .LT. KSCAL) GO TO 50
            DO 40 ISCAL = KSCAL, N
               A(ISCAL,K) = T*A(ISCAL,K)
   40       CONTINUE
   50       CONTINUE
C
C           ROW ELIMINATION WITH COLUMN INDEXING
C
            DO 90 J = KP1, N
               T = A(L,J)
               IF (L .EQ. K) GO TO 60
                  A(L,J) = A(K,J)
                  A(K,J) = T
   60          CONTINUE
               KAXPY = K + 1
               IF (N .LT. KAXPY) GO TO 80
               DO 70 IAXPY = KAXPY, N
                  A(IAXPY,J) = A(IAXPY,J) + T*A(IAXPY,K)
   70          CONTINUE
   80          CONTINUE
   90       CONTINUE
         GO TO 110
  100    CONTINUE
            INFO = K
  110    CONTINUE
  120 CONTINUE
  130 CONTINUE
      IPVT(N) = N
      IF (A(N,N) .EQ. 0.0E0) INFO = N
      RETURN
      END
```

SUBROUTINE SGEFA3 - EXAMPLE WITH INLINE CODE THAT HAS BEEN PASSED THROUGH THE VECTORIZER

```
      SUBROUTINE SGEFA3(A,LDA,N,IPVT,INFO)
      INTEGER LDA,N,IPVT(1),INFO,ISAMAX,J,K,KP1,L,NM1
      REAL A(LDA,1),T
      INTEGER NITER,OUTRXS,OUTRMX,OUTRWD,ISTRIP,OUTRNX
C
C     SGEFA3 FACTORS A REAL MATRIX BY GAUSSIAN ELIMINATION.
C
C     SGEFA3 IS USUALLY CALLED BY SGECO, BUT IT CAN BE CALLED
C     DIRECTLY WITH A SAVING IN TIME IF  RCOND  IS NOT NEEDED.
C     (TIME FOR SGECO) = (1 + 9/N)*(TIME FOR SGEFA3) .
C
C     ON ENTRY
C
C        A       REAL(LDA, N)
C                THE MATRIX TO BE FACTORED.
C
C        LDA     INTEGER
C                THE LEADING DIMENSION OF THE ARRAY  A .
C
C        N       INTEGER
C                THE ORDER OF THE MATRIX  A .
C
C     ON RETURN
C
C        A       AN UPPER TRIANGULAR MATRIX AND THE MULTIPLIERS
C                WHICH WERE USED TO OBTAIN IT.
C                THE FACTORIZATION CAN BE WRITTEN  A = L*U  WHERE
C                L  IS A PRODUCT OF PERMUTATION AND UNIT LOWER
C                TRIANGULAR MATRICES AND  U  IS UPPER TRIANGULAR.
C
C        IPVT    INTEGER(N)
C                AN INTEGER VECTOR OF PIVOT INDICES.
C
C        INFO    INTEGER
C                = 0  NORMAL VALUE.
C                = K  IF  U(K,K) .EQ. 0.0 .  THIS IS NOT AN ERROR
C                     CONDITION FOR THIS SUBROUTINE, BUT IT DOES
C                     INDICATE THAT SGESL3 OR SGEDI3 WILL DIVIDE BY ZERO

C                     IF CALLED.  USE  RCOND  IN SGECO FOR A RELIABLE
C                     INDICATION OF SINGULARITY.
C
C     LINPACK. THIS VERSION DATED 07/14/77 .
C     CLEVE MOLER, UNIVERSITY OF NEW MEXICO, ARGONNE NATIONAL LABS.
C
C     SUBROUTINES AND FUNCTIONS
C
C     BLAS SAXPY,SSCAL,ISAMAX
C
C     INTERNAL VARIABLES
C
```

```
C
C
C     GAUSSIAN ELIMINATION WITH PARTIAL PIVOTING
C
      INFO=0
      NM1=N-1
      IF(NM1.LT.1) GO TO 130
      DO 120 K=1,NM1
      KP1=K+1
      ISAMAX=K
      IF(N.LT.K) GO TO 20
      DO 10 IAMAX=K,N
      IF(ABS(A(IAMAX,K)).GT.ABS(A(ISAMAX,K))) ISAMAX=IAMAX
      CONTINUE
   10 CONTINUE
   20 CONTINUE
C
C        FIND L = PIVOT INDEX
C
      L=ISAMAX
      IPVT(K)=L
C
C        ZERO PIVOT IMPLIES THIS COLUMN ALREADY TRIANGULARIZED
C
      IF(A(L,K).EQ.0.0) GO TO 100
C
C           INTERCHANGE IF NECESSARY
C
      IF(L.EQ.K) GO TO 30
      T=A(L,K)
      A(L,K)=A(K,K)
      A(K,K)=T
   30 CONTINUE
C
C           COMPUTE MULTIPLIERS
C
      T=-1.0/A(K,K)
      KSCAL=K+1
      IF(N.LT.KSCAL) GO TO 50
      NITER=N+1-KSCAL
      CALL LOPCXXX(NITER,OUTRXS,OUTRMX)
      CALL SETLX(OUTRXS)
      OUTRWD=OUTRXS
      ISTRIP=KSCAL
      DO 90000 OUTRNX=1,OUTRMX
      CALL ILDVX00(A(ISTRIP,K))
      CALL SRPVX01(T)
      CALL ISTV10X(0,0,A(ISTRIP,K))
      ISTRIP=OUTRWD+ISTRIP
      OUTRWD=64
      CALL SETLX(64)
90000 CONTINUE
   40 CONTINUE
   50 CONTINUE
C
C           ROW ELIMINATION WITH COLUMN INDEXING
C
      DO 90 J=KP1,N
      T=A(L,J)
      IF(L.EQ.K) GO TO 60
      A(L,J)=A(K,J)
      A(K,J)=T
   60 CONTINUE
```

```
      KAXPY=K+1
      IF(N.LT.KAXPY) GO TO 80
      NITER=N+1-KAXPY
      CALL LOPCXXX(NITER,OUTRXS,OUTRMX)
      CALL SETLX(OUTRXS)
      OUTRWD=OUTRXS
      ISTRIP=KAXPY
      DO 90001 OUTRNX=1,OUTRMX
      CALL ILDVX00(A(ISTRIP,K))
      CALL SRPVX01(T)
      CALL ILDVX02(A(ISTRIP,J))
      CALL VRAV213
      CALL ISTV30X(0,0,A(ISTRIP,J))
      ISTRIP=OUTRWD+ISTRIP
      OUTRWD=64
      CALL SETLX(64)
90001 CONTINUE
   70 CONTINUE
   80 CONTINUE
   90 CONTINUE
      GO TO 110
  100 CONTINUE
      INFO=K
  110 CONTINUE
  120 CONTINUE
  130 CONTINUE
      IPVT(N)=N
      IF(A(N,N).EQ.0.0) INFO=N
      RETURN
      END
```

SUBROUTINE SAXPY - EXAMPLE OF A BLA WITH UNROLLED LOOP

```
      SUBROUTINE SAXPY(N,SA,SX,INCX,SY,INCY)
C
C     CONSTANT TIMES A VECTOR PLUS A VECTOR.
C     USES UNROLLED LOOP FOR INCREMENTS EQUAL TO ONE.
C     JACK DONGARRA, LINPACK, 6/17/77.
C
      REAL SX(1),SY(1),SA
      INTEGER I,INCX,INCY,IX,IY,M,MP1,N
C
      IF(N.LE.0)RETURN
      IF (SA .EQ. 0.0) RETURN
      IF(INCX.EQ.1.AND.INCY.EQ.1)GOTO 20
C
C        CODE FOR UNEQUAL INCREMENTS OR EQUAL INCREMENTS
C          NOT EQUAL TO 1
C
      IX = 1
      IY = 1
      IF(INCX.LT.0)IX = (-N+1)*INCX + 1
      IF(INCY.LT.0)IY = (-N+1)*INCY + 1
      DO 10 I = 1,N
        SY(IY) = SY(IY) + SA*SX(IX)
        IX = IX + INCX
        IY = IY + INCY
   10 CONTINUE
      RETURN
```

```
C
C        CODE FOR BOTH INCREMENTS EQUAL TO 1
C
C
C        CLEAN-UP LOOP
C
   20 M = MOD(N,4)
      IF( M .EQ. 0 ) GO TO 40
      DO 30 I = 1,M
        SY(I) = SY(I) + SA*SX(I)
   30 CONTINUE
      IF( N .LT. 4 ) RETURN
   40 MP1 = M + 1
      DO 50 I = MP1,N,4
        SY(I) = SY(I) + SA*SX(I)
        SY(I + 1) = SY(I + 1) + SA*SX(I + 1)
        SY(I + 2) = SY(I + 2) + SA*SX(I + 2)
        SY(I + 3) = SY(I + 3) + SA*SX(I + 3)
   50 CONTINUE
      RETURN
      END
```

STAR-100: GOOD NEWS AND BAD NEWS

by

T. Rudy
Lawrence Livermore Laboratory
Livermore, California 94550

ABSTRACT

The optimal utilization of the CDC STAR-100 requires the programmer to investigate several alternatives in code restructuring. This paper presents performance measurements for various programs and describes the trade-offs programmers must address in using a vector machine.

HOW TO GET MORE OUT OF YOUR VECTOR PROCESSOR

by

Brian Q. Brode
Massachusetts Computer Associates, Inc.

ABSTRACT

This paper presents suggestions to aid in "restructuring" a FORTRAN program so that it more efficiently uses a vector machine. Code conversion and restructuring techniques are discussed; in the final section, working examples and timings are presented.

I. INTRODUCTION

To get more out of your Vector Processor, as much of your code as possible must execute in vector mode. This paper presents a "bag of tricks" to aid in "restructuring" a FORTRAN program so that it more efficiently uses a vector machine. In the following sections the entire process of converting a code is briefly examined, beginning with the code as a whole and telescoping down to the statement level. Special attention is given to the inherent overhead introduced by some restructuring techniques. The discussion is kept as machine-independent as possible, with only a knowledge of FORTRAN assumed. No explicit vector syntax is used; but the conversion to any vector syntax from our restructured FORTRAN should be very straightforward for human or machine. In the final section, working examples and timings are presented.

II. FIRST PASS: CODE OVERVIEW

When faced with a program to speed up, the first step is to determine where all the time is spent. Restructuring can be a time-consuming, error-prone process and it would be foolish to waste effort on insignificant parts before hitting the main chunks of the program. Frequently a very small percentage of the code can eat up a huge portion of the CPU time, and these are the places where conversion should begin. An idea of which sections are greediest about CPU time can usually be gathered just by eyeballing the program, but actual timings of each segment of the code are a much better starting point. Some installations have sophisticated programs that print out histograms of CPU usage; these tools prove to be very valuable when restructuring a program.

We should next check to see if the particular techniques being used to solve the problems in the program can be converted to a more optimal method for a vector processor. If the technique is very inefficient for use on a vector machine, one should investigate an entirely different (more array-processing oriented) algorithm. Vector versions for such commonly used techniques as fast fourier transforms already exist. Many times, better results and less effort come from a total rewrite of a particular algorithm with a vector processor in mind as the target machine.

Finally, we need to have some knowledge about the normal values of the important flow-changing variables. It is important to know the lengths of DO loops, common values of switches, normal density of arrays, and other calculation dependent features if a good restructured version is to be obtained. Another important piece of knowledge is the amount of memory available that can be safely used for vector temporaries.

III. GENERAL RESTRUCTURING COMMENTS: THINGS TO WATCH FOR

On machines with no built-in "preferred" vector length (STAR, 7600, 176), the longer the length of your vectors the better your performance will be. The overhead involved in setting up vector operations can be large and is diluted significantly only for relatively long vector lengths. Long vectors are imperative for good timing results. Specific suggestions on increasing vector length appear in the next section (IV.12), and timings demonstrating its import are in the last section. On machines with a hardware-influenced "natural" vector length (CRAY, ILLIAC), the above discussion applies up to the built-in length and cycles modulo(built-in length).

Restructuring almost always involves the introduction of temporary arrays to hold intermediate results that were previously held in scalar locations. If care is not exercised to conserve temporaries, excessive memory will be gobbled up. Reusing temporary arrays from one loop to another, and placing them in a common block specially for reuse in other routines can aid in reducing memory requirements. (On some machines, temporary arrays should also be allocated in such a way as to minimize memory conflicts.)

Frequently overhead must be introduced when restructuring some sequences of code. Care must be taken so that this overhead doesn't cancel any speedup offered by using a vector processor. An example of such a problem arises when one is faced with a DO loop with several loop-dependent branches. One technique for optimizing such a condition is to calculate each path for the entire range of the DO loop and merge the results accordingly. Such a technique may introduce so much extra calculation that the vector code may execute slower than the original.

IV. SPECIFIC TECHNIQUES: A BAG OF TRICKS

What follows is an attempt to catalog most known restructuring techniques; some sections may be less interesting than others for the experienced restructurer. The emphasis in this section is on examples rather than lengthy explanation.

1.   Change IF loops to DO loops where possible.
     A simple example:
            Original: (IF loop)

```
                N=1
         100    CONTINUE                          (SCALAR)
                A(N)=B(N)
                N=N+1
                IF (N.LE.10) GO TO 100
```

            Restructured: (DO loop)

```
                DO 1 N=1,10                       (VECTOR)
                A(N)=B(N)
            1   CONTINUE
```

More complex is the case where the IF statement depends on a variable set in the IF loop. If a lot of calculation is contained in the IF loop and we have reason to believe that most of the array would run through the loop before the IF statement kicks us out, we can restructure as in the following example. Of course, we introduce overhead (unused calculations) by doing this.
            Original:

```
                N=1
         100    CONTINUE
                A(N)=B(N)**2+.5*C(N)*D(N)/E(N)    (SCALAR)
                N=N+1
                IF (A(N).GT.0) GO TO 100
```

            Restructured:

```
                DO 1 N=1,ADIM        (ADIM: Dimension of A)
       C      1 TEMP(N)=B(N)**2+.5*C(N)*D(N)/E(N) (VECTOR)
              2 CONTINUE
                A(N)=TEMP(N)
                N=N+1                             (SCALAR)
                IF (A(N).GT.0) GO TO 2
```

2.   Pull functions and subroutines into loops they are called from. This maneuver should actually cut down on overhead as it will eliminate the subroutine linkage time. If the routines cannot be pulled into the calling loops (because there are too many occurences of calls to the routine to make code repetition feasible, or because of disruption to the logical structure of the code) then perhaps the loop can be pushed into the routine. Regardless, a vector loop must be contiguous in its flow of control so if the subroutine cannot be altered, its call must be isolated from the rest of the loop (see IV.4).
            Original:

```
                DO 100 I=1,100
         100    CALL SUB(A(I),B(I))               (SCALAR)
                  ⋮
                END
                SUBROUTINE SUB(X,Y)
                X=Y/2.+X
                RETURN
                END
```

```
          Restructured:

                DO 100 I=1,100                    (VECTOR)
           100  A(I)=B(I)/2.+A(I)
```

3. Eliminate temporary scalars by introducing vector temporaries. If the machine does not have vector registers, this step adds some overhead as scalar temps usually are kept in fast scalar registers, while the vector elements must be fetched from memory. If the "temporary" scalar is used outside the loop, the last value of the corresponding vector temporary must be assigned to it after the end of the loop. (If a machine has vector registers and a smart compiler or pre-compiler that can propagate scalar temporaries to vector registers, then the reverse process of changing vector temporary arrays (in memory) into temporary scalars should be employed to save needless vector fetches and stores.)

Example 1:

```
          Original:
                DO 100 I=1,100
                SCA1=A(I)*B(I)+C(I)
           100  C(I)=E(I)/SCA1+SCA1

          Restructured:
                DO 100 I=1,100
                TEMP(I)=A(I)*B(I)+C(I)
           100  D(I)=E(I)/TEMP(I)+TEMP(I)
```

Example 2: (A(I) is a "Pseudo Scalar")

```
          Original:
                DO 100 J=1,100                    (SCALAR)
                A(I)=B(I,J)*C(J)+F(J)
           100  D(I,J)=E(I,J)/A(I)+A(I)

          Restructured:
                DO 200 J=1,100                    (VECTOR)
                TEMP(J)=B(I,J)*C(J)+K(J)
           200  D(I,J)=E(I,J)/TEMP(J)+TEMP(J)
                A(I)=TEMP(100)
           100  CONTINUE
```

4. Split out non-vectorizable code into separate loops. When an inherently un-vectorizable element exists in the body of a loop, it must be broken out into a loop of its own so that the rest of the original loop can be converted to vector code. Examples of things that need isolation are intrinsic and basic external functions that have no vector counterpart, programmer routines that cannot be moved inside the loop (see IV.2), input and output statements, recursive statements (see IV.8), reduction functions (see IV.9) and non-subscript references to the loop index (see IV.10).

Example:

```
          Original:
                DO 100 I=1,101                    (SCALAR)
                A(I)=B(I)*C(I)
                D(I)=ASINH(A(I)**2+2.0)      (Assuming ASINH is not a
           100  E(I)=D(I)/B(I)+A(I)           convertible function)
          Restructured:
                 DO 200 I=1,101                     (VECTOR)
                 A(I)=B(I)*C(I)
           200   TEMP(I)=A(I)**2+2.0
          C
                 DO 300 I=1,101                     (SCALAR)
           300   D(I)=ASINH(TEMP(I))
          C
                 DO 400 I=1,101                     (VECTOR)
           400   E(I)=D(I)/B(I)+A(I)
```

5.   If the loop contains conditional assignment or conditional GOTO statements which are independent of the loop index, the loop must be split up so that the loop-independent decisions are not made inside a loop.
Example:

```
          Original:
                 DO 100 J=1,100
                 A(I)=0.0
                 IF (NSWITCH.EQ.0) A(I)=1.0
                 B(I)=A(I)+.5*C(I)
                 IF (NDONE.EQ.1) GO TO 100
                 D(I)=A(I)+B(I)**2
           100   CONTINUE

          Restructured:
                 IF (NSWITCH.EQ.0) GO TO 110
          C
                 DO 100 J=1,100
           100   A(I)=0.0
                 GO TO 130
          C
           110   DO 120 I=1,100
           120   A(I)=1.
           130   CONTINUE
          C
                 DO 140 I=1,100
           140   B(I)=A(I)+.5*C(I)
                 IF (NDONE.EQ.1) GO TO 200
          C
                 DO 150 I=1,100
           150   D(I)=A(I)+B(I)**2
           200   CONTINUE
```

9.   Loop-dependent conditional assignments can be handled with vector merge routines. If such routines are not available, the offending statement must be isolated in its own loop so that the rest of the loop is not dragged down into scalar mode with it.
Example:

```
          Original:
                 DO 100 I=1,100
                 A(I)=B(I)+C(I)
                 IF (A(I).EQ.0) D(I)=E(I)+F(I)      (SCALAR)
           100   G(I)=A(I)+D(I)
          Restructured:            (no MERGE)
                 DO 100 I=1,100
           100   A(I)=B(I)+C(I)                     (VECTOR)
                 DO 110 I=1,100
           110   IF (A(I).EQ.0) D(I)=E(I)+F(I)      (SCALAR)
                 DO 120 I=1,100
           120   G(I)=A(I)+D(I)                     (VECTOR)
          Restructured:            (with CRAY CFT style MERGE function)
                 DO 100 I=1,100
                 A(I)=B(I)+C(I)                     (VECTOR)
                 D(I)=CMVGZ(E(I)+F(I),D(I),A(I))
           100   G(I)=A(I)+D(I)
```

NOTE: CVMGZ is Conditional Vector Merge Zero. CVMGZ(true vector, false vector, test vector) merge or controlled-store type functions exist for the 7600, STAR, and other machines having a hardware capability of this kind.

A frequently used application of loop-dependent conditional statements is to make sure that division by zero does not occur. Handling of these singularities requires special restructuring to avoid a zero divide or other floating-point error.

```
Example:   Singularity Handling
          Original:
                 DO 100 I=1,100
           100   IF (C(I).NE.0.0) A(I)=B(I)/C(I)

          Restructured:
                 DO 100 I=1,100
                 CT(I)=1.0
                 TEMP(I)=A(I)
                 IF (C(I).NE.0.0) CT(I)=C(I)        (MERGE)
           100   IF (C(I).NE.0.0) A(I)=B(I)/CT(I)  (MERGE)
```

7. The loop dependent conditional forward transfer. This is a very interesting case, since one must always introduce some overhead in the restructuring process. There are three options available; 1) doing all pathways over the whole vector length and merging the results (overhead: those computations which were not performed in scalar mode but are in vector mode), 2) pre-packing the arrays for each path and doing the computations for only those elements of the original array who need it (overhead: packing and unpacking logic, reduced vector length), and 3) leaving part of the loop in scalar mode. Which method to select depends on the number of paths, frequency of execution of each particular path and the amount of calculation done in each path. In the simple case of a transfer around a few statements, if the transfer does not occur very often (i.e., most of the array elements get set), vector merging is warranted. (Not too many unused calculations.) If the transfer occurs more often (or there are several different paths), pre-packing the arrays may be an alternative strategy. This method should be used only with very long vectors, as the packing/unpacking overhead can be considerable. If a path is rarely executed, it should be left scalar. Frequently, it is difficult to glean information about how many times a certain pathway is run through by looking at the code or the data; in these cases it is advisable to put counters in the code, run several test cases, and print out exactly what the program is doing.

Example 1 (Merging):

Original:
```
       DO 100 I=1,100
       A(I)=B(I)*C(I)
       IF (A(I).EQ.0) GO TO 110
       A(I)=D(I)*E(I)
       B(I)=1.
 110   CONTINUE
       F(I)=A(I)+B(I)
 100   CONTINUE
```

Restructured: (with MERGE function)
```
       DO 100 I=1,100
       A(I)=B(I)*C(I)
       B(I)=CVMGZ(1.,B(I),A(I))
       A(I)=CVMGZ(D(I)*E(I),A(I),A(I))
       F(I)=A(I)+B(I)
 100   CONTINUE
```

Example 2 (Pre-packing):

Original:
```
       DO 100 I=1,1000                         (SCALAR)
       A(I)=B(I)*C(I)
       IF (A(I).EQ.0) GO TO 110
       IF (A(I).LT.0) GO TO 120
       R(I)=SQRT(A(I)**2+W(I))*.05+D(I)
       S(I)=A(I)*D(I)
       GO TO 100
 110   CONTINUE
       R(I)=ALOG(B(I)**2+W(I))*.05+E(I)
       S(I)=B(I)*E(I)
       GO TO 100
 120   CONTINUE
       R(I)=EXP(C(I)**2+W(I))*.05+F(I)
       S(I)=C(I)*F(I)
 100   CONTINUE
```

Restructured:
```
 IGT=0
 IEQ=0
 ILT=0
       DO 100 I=1,1000                         (VECTOR)
 100   A(I)=B(I)*C(I)
C
C PACK
C
       DO 110 I=1,1000                         (SCALAR)
       IF (A(I).EQ.0) GO TO 120
       IF (A(I).LT.0) GO TO 130
       IGT=IGT+1
       TGTA(IGT)=A(I)
       TGTW(IGT)=W(I)
       TGTD(IGT)=D(I)
       GO TO 110
 120   IEQ=IEQ+1
       TEQB(IEQ)=B(I)
       TEQW(IEQ)=W(I)
       TEQE(IEQ)=E(I)
       GO TO 110
 130   ILT=ILT+1
       TLTC(ILT)=C(I)
       TLTW(ILT)=W(I)
       TLTF(ILT)=F(I)
 110   CONTINUE
C
C CALCULATE
C
       DO 200 I=1,IGT                     (VECTOR)
       TGTR(I)=SQRT(TGTA(I)**2+TGTW(I))*.05+TGTD(I)
 200   TGTS(I)=TGTA(I)*TGTD(I)
       DO 300 I=1,IEQ                     (VECTOR)
       TEQR(I)=ALOG(TEQB(I)**2+TEQW(I))*.05+TEQE(I)
 300   TEQS(I)=TEQB(I)*TEQE(I)
       DO 400 I=1,ILT                     (VECTOR)
       TLTR(I)=EXP(TLTC(I)**2+TLTW(I))*.05+TLTF(I)
 400   TLTS(I)=TLTC(I)*TLTF(I)
C
C UNPACK
C
       IGT=IEQ=ILT=0
       DO 500 I=1,1000                    (SCALAR)
       IF (A(I).EQ.0) GO TO 510
       IF (A(I).LT.0) GO TO 520
       IGT=IGT+1
       R(I)=TGTR(IGT)
       S(I)=TGTS(IGT)
       GO TO 500
 510   IEQ=IEQ+1
       R(I)=TEQR(IEQ)
       S(I)=TEQS(IEQ)
       GO TO 500
 520   ILT=ILT+1
       R(I)=TLTR(ILT)
       S(I)=TLTS(ILT)
 500   CONTINUE
```

There are other methods of handling conditional statements as well. Some conditionals can be replaced by minimum and maximum functions, and sometimes the use of a temporary array containing ones and zeros as multiplicative factors can do the job.

Example: Elimination of Conditional with min/max

```
       DO 100 I=1,100
 100   IF (B(I).GT.0.0) B(I)=A(I)+B(J)
```

Restructured:

```
       DO 100 I=1,100
 100   B(I)=A(I)+AMAX1 (B(I),0.)
```

The next four items (recursiveness, reduction functions, non-subscript loop index uses, and indirect addressing) have in common that they usually must be split out of the main loop and either be left scalar or handled with special array utility subroutine calls. (Some reduction functions are recognized by sophisticated vectorizing compilers and pre-compilers.) The first three can all use the trick of switching the order of double nested loops so that the inner loop becomes convertible.

8. Recursiveness. When array element assignments use a value set on a previous pass through the loop, we have a recursive relationship, which cannot be vectorized because successive loop passes are not independent. Here there is an additional option to those mentioned above: find a closed form expression (if a simple one exists) that is non-recursive and does the same calculation.

```
Example 1:  Split Out
            Original:
                  DO 100 I=2,100                       (SCALAR)
                  A(I)=B(I)*C(I)
             100  D(I)=D(I-1)*A(I)+C(I)

            Restructured:  Isolated
                  DO 100 I=2,100                       (VECTOR)
             100  A(I)=B(I)*C(I)
                  DO 110 I=2,100                       (SCALAR)
             110  D(I)=D(I-1)*A(I)+C(I)

            Restructured:  Special Call (optimal assembly-language subroutine
                  DO 100 I=2,100                                 for 7600)
             100  A(I)=B(I)*C(I)
                  INDEX=. . . . .
                  CALL RVTVP(INDEX,D,A,C)

Example 2:  Switch Loops

            Original:
                  DO 100 I=1,100
                  DO 100 J=2,100                       (SCALAR)
             100  A(I,J)=A(I,J-1)*B(I,J)

            Restructured:      (Inner Loop is now not recursive)
                  DO 100 J=2,100
                  DO 100 I=1,100                       (VECTOR)
             100  A(I,J)=A(I,J-1)*B(I,J)
```

9. Summations and other reduction functions. This category includes totals (sum of array elements), Dot products (a.k.a. inner products, scalar products), sum of the squares of the elements of a vector, minimum and maximum elements of a vector, and any other function that maps a vector argument into a resultant scalar.

```
Example 1:  Summation
            Original:
                  DO 100 J=1,100                       (SCALAR)
             100  S=S+A(I)*B(I)/C(I)+.05*D(I)

            Restructured:
                  DO 100 I=1,100                       (VECTOR)
             100  TEMP(I)=A(I)*B(I)/C(I)+.05*D(I)
                  S=VTOTAL(TEMP(I),1)              (or equivalent loop)
```

```
Example 2:  Min and Max of Vector
            Original:
                   DO 100 I=1,100                    (SCALAR)
                   A(I)=C(I)**2+.05*F(I)
                   B(I)=E(I)**2-D(I)
                   ASCA=AMIN1(A(I),ASCA)
                   IF (BSCA.LT.B(I))BSCA=B(I)
             100   CONTINUE

            Restructured:
                   DO 100 I=1,100                    (VECTOR)
                   A(I)=C(I)**2+.05*F(I)
             100   B(I)=E(I)**2-D(I)
                   ASCA=AMIN1(ASCA,VMIN(A,100,1))    (or loop)
                   BSCA=AMAX1(BSCA,VMAX(B,100,1))

Example 3:  Switch Loops
            Original:
                   DO 100 I=1,100
                   DO 100 J=1,100
             100   B(I)=B(I)+A(I,J)        (A(I,J) is summed across J)

            Restructured:
                   DO 100 J=1,100
                   DO 100 I=1,100
             100   B(I)=B(I)+A(I,J)    (Vector B = Vector B + Vector A(1,J))
```

10. Non-subscript use of loop index. The loop index may be replaced by a real temporary array containing the appropriate sequential values. This array can be set up by a data statement (if the DO loop length is known before execution) or by a special call, or in an isolated DO loop.

```
Example:  Data Statement
            Original:
                   DO 100 I=1,100                    (SCALAR)
             100   A(I)=COS(D(I)*I)

            Restructured:
                   DATA TEMP/1.0,2.0,3.0,4.0,5.0,6.0,7.0,. . . 100.0/
                   DO 100 I=1,100                    (VECTOR)
             100   A(I)=COS(D(I)*TEMP(I))

Example:  Loop Switch
                   DO 100 I=1,100                    (SCALAR)
                   DO 100 J=1,100
             100   A(I,J)=COS(D(I)*J)

            Restructured:
                   DO 100 J=1,100
                   SCA=FLOAT(-J)
                   DO 100 I=1,100                    (VECTOR)
             100   A(I,J)=COS(D(I)*SCA)
```

11. Indirect addressing. Arrays that are addressed via an integer array subscript must be "scrambled" (either by a small DO loop or special subroutine calls) into a temporary vector whose elements are accessed sequentially in the DO loop. This procedure does involve some overhead (all the extra stores and fetches of the temporary), but not as much as the merge problem. If indirect addressing occurs on the left side of the assignment statement, the situation

is more complex. If the variable occurs only on the left side of the equal sign, then we can use an "un-scramble" process and there is no problem. If the variable occurs on both sides, then the unscramble should not be performed as incorrect results are possible if the integer array has any duplicate elements.
Example:

```
          Original:
                 DO 100 I=1,100
           100   A(IA(I))=B(IB(I))*C(I)+D(ID(I))**2-.5

          Restructured:
                 DO 100 I=1,100         (or use array utility calls)
                 TEMPB(I)=B(IB(I))      (VSL: SCRMBL (INDEX,B,IB,TEMPB))
           100   TEMPD(I)=D(ID(I))
          C
                 DO 200 I=1,100                     (VECTOR)
           200   TEMPA(J)=TEMPB(I)*C(I)+TEMPD(J)**2-.5
          C
                 DO 300 I=1,100         (or use array utility calls)
           300   A(IA(I))=TEMPA(I)      (VSL: USCRMBL (INDEX,TEMPA,IA,A))
```

12. Maximize DO loop length. In a multiply-nested situation, make sure the longest loop is the inner one. Sometimes double-nested loops can be combined into one super-long loop and this is even better.

```
Example 1:  Two loops to one long loop
            Original:
                   DO 50 J=1,JN
                   DO 50 I=1,IN
             50    A(I,J)=B(I,J)*C(I,J)

            Restructured:
              (    REAL T1(1),T2(1),T3(1)      )
              (    EQUIVALENCE (T1(1),A(1,1)),(T2(1),B(1,1)),(T3(1),C(1,1))  )

                   INJN=IN*JN
                   DO 50 I=1,INJN
             50    T1(I)=T2(I)*T3(I)

Example 2:  More complex case
            Original:
                   DO 100 I=1,IM
                   DO 100 J=1,JM
                   K=IM*(J-1)+I
                   Z(K)=A(K)*B(K)+C(K)
             100   W(K)=Z(K)*V(I)*Q(J)

            Restructured:
                   DO 100 I=1,IM
                   DO 100 J=1,JM
                   K=IM*(J-1)+I
             100   TEMP(K)=V(I)*Q(J)
            C
                   KM=IM*JM
            C
                   DO 200 K=1,KM
                   Z(K)=A(K)*B(K)+C(K)
             200   W(K)=Z(K)*TEMP(K)
```

13. Environment Dependent Consideration. Although I have ignored them so far, the conditions under which the code will be run will affect the restructuring process. If vectorizing compilers or pre-compilers are to be used, their quirks and capabilities must be taken into account. If hand coding in a vector syntax, machine particulars need to be understood (CRAY: chaining, vector register utilization, strip-mining, 7600: memory conflicts, LCM, etc.)

One very powerful restructuring tool is the Vectorizer (TM) (marketed by Massachusetts Computer Associates), a pre-compiler which detects array operations in FORTRAN DO loops. It is able to do the tricks described in Sections IV.3,4,5,6,9, and some of IV.10 and IV.12, as well as being able to output several different explicit vector syntaxes. It also gives informative diagnostics that aid in the restructuring process. Someone using the Vectorizer will have less hand restructuring to do and will have an easier time doing the restructuring that remains.

V. EXAMPLES AND TIMINGS: THE REAL WORLD

The following example demonstrates several of the techniques put forward in Section IV.

```
                  ⋮
            DO 120 J=1,NZ
      120   AD(J)=D(J,NZ,JET)
                  ⋮
            END
            REAL FUNCTION D(K,N,M)
            COMMON X(66)
            D=1.0
            Q=X(K)
            DO 3 L=1,M
            DO 2 J=L,N,M
            IF (J-K) 1,2,1
          1 D=2.0*D*(Q-X(J))
          2 CONTINUE
          3 CONTINUE
            D=1.0/D
            RETURN
            END
```

The first thing to do is pull FUNCTION D into the 120 loop (making the appropriate substitutions). Then we must get rid of the IF statement and put the longest loop (J=1,NZ) on the inside.

```
     Restructured:
            DO 119 J=1,NZ
      119   AD(J)=1.0
     C
            DO 3 L=1,JET
            DO 2 K=L,NZ,JET
            Q=X(K)
            TEMP=AD(K)
     C
```

```
       DO 120 J=1,NZ
 120   AD(J)=2.0*AD(J)*(X(J)-Q)
C
       AD(K)=TEMP
     2 CONTINUE
     3 CONTINUE
C
       DO 121 J=1,NZ
 121   AD(J)=1.0/AD(J)
```

This example was timed on a CYBER 176 using VSL (Vector Subroutine Library, a product of Massachusetts Computer Associates). Note the dependence on vector length. (As the vector length increases, the subroutine overhead is further diluted).

FUNCTION D TIMINGS (Microseconds)

VECTOR LENGTHS	SCALAR FTN (4.5 OPT=2)	VECTOR VSL	SCALAR/VECTOR
10	130	120	1.08
20	390	290	1.35
30	850	560	1.52
40	1460	870	1.68
50	2200	1290	1.71
60	3160	1750	1.81
70	4240	2310	1.84
80	5470	2920	1.87
90	6810	3610	1.89
100	8430	4360	1.93

Next we turn to an interesting real-life example (run on the CRAY) showing the overhead vs. vector speedup trade-off. We have a loop where as soon as we find certain conditions to be true, we branch out of the loop. What we have done is to always look at the whole vector length and use the vector mask instruction to tell us whether any of the elements of the vector met the conditions (if the vector mask word is zero, no elements met the conditions). Thus if the conditions are met very often (say on the second iteration through the loop) all the overhead of looking at the whole array (when we could have stopped at number 2) will drag us down. However, if on the average the conditions are

```
      DO 120 J=1,NZ
 120  AD(J)=2.0*AD(J)*(X(J)-Q)
C
      AD(K)=TEMP
    2 CONTINUE
    3 CONTINUE
C
      DO 121 J=1,NZ
 121  AD(J)=1.0/AD(J)
```

This example was timed on a CYBER 176 using VSL (Vector Subroutine Library, a product of Massachusetts Computer Associates). Note the dependence on vector length. (As the vector length increases, the subroutine overhead is further diluted).

FUNCTION D TIMINGS (Microseconds)			
VECTOR LENGTHS	SCALAR FTN (4.5 OPT=2)	VECTOR VSL	SCALAR/VECTOR
10	130	120	1.08
20	390	290	1.35
30	850	560	1.52
40	1460	870	1.68
50	2200	1290	1.71
60	3160	1750	1.81
70	4240	2310	1.84
80	5470	2920	1.87
90	6810	3610	1.89
100	8430	4360	1.93

Next we turn to an interesting real-life example (run on the CRAY) showing the overhead vs. vector speedup trade-off. We have a loop where as soon as we find certain conditions to be true, we branch out of the loop. What we have done is to always look at the whole vector length and use the vector mask instruction to tell us whether any of the elements of the vector met the conditions (if the vector mask word is zero, no elements met the conditions). Thus if the conditions are met very often (say on the second iteration through the loop) all the overhead of looking at the whole array (when we could have stopped at number 2) will drag us down. However, if on the average the conditions are rarely met and most of the time we would be looking at all or most of the array anyway, then we will do well. A listing of the original, its translation for the CRAY, and the timings follow. Notice that if we get through the loop at least three times before being kicked out, the overhead is beaten out by the savings from vector execution. In the actual code, the average is <u>much</u> greater than three iterations, so we win by a lot.

LOOP FROM MATCHM

```
          DO 50 K=1,ID
          IF (KCN(K,KP,4)) GO TO 50
          P=PPLANE(K,KP)
          IF(P.GT.TOP) GO TO 50
          IF(P.LT.BOT) GO TO 50
          IASSOC=IASSOC+1
          GO TO 60
     50   CONTINUE
```

CVP TRANSLATION

```
          CALL SETL("X",ID)
          CALL ILDV("X00",KCN(1,KP,4))
          CALL SVMN("0")
          CALL VMSI("X",MASK1)
          CALL ILDV("X01",PPLANE(1,KP))
          CALL SRSV("X12",TOP)
          CALL SVMP("2")
          CALL VMSI("X",MASK2)
          CALL SRSV("X13",BOT)
          CALL SVMM("3")
          CALL VMSI("X",MASK3)
          MASK=MASK1.AND.MASK2.AND.MASK3
          IF (MASK.NE.0)IASSOC=IASSOC+1
```

MATCHM TIMES IN SECONDS			
VECTOR LENGTH	SCALAR	VECTOR	SCALAR/VECTOR
1	1.021×10^{-3}	1.468×10^{-3}	.70
2	1.276×10^{-3}	1.468×10^{-3}	.87
3	1.531×10^{-3}	1.468×10^{-3}	1.04
4	1.786×10^{-3}	1.468×10^{-3}	1.22
5	2.041×10^{-3}	1.468×10^{-3}	1.39
10	3.316×10^{-3}	1.468×10^{-3}	2.26
15	4.591×10^{-3}	1.468×10^{-3}	3.13
20	5.866×10^{-3}	1.468×10^{-3}	4.00
25	6.716×10^{-3}	1.468×10^{-3}	4.57

REFERENCES

1) S.M. Einstein, J.M. Levesque, G. Wagenbreth, G.M. Waller, Optimal Utilization of Super Computers, Vol. 1 - The Control Data 7600, R & D Associates.

2) Mathew Myszewski, David Loveman, Vectorizer System User's Manual, Massachusetts Computer Associates

3) J.M. Levesque, Vectorizer System Seminar, Massachusetts Computer Associates.

POISSON SOLVERS ON A LARGE ARRAY COMPUTER

by

Chester E. Grosch
Institute of Oceanography
and
Department of Mathematical and Computing Sciences
Old Dominion University
Norfolk, Virginia

ABSTRACT

A recent analysis of the performance of a proposed large array computer for solving numerically the Navier-Stokes equations showed that the major portion of the time was spent solving the Poisson equation for the pressure field.[1] It was assumed, in the analysis, that a parallel relaxation scheme (Red-Black SOR) was used to solve the Poisson equation at each time step. It is well known that, on serial computers, direct methods are more efficient than relaxation methods for the solution of the Poisson equation. This is not necessarily true for the proposed array computer because of the architecture. The memory is distributed throughout the array and data transfers are only possible between nearest neighbor elements in the array; hence long range data transfer is very time consuming. The performance of three classes of Poisson solvers, standard parallel relaxation methods, parallel multi-grid relaxation methods, and parallel direct methods, on this array computer are analyzed. The analysis includes both the arithmetic operation time and the data transfer time imposed by the array architecture.

1. INTRODUCTION

The numerical calculation of turbulent flow fields is one of the major research areas in fluid dynamics.[2] These calculations can be divided, quite roughly, into two classes: true simulations which resolve all of the dynamically significant scales of motion; and phenomenological calculations which make use of turbulence models. The computing power of existing computers is the effective limit on both classes of calculations. There is a need, for both these types of calculation, to increase the number of mesh points (or modes, for spectral calculations) in the flow volume and to reduce the average computation time per mesh point. This requires an increase in both the memory size and speed of computers.

Increasing the memory size allows the use of a finer spatial resolution for a fixed volume, the calculation of a flow in a larger volume at fixed resolution, the use of a more complex, and presumably more accurate, turbulence model, or some combination of these improvements. It is, of course, well known that the number of operations increases at a slightly faster rate than the number of mesh points, so that large problems require substantially more computational effort than small problems.

There have been some major improvements in algorithms; perhaps the most important are the development of efficient spectral methods, via the Fast Fourier Transform,[3] and the various algorithms for the efficient direct solution of the Poisson Problem.[4-6] Quite recently a new algorithm, the Multi-Grid Algorithm, has been developed[7-8] which appears to offer a substantial improvement in performance for a wide class of problems.

However, despite these major improvements in algorithms, progress in computational fluid dynamics has generally been tied to the development of larger and more powerful general purpose computers. Today's large computers incorporate very large memories, arranged in a hierarchy of size and speed and substantial amounts of pipelining and/or parallelisim in their CPU's, in addition to using high speed components. Despite these improvements in computing power, existing computers are barely adequate for many computational fluid dynamics problems.

Some years ago the computational problems associated with turbulence simulations were reexamined in the light of existing and projected computer capabilities.[9] The conclusions of that study were that a few simulations had already been carried out at moderate Reynolds numbers and that, even with the most powerful computers which were likely to be available in the near future, the range of Reynolds numbers for which turbulent flows could be simulated was quite limited. One of the recommendations of the study was to examine the feasibility of building a special purpose computer to carry out numerical flow simulations as well as flow calculations using turbulence models.

Quite recently a tentative design of the architecture has been proposed for a "Navier-Stokes" computer.[1] This architecture, which will be described below, is in essence a large number ($O(10^4)$ to (10^5)) of simple processing and memory cells arranged in an array. An analysis and simulation of a laminar to turbulent flow transition simulation on such a computer showed that[1,10] the major part of the computational effort (more than 80%) was expended in solving the Poisson problem for the pressure field. The efficient parallel solution of the Poisson problem thus appears to be a key element in utilizing a large array computer to solve the Navier-Stokes equations.

There exists an extensive literature on the structure and cost of parallel algorithms.[11,12] The Poisson problem, its generalizations and simplifications, appears to be a favorite test problem for parallel algorithms. The work of Stone[13,14] on parallel tridiagonal equation solvers and Sameh, Chen and Kuck[15] on parallel Poisson and Biharmonic solvers are typical examples. These studies usually consider only the arithmetic operation counts and neglect "overhead" costs such as the time associated with data rearrangement and transfer. Ortega and Voight[16] give an up to date review of the solution of partial differential equations on vector computers. Most of this review is devoted to consideration of Elliptic systems and Ortega and Voight point out the importance of start-up time on pipeline computers as well as the necessity for efficient data management in using computers like Illiac IV and Cray 1.

In this paper an analysis is made of the solution of the Poisson equation on this array computer using a number of different algorithms. It will be shown that the data transfer costs for the parallel algorithms are at least as important, and in some cases, more important than the cost of computation.

The architecture of the array computer is briefly described in section 2. Section 3 contains an analysis of the arithmetic and data transfer counts and total time cost for a number of parallel relaxation schemes for the solution of the Poisson problem. A description of the Multi-Grid Algorithm for the solution of the Poisson problem and its adaptation to the array computer is given in section 4. The results of a number of simulations of the use of standard parallel relaxation schemes as well as multi-grid relaxation methods on the array computer are presented in section 5. Section 6 contains the analysis of the solution of the Poisson problem by a direct method on the array computer, including arithmetic operation and data transfer counts as well as the total cost in time. The results of the analysis and simulations presented in sections 3 through 6 are compared in section 7. Section 8 is a brief summary of conclusions.

2. THE ARRAY COMPUTER

The array computer is a two-dimensional array of N by N cells, see Figure 1. Each cell can communicate directly with its nearest neighbors above and below and to the left and right. There is an end-around connection from top to bottom and the left end to the right end. This interconnection scheme permits the array to be rearranged logically as an N_1 by N_2 array where $N_1 \cdot N_2 \leq N^2$.

A single cell (hereafter called a processing cell or PC) is assumed to contain a modest amount of memory, say 1K or 2K ($K \equiv 1024$) words; some registers, 32 or 64; and an adder-shifter. The data is stored in the memory and the registers store state information and intermediate results. The add-shifter is used to perform binary addition and multiplication by shift-and-add.

All of the PC's are connected to a control unit which issues instructions to the PC's, and receives and interprets state information from them. It is assumed that every PC performs the same operation or is turned off. One way to handle input-output is to treat the array as a set of N parallel shift registers. All data in the PC memories can be shifted, one word at a time, to the right and out the right side of the array. If there are N_3 words of memory per PC, then $N \cdot N_3$ cell-to-cell transfers are required to dump or load the entire memory on N output wires.

There appear to be three potential advantages of this architecture. First, it seems possible to build such a computer using existing technology at fairly modest cost, (each PC is only a few microprocessor chips, perhaps only a single chip) particularly if the intercelluar connections are as minimal as in this preliminary design. Second, technological developments in the semi-conductor industry are leading to increasing complexity and density per chip as well as lower cost per chip which will make the proposed computer even more cost effective. Third, if an N^3 computational domain can be efficiently mapped onto the array such that N^2 operations can be performed in parallel in the N^2 PC, the number of sequential operations can be reduced by $O(N^2)$, a very considerable reduction in time.

In order to exploit the full power of the array computer concept the numerical algorithms must match the architecture. Local communication is relatively cheap and long range communication may be prohibitively expensive because of the cell-to-cell communication. Serial computation,i.e. computation which uses only a small subset of the N^2 processors, must be avoided because such computation makes very ineffective use of the parallel array hardware and is, therefore, very expensive. The challenge in applying the array processor concept is to couch the physical (numerical) problem in terms which do not require large amounts of long range communication and in which almost all PC's perform the same operations nearly all the time.

The basic cell array is rectangular but that does not, of course, restrict the problem which can be solved on the array computer to those in rectangular geometries any more than does the 'rectangular' indexing of FORTRAN arrays. Any geometry which can be mapped onto a cube can be solved on this array computer. Internal boundaries can be easily treated by initializing interior cells as boundary cells.

In discussing problems to be solved on the array computer, it is assumed that one coordinate direction is mapped into the array such that the data at all mesh points at fixed values of the other two coordinates are stored in a single PC. This is called a rod of data and, in terms of the indices (i,j,k), is all data at fixed (i,j). All data at constant (j,k) is called a column and all data at fixed (i,k) is called a row. If k is constant, the data is said to be in a plane.

In order to estimate the time to solve the Poisson problem using different techniques it is necessary to make some estimates of the time required to carry out the primitive operations in a PC. These operations, and the estimated operation times are: In cell transfer- 100 nsec; Cell to cell transfer - 3 μsec; Addition - 500 nsec; and Multiplication - 5 μsec. Although the absolute values of these operation times depend on the technology used to produce the PC's, it is believed that the operation time ratios will be reasonably independent of the technology used to manufacture the PC's. The multiplication time is used as the basic time unit so that the relative cost of the operations are: In cell transfer - 0.02; Cell to cell transfer - 0.6; Addition - 0.1; and Multiplication - 1.0.

3. PARALLEL RELAXATION SCHEMES

It is required to solve the Poisson Problem

$$\nabla^2 p_{i,j,k} = q_{i,j,k} \tag{1}$$

where ∇^2 is a discrete form of the Laplace operator

$$\nabla^2 \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$$

on a spatial grid with suitable (Neumann, Dirchlet, Periodic, or Mixed) boundary conditions and with given $q_{i,j,k}$.

Applying the seven point centered difference operator to Eq. (1), we have

$$p_{i+1,j,k} - 2p_{i,j,k} + p_{i-1,j,k} + \beta^2(p_{i,j+1,k} - 2p_{i,j,k} + p_{i,j-1,k}) +$$
$$\gamma^2(p_{i,j,k+1} - 2p_{i,j,k} + p_{i,j,k-1}) = (\Delta x)^2 q_{i,j,k}, \qquad (2)$$

where

$$\beta = \Delta x/\Delta y, \ \gamma = \Delta x/\Delta z\,.$$

It is assumed that there are periodic boundary conditions in the z direction,

$$p_{i,j,0} = p_{i,j,N_3} \qquad (3)$$

and Neumann boundary conditions on the other faces,

$$p_{1,j,k} - p_{-1,j,k} = 2\Delta x\tilde{f}_{0,j,k} = f_{0,j,k} \qquad (4a)$$

$$p_{n+1,j,k} - p_{n-1,j,k} = 2\Delta x\tilde{f}_{n,j,k} = f_{n,j,k}, \qquad (4b)$$

$$p_{i,1,k} = p_{i,-1,k} = 2\Delta y\tilde{g}_{i,0,k} = g_{i,0,k} \qquad (4c)$$

$$p_{i,m+1,k} - p_{i,m-1,k} = 2\Delta y\tilde{g}_{i,m,k} = g_{i,m,k}. \qquad (4d)$$

This set of boundary conditions arises, for example, in the solution for the pressure field in the simulation of transition from laminar to turbulent flow.[1]

Fourier transforming $p_{i,j,k}$; $q_{i,j,k}$ and the boundary conditions along the z(k) direction yields

$$p_{i-1,j,\alpha} - 2\sigma\, p_{i,j,\alpha} + p_{i+1,j,\alpha} + \beta^2(p_{i,j-1,\alpha} + p_{i,j+1,\alpha}) = (\Delta x)^2\, q_{i,j,\alpha}, \tag{5}$$

where the $\{p_{i,j,\alpha}\}$ and $\{q_{i,j,\alpha}\}$ are the Fourier coefficients of the $\{p_{i,j,k}\}$ and $\{q_{i,j,k}\}$, and

$$\sigma = 1 + \beta^2 + \gamma^2 - \gamma^2 \cos\left(2\pi \frac{\alpha}{N_3}\right). \tag{6}$$

The transformed boundary conditions are of the same form as Eq. (4a-d) with k replaced by α.

This reduces the solutions of one 3D Poisson problem to the solution of N_3 uncoupled 2D problems. In this section the solution of the 2D problems by relaxation will be analyzed. The subscript α will be dropped hereafter to simplify the notation.

The relaxation scheme must be a parallel one in order to exploit the inherent speed-up of the array architecture. The possible relaxation schemes appear to be:

The Jacobi method,[17]

$$p_{i,j}^{(k+1)} = \frac{1}{2\sigma}\left\{ p_{i+1,j}^{(k)} + p_{i-1,j}^{(k)} + \beta^2(p_{i,j+1}^{(k)} + p_{i,j-1}^{(k)}) - (\Delta x)^2\, q_{i,j}\right\}; \tag{7}$$

The single weight Jacobi method,[7]

$$p_{i,j}^{(k+1)} = p_{i,j}^{(k)} + \omega_0\, \bar{p}_{i,j}^{(k)}, \tag{8}$$

where $\bar{p}_{i,j}^{(k)}$ is calculated from Eq. (7) and ω_0 is a weight;

The five weight Jacobi method,[7]

$$p_{i,j}^{(k+1)} = p_{i,j}^{(k)} + \omega_1\, \bar{p}_{i,j}^{(k)} + \omega_2\, (\bar{p}_{i+1,j}^{(k)} + \bar{p}_{i-1,j}^{(k)} + \bar{p}_{i,j+1}^{(k)} + \bar{p}_{i,j-1}^{(k)}) \tag{9}$$

where $\bar{p}^{(k)}_{i,j}$, $\bar{p}^{(k)}_{i\pm1,j}$, and $\bar{p}^{(k)}_{i,j\pm1}$ are calculated from Eq. (7) and ω_1 and ω_2 are weights;

The Red-Black SOR method[17]

$$p^{(k+1)}_{i,j} = (1 - \omega)\ p^{(k)}_{i,j} + \omega\bar{p}^{(k)}_{i,j} \tag{10}$$

for i + j odd and, seperately, i + j even, ω is a weight, and $\bar{p}^{(k)}_{i,j}$ is computed from Eq. (7). In all of the above the superscripts denote the iteration number.

It is a simple matter to find the operation counts (in-cell transfers N_t, cell to cell transfers N_T, additions N_a, and multiplications N_m) for each of these methods. For example, the Jacobi method requires no in-cell transfers, four cell to cell transfers, to obtain $p_{i\pm1,j}$ and $p_{i,j\pm1}$, four additions and two multiplications. It is assumed that $(\Delta x)^2 q_{i,j}$, and $1/(2\sigma)$ have been precomputed and stored along with β^2.

After each relaxation sweep the current residuals,

$$r^{(k)}_{i,j} = \nabla^2 p^{(k)}_{i,j} - q_{i,j}, \tag{11}$$

must be calculated.

These operation counts are given in Table I. The time for each of these procedures, in units of one multiplication time, are also given. The basic times for each of the operations are as given in section 2.

The results given in Table I show that all of these relaxation schemes, except the five weight Jacobi, cost about the same. The five weight Jacobi method is nearly twice as costly as the others and the calculation of the residuals costs about 50% more than a typical relaxation sweep. Because of its superior rate of convergence and low cost the Red-Black SOR method is clearly the best of these parallel relaxation methods.

4. ANALYSIS OF THE MULTI-GRID ALGORITHM (MGA)

The MGA is a multi-level adaptive solution strategy which intermixes the discretization and solution processes for the numerical solution of boundary value problems. Brandt[7] has described, analyzed, and given numerous examples of the MGA. In this section, the application of the MGA to the solution of two and three dimensional Poisson problems on the array computer described in section 2 will be analyzed.

4.1 Multi-Grid Solution of the Poisson Problem

It is desired to solve the Poisson problem, as described in section 3, on some grid. The given grid is taken to be the finest grid. It is convenient, but not necessary, to assume that the number of grid points, including boundary points, in each direction is of the form $2^{M_\alpha} + 1$, M_α an integer; $\alpha = 1, 2,$ or 3, corresponding to the x, y, z coordinates. This grid is called the M grid, $M = \min\{M_\alpha\}$. Now consider a sequence of coarser grids, $M - 1, M - 2, \ldots, M - m$, $m \geq M - 2$, which are nested within the finest grid. Each of these grids has $2^{M_\alpha - j} + 1$, $1 \leq j \leq m$, grid points in each coordinate direction. Brandt has shown that this ratio of 1:2 in the grid spacing for the different grids is optimum. The restriction that $m \geq M - 2$ ensures that there are a minimum of five grid points in each direction on the coarsest grid, so that there is at least one purely interior point in the coarsest grid.

The basic solution technique is relaxation using one of the parallel relaxation schemes described in section 3. The first few relaxation sweeps on the finest grid are very effective in reducing the norm of the residuals, $||r_{i,j}||$. The rate of reduction of the residuals decreases after a few relaxation sweeps and, asymptotically, becomes very slow even for the best relaxation methods. The objective of the MGA is to preserve the rapid reduction of $||r_{i,j,}||$ and prevent the asymptotic reduction of the convergence rate.

The MGA procedes according to the rules:

(a) If the rate of reduction of the norm of the residuals slows down so that

$$||r_{i,j}^{(\ell+1)}|| / ||r_{i,j}^{(\ell)}|| \geq \eta, \tag{12}$$

η a constant and < 1, and ℓ the relaxation index, transfer the residuals to a coarser grid.

(b) Relax the solution on the coarser grid so as to reduce the residuals to zero. If the rate of reduction slows down, according to criterion (12), transfer to a still coarser grid. If the relaxation converges on the coarse grid;

$$||r_{i,j}||_{coarse}/||r_{i,j}||_{fine} \le \delta, \qquad (13)$$

where $||\ ||_{coarse}$ is the latest value of the norm on the coarse grid, $||\ ||_{fine}$ is the last value of the norm on the next finer grid, and δ is a constant, < 1; transfer the corrections to $p_{i,j}$, back to the finer grid and proceed.

(c) On the coarsest grid, continue the relaxation until the convergence criterion (13) is satisfied.

(d) Overall convergence occurs when the norm of the residuals is reduced to the desired level on the finest grid.

The MGA proceeds by smoothing the residuals by relaxation on a grid. The relaxation process is most effective for the Fourier components of the residuals whose wavelengths are of the order of twice the grid spacing. After these short wavelength residuals are destroyed, the rate of reduction of the residuals decreases because relaxation is less effective at smoothing the long wavelength residuals. The MGA speeds up convergence by shifting to a coarser grid where some of the longer wavelength residuals have wavelengths of the order of twice the grid spacing.

The efficiency of the MGA is due to; first the actual reduction of the number of iterations required for convergence; and second, the reduction in the number of arithmetic operations necessary to carry out a sequential relaxation sweep through a coarse grid as compared to a fine grid. The reduction of the number of iterations is quite substantial but the reduction in the number of arithmetic operations can be very great for a computer which is sequential or almost so (the degree of parallelism is much smaller than the number of mesh points in the grid). Shifting from any grid to the next coarser grid reduces the number of sequential arithmetic operations by a factor of 2^d, where d is the dimension of the problem. The combination of fewer iterations and, on average, fewer arithmetic operations per iteration results in a very large speedup in the relaxation process.

4.2 Analysis of the Multi-Grid Algorithm on an Array Computer

The architecture of the array computer imposes severe constraints on the MGA. In a conventional computer which has a central memory every piece of data is equally accessable, or inaccessable. Because the memory is distributed in the array computer the transfer of a data word between two PC's whose indices differ by N requires N data transfers.

Assume that the finest grid coincides with part of, perhaps the whole of, the array. Further assume that a five point star is used for a 2D relaxation and a seven point star is used for a 3D relaxation. Then, all the data required for a relaxation is in a PC or, at most, requires a single transfer per word from a nearest neighbor PC. After a shift to the first coarse grid the rods of data needed for the relaxation sweep are in PC's whose indices differ by two so that two data transfers per word are required. After a further transfer to the next coarsest grid the PC indices differ by four and four data transfers are needed per word, and so on.

There are two possible strategies that can be used in the application of the MGA. The first is to leave the data in place and accept the increase in the number of data transfers for each relaxation. The second strategy is to transfer the data into nearest neighbor PC's; in effect compress the working array and pay the increase transfer cost once per grid shift. Note that this strategy requires that the working array be expanded and a similar transfer penalty be paid when there is a shift from a coarse to a finer grid.

If the first strategy, leaving the data in place, is used, the number of cell to cell data transfers per relaxation is increased by the factor $2^{M-\ell}$ on grid ℓ, for a 2D or 3D problem. The data transfers in the third coordinate direction are along the data rod and only require internal transfers. A typical MG cycle has K iterations on grid M; a shift to grid M - 1, followed by K iterations, and so on until the coarsest grid, grid 2, is reached. After K iterations on grid 2, there is a shift to grid 3, K iterations, and so on until grid M is reached. If there are L_0 cell to cell data transfers per iteration, the total number of cell to cell transfers in one MG cycle is then,

$$L_1 = KL_0 + 2KL_0 \sum_{\ell=2}^{M-1} 2^{M-\ell} = KL_0\,(2^M - 3). \qquad (14)$$

The second strategy requires that the working array be compressed each time there is a shift from a fine to coarse grid and an expansion every time there is a shift from a coarse to a fine grid. The number of data transfers to do one array compression and expansion between grids ℓ and $\ell - 1$ is $4 \cdot 2^{\ell-1}$

for a 2D array or one plane of a 3D array. Therefore the total number of grid transfers to compress and expand a 2D array or one plane of a 3D array for a MG cycle is

$$\sum_{j=2}^{M} 2^{\ell+1} = 2(2^{M+1} - 4) .$$

There are M - 2 grid levels, K relaxations at each level, and L_0 transfers per relaxation so that the total number of data transfers for a 2D problem, using the second strategy, L_{22}, is

$$L_{22} = KL_0(M - 2) + 2(2^{M+1} - 4) . \tag{15}$$

For a 3D Poisson problem with N_3 layers, the total number of data transfers is

$$L_{23} = KL_0(M - 2) + 2(2^{M+1} - 4)N_3 . \tag{16}$$

The ratio of L_{22} to L_1 is

$$L_{22}/L_1 = \left(\frac{M - 2}{2^M - 3}\right) + \frac{2(2^{M+1} - 4)}{KL_0(2^M - 3)} . \tag{17}$$

For M > > 1,

$$L_{22}/L_1 \simeq 4/KL_0 \le \frac{1}{2} , \tag{18}$$

because $K \ge 2$ and $L_0 \ge 4$. For any $M \ge 3$, $K \ge 2$, and $L_0 \ge 4$, Eq. (17) shows that $L_{22}/L_1 < 1$. Exact values of this ratio are given in Table II.

Next considering the 3D problem, the ratio of L_{23}/L_1 is

$$L_{23}/L_1 = \left(\frac{M - 2}{2^M - 3}\right) + \frac{2(2^{M+1} - 4)}{KL_0(2^M - 3)} N_3 . \tag{19}$$

If $M >> 1$,

$$L_{23}/L_1 \simeq 4N_3/KL_0 >> 1 \tag{20}$$

because the number of planes in the third direction, $N_3 >> 1$. Exact values of this ratio are also given in Table II, and it can be seen that this ratio is greater than 10 for reasonable choices of K, L_0, and N_3.

It can be concluded that in applying the MGA to 2D problems the array should always be compressed and expanded, but that the data should always be left in place for 3D Poisson problems. This analysis also suggests that the introduction of a few additional data paths, connecting the first and ninth PC's, the ninth and seventeenth PC's etc. but not the second and tenth, could decrease the number of transfers sufficiently so as to make the in place strategy optimum for the 2D problem.

Shifting from a fine to a coarse grid also involves the additional cost of the transfer of the residuals. Simple injection has the minimum cost; the residuals at points where the coarse and fine grids coincide are multiplied by four, for 2D problems or by eight for 3D problems, in order to take into account the doubling of the mesh size. In addition there are two in-cell transfers, one to store the fine grid solution and the other to store the fine grid residuals.

Certain relaxation schemes require a weighting of the residuals. The full weighting scheme gives, in 2D,

$$\begin{aligned}\tilde{r}_{i,j} &= r_{i,j} + \tfrac{1}{2}(r_{i,j+1} + r_{i,j-1} + r_{i-1,j} + r_{i+1,j}) + \\ &\tfrac{1}{4}(r_{i-1,j+1} + r_{i-1,j-1} + r_{i+1,j+1} + r_{i+1,j-1})\end{aligned} \tag{21}$$

where $\tilde{r}_{i,j}$ is the weighted coarse grid residual, and $r_{i,j}$ is the fine grid residual. This method uses no internal transfers, six cell to cell transfers, eight additions and two multiplications and is, therefore, very costly as compared to simple injection.

Shifting from a coarse to fine grid is somewhat more costly. The simplest possible method of transfering the corrections to the solution from the coarse to fine grid is by linear interpolation.

Figure 2 shows a section of the array. Each block represents one cell which is labeled by the appropriate indices. The heavy lines deliminate the primitive correction block; all fine grid solutions in this block are corrected by the local coarse grid solution and the correction algorithm is periodic, with period 2, in both the i and j indices. If cell (i,j) is in both the fine and coarse grids, the corrected fine grid solution (in 2D) is

$$p_{i,j} = \hat{p}_{i,j} + \tilde{p}_{i,j}, \tag{22}$$

where $p_{i,j}$ is the corrected solution $\hat{p}_{i,j}$ is the old fine grid solution, and $\tilde{p}_{i,j}$ is the correction to the fine grid solution which is obtained from the coarse grid solution. Similarly in fine grid cells, (i,j-1) and (i-1,j)

$$p_{i,j-1} = \hat{p}_{i,j-1} + \frac{1}{2}(\tilde{p}_{i,j-1} + \tilde{p}_{i,j}), \tag{23}$$

$$p_{i-1,j} = \hat{p}_{i-1,j} + \frac{1}{2}(\tilde{p}_{i-1,j} + \tilde{p}_{i,j}), \tag{24}$$

and in cell (i-1, j-1)

$$p_{i-1,j-1} = \hat{p}_{i-1,j-1} + \frac{1}{4}(\tilde{p}_{i-1,j} + \tilde{p}_{i-1,j+1} + \tilde{p}_{i,j-1} + \tilde{p}_{i,j+1}). \tag{25}$$

Note that $\tilde{p}_{i-1,j}$, which is in the coarse mesh, is two cells to the left of $\hat{p}_{i,j}$ and one cell to the left of $p_{i-1,j}$ in the fine mesh.

A study of Figure 2 and Eq. (22) - (25) shows that linear interpolation requires four internal transfers, six cell to cell transfers, four additions and one multiplication. The particular sequence of operations which has been chosen minimizes the number of cell to cell transfers and multiplications at the cost of adding some (very inexpensive) internal transfers. The operation counts and costs of these operations, compression, expansion, injection, both simple and weighted as well as linear interpolation are given in Table III.

This analysis shows that there are substantial overhead costs in applying the MGA on the array computer because of the constraints imposed by the architecture. A coarse grid relaxation is executed by turning off the fine grid cells which are not in the coarse grid. Therefore the cost, in operations, of a coarse grid relaxation is exactly the same as that of a fine grid relaxation.

It can be seen that, when the overhead operations of compression and expansion or long range transfer, injection, and interpolation, are included, a coarse grid relaxation is more expensive in operations and time than a fine grid relaxation. Problems in 3D are relatively less expensive than 2D problems because the inplace strategy reduces the number of operations on a coarse grid to one-half the number on a fine grid. Basically, the major improvement in using the MGA on an array computer is due to the reduction in the absolute number of iterations needed for convergence.

It is clear that 2D Poisson problems are a 'worst case' because there are no reductions for the operation counts of a coarse grid relaxation compared to a fine grid relaxation. Therefore, from this point on, only 2D Poisson problems will be considered.

The relative costs given in Table I show that the cost per grid sweep, which is a relaxation sweep plus a calculation of the residuals is roughly the same for all of the methods. From the results given in Table II, it is clear that simple injection is very inexpensive and weighted injection and linear interpolation cost of the order of one-half of a grid sweep. In contrast compression and expansion are very costly. A compression from level 7 to level 6 and an expansion from level 6 to level 7 has a cost of 153.6 units, which is of the order of ten grid sweeps. It is clear then that the use of the MGA must reduce the number of grid sweeps by a considerable factor in order for it to be competitive with simple relaxation.

It is possible to derive a theoretical 'efficiency rating,' λ, for the MGA; but note that this differs from the convergence factor defined by Brandt.[7] Let $\bar{\mu}$ be the theoretical smoothing rate, the factor by which the high frequency errors are reduced by a single sweep over a grid. Brandt[7] has calculated $\bar{\mu}$ for a number of relaxation schemes and shown that a MG cycle with K sweeps on each grid level will reduce all the error components to, approximately, $\bar{\mu}^K$ of the original error.

Now let w be the cost of a MG cycle. If M is the finest grid level and the coarsest grid level is 2, K grid sweeps are carried out on each level, and grid compression and expansion are used, then

$$w = K \cdot (M - 2) \cdot (\text{relaxation cost} + \text{residual cost}) + (M - 2) \cdot (\text{injection cost} + \text{interpolation cost}) + 2(\text{single transfer cost}) \sum_{j=2}^{M} 2^j . \tag{26}$$

The error reduction efficiency, $\tilde{\mu}$, is

$$\tilde{\mu} = 1 - \bar{\mu}^K; \tag{27}$$

and the error reduction efficiency per unit work, λ, is given by

$$\lambda = \tilde{\mu}/w = (1 - \bar{\mu}^K)/w \tag{28}$$

Clearly the larger $\tilde{\mu}$, and more important, λ, is the more efficient is the algorithm. Table IV lists values of λ for various K and M. The values of $\bar{\mu}$ given in Table IV are those given by Brandt.[7] It can be seen that, for a given method and size of grid (M), there is an optimum number of iterations per grid level. This optimum is two for the Single Weight Jacobi and Red-Black SOR methods and one for the Five Weight Jacobi method. However, for M = 7, the problem size of most interest, the use of two iterations per grid level is only about 11% less efficient than performing only a single iteration per grid level. It can also be seen that, in general, the Five Weight Jacobi method is, theoretically, the most efficient followed by Red-Black SOR and the Single Weight Jacobi methods.

5. SIMULATION OF STANDARD AND MG RELAXATION METHODS

A set of Fortran programs have been written which simulate the relaxation solution of 2D problems using standard parallel algorithms and the MG relaxation algorithm on the array computer. The MG simulators incorporate parts of the sample program (called CYCLEC) given by Brandt.[7] The simulators can handle rectangular or square grids up to M = 7 ie. grids which have less points than 129 by 129. In addition to the number of relaxations and residual calculations, the number of 'overhead' operations; compressions, expansions, injections, and interpolations, are also counted. The output of the residuals after each relaxation permits the observed error reduction efficiency, $\tilde{\mu}$, to be calculated. The total cost is also calculated using the unit operation costs given in Table I and III. Finally the error reduction efficiency per unit work and the total cost of obtaining a solution are computed.

The results of the simulations are relatively insensitive to the grid aspect ratio as long as it is not too large. Large aspect ratio grids require the use of line relaxation methods, whose implementation on the array computer have not yet been analyzed or simulated. Therefore only the results of simulations on square grids with M = 5(33 by 33), 6(65 by 65) and 7(129 by 129) will be given here.

Simulations of the relaxation solution of two different Poisson problems have been run so far. For both of these problems the solution is to be found in $0 \le x \le 1$, $0 \le y \le 1$ with Dirchlet conditions on the boundary. For problem 1 the right hand side of the Poisson equation is $\sin[3(x + y)]$ and the boundary values are given by $\cos[2(x + y)]$. The right hand side for problem 2 is a set of random numbers in (-1,1) with a mean of zero and the boundary values are zero. For both problems the initial approximation was taken to be the boundary values, i.e.

$$p_{i,j} = \cos[2(x_i + y_j)]; \text{ problem 1}$$

$$p_{i,j} = 0 \qquad ; \text{ problem 2.}$$

Results of the simulations of the MGA for these two problems are given in Table V. The relaxation methods are the Single and Five Weight Jacobi methods and Red-Black SOR. The weights, listed in Table V, are the optimum weights for these methods as determined by Brandt.[7] Table V lists the method; the problem (1 or 2); the grid size, M (there are $2^M + 1$ grid points in each direction); the theoretical and observed error reduction efficiency, $\tilde{\mu}$; (with K = 2) and the theoretical and observed error reduction efficiency per unit work, λ.

Examination of Table V shows that, first, the observed error reduction efficiency per unit work is always greater than the theoretical value. The difference ranges from about 5% to 40% for both problems. The observed values of $\tilde{\mu}$ and λ for the Five Weight Jacobi method fall between the theoretical values for K = 2 and K = 1. This can be explained by noting that in the simulations the algorithm always performed two iterations per grid level on the fine to coarse grid shift but only one iteration per grid level on the coarse to fine shift.

The Red-Black SOR method has an observed error reduction efficiency close to that for K = 4 ($\tilde{\mu} = 0.9375$) but the cost per MG cycle is about that for K = 2 so that the error reduction efficiency per unit work is even greater than that of the Five Weight Jacobi. The parallel execution of both a Red and a Black relaxation makes this the most efficient method for the MGA on the array computer.

The results given in Table V are quite revealing, but the most meaningful comparison between methods is given by the total cost to achieve convergence. These results are given in Table VI. In Table VI the column labeled 'Transfers' has sub-columns labeled 7-6-...-2. These indicate grid compressions from grid 7 to 6, 6-5, etc. There is a grid expansion for every compression. The column labeled 'Relaxations' is the total number of relaxations on all grids and the column labeled 'Overhead Cost' gives the fraction in percent of the total work expended in 'Overhead', which is here defined as compression, expansion, injection, and interpolation. The column labeled 'Transfer Cost' lists the fraction in percent of the total cost that is due to data transfers. Finally the column labeled 'Total Cost' is just that, in units of a multiplication time. The solution is considered to have converged when the norm of the residuals is less than ε; the values of ε used are also listed in Table VI.

The results given in Table VI clearly show the superiority of the MGA using the Red-Black SOR relaxation scheme. This method (depending on the problem, grid size, and convergence criterion) costs 15% to 35% less than the corresponding Five Weight Jacobi method and only about 50% of the Single Weight Jacobi method. The improvement over the standard parallel Red-Black SOR method is even more striking. The MGA reduces the cost by a factor of from four to seven as compared to the standard parallel Red-Black SOR method for large (M = 7) problems. The standard method has, however, lower cost for small problems (M = 5). These results suggest that, the MGA for Red-Black SOR is the best relaxation method to use on the array computer for the large problems of most interest in fluid dynamic simulations. A $(129)^3$ grid is about the smallest grid for which it would be worthwhile to build an array computer to carry out fluid dynamic simulations. It is hoped that the technology will eventually be available to build a 1025 by 1025 array computer with, say 512 grid points per PC. The advantage of the MG Red-Black SOR for these large problems would be even greater.

The results given in Table VI also show that overhead cost (arithmetic plus transfers) of all of these relaxation methods is fairly substantial. It is of the order of 40% to 60% of the total cost. The transfer cost is even larger; counting transfers in relaxation, residual calculation and overhead, it accounts for about 60% to 75% of the total cost. That is the arithmetic operations account for only 25% to 40% of the total cost. This shows the necessity of accounting for overhead and transfers in determining the cost of running an algorithm on this computer. If overhead operations had been neglected, the time cost would have been underestimated by about a factor of two. If data transfer costs had been ignored the total time cost would have been underestimated by a factor of three or four. If both data transfer costs and overhead costs had been neglected the cost would have been underestimated by a factor of five or more.

6. DIRECT SOLUTION OF THE POISSON EQUATION

There are a number of algorithms for the direct solution of the Poisson equation in 2D and 3D which are mathematically equivalent. Each of these algorithms generate a solution in a finite number of steps which is, in principle, exact to within the computer round-off error. The Buneman form of the cyclic odd-even reduction algorithm has been chosen for analysis because it is both representative of this class of algorithms and is a practical (numerically stable) algorithm. The notation used here is a slight variant on that used by Buzbee, Golub, and Nielson[6] in their discussion of this, and other, direct algorithms for the Poisson equation.

The vectors $\{ X_j \}$ and $\{ Y_j \}$, $j = 0,1,\ldots,m$ are defined so that X_j is the column vector with elements $\{ p_{i,j,\alpha} \}$, $i = 0,1,\ldots,m$ and

$$Y_j = (\Delta y)^2 \, Y_j^{(1)} + (\delta_{0j} + \delta_{mj}) \, Y_j^{(2)} + \beta^{-2} \, Y_j^{(3)}, \qquad (29)$$

where $Y_j^{(1)}$ is the column vector with elements $\{ q_{i,j,\alpha} \}$

$Y_j^{(2)}$ is the column vector with elements $\{ g_{i,j,\alpha} \}$

$Y_j^{(3)}$ is the column vector with elements $\{ f_{i,j,\alpha} \}$,

(see equations 4a-4d) with $i = 0,1,\ldots,n$, and where δ_{0j} and δ_{mj} have the usual meaning. The equations (4) and (5) are equivalent to the system

$$A X_0 + 2 X_1 = Y_0 \tag{30a}$$

$$X_{j-1} + A X_j + X_{j+1} = Y_j, \; j \neq 0,m \tag{30b}$$

$$2 X_{m-1} + A X_m = Y_m . \tag{30c}$$

In these equations A is β^{-2} times a tridiagonal matrix with diagonal elements equal to -2σ, super-diagonal elements $\{ 2,1,\ldots,1 \}$, and sub-diagonal elements $\{ 1,1,\ldots,1,2 \}$. All of the matrices are (n+1) by (n+1) and $m = 2^{k+1}$, with k an integer. In order to compare the results of this section with the results given in previous sections for the relaxation methods, note that $k = M - 1$. It will be assumed that the $\{ Y_j \}$ have been precomputed and stored in the rows of the array computer. The $\{ X_j \}$ will also be stored in the rows.

The Buneman algorithm has two variants, of which the second is less efficient than the first on the array computer, because it requires considerably more transfers. Therefore, only the first variant will be analyzed.

The Buneman algorithm has two stages; reduction and back-substitution (for details see Buzbee, Golub, and Nielson[6]). The reduction stage requires the computation and storage in the rows of the array computer of a set of column vectors $\{ P_j^{(r)} \}$, $\{ Q_j^{(r)} \}$, and $\{ R_j^{(r)} \}$, $r = 1,2,\ldots,k+1$.

These are computed from

$$A^{(0)} R_j^{(1)} = Y_j \tag{31}$$

$$P_j^{(1)} = R_j^{(1)}, \; j = 0,2,\ldots,m \tag{32}$$

$$Q_0^{(1)} = 2(Y_1 - P_0^{(1)}) \tag{33a}$$

$$Q_j^{(1)} = Y_{j+1} + Y_{j-1} - 2P_j^{(1)}, \; j = 2,4,\ldots,m-2 \tag{33b}$$

$$Q_m^{(1)} = 2(Y_{m-1} - P_m^{(1)}) . \tag{33c}$$

Then, for $r = 1,2,\ldots,k-1$

$$A^{(r)} R_0^{(r)} = 2P_{2^r}^{(r)} - Q_0^{(r)}, \tag{34a}$$

$$A^{(r)} R_j^{(r)} = P_{j+2^r}^{(r)} + P_{j-2^r}^{(r)} - Q_j^{(r)}, \tag{34b}$$

$$A^{(r)} R_m^{(r)} = 2P_{m-2^r}^{(r)} - Q_m^{(r)}, \tag{34c}$$

$$P_j^{(r+1)} = P_j^{(r)} - R_j^{(r)} \tag{35}$$

$$Q_0^{(r+1)} = 2(Q_{2^r}^{(r)} - P_0^{(r+1)}) \tag{36a}$$

$$Q_j^{(r+1)} = Q_{j+2^r}^{(r)} + Q_{j-2^r}^{(r)} - 2P_j^{(r+1)} \tag{36b}$$

$$Q_m^{(r+1)} = 2(Q_{m-2^r}^{(r)} - P_m^{(r+1)}) \tag{36c}$$

where $j = i2^{r+1}$, $i = 1,2,\ldots,2^{k-r}-1$.

Finally for $r = k$

$$A^{(k)} R_{2k}^{(k)} = P_m^{(k)} + P_0^{(k)} - Q_{2k}^{(k)}, \tag{37}$$

$$P_{2k}^{(k+1)} = P_{2k}^{(k)} - R_{2k}^{(k)}. \tag{38}$$

In these equations

$$A^{(0)} = A, \tag{39a}$$

$$A^{(r+1)} = 2I - (A^{(r)})^2, \tag{39b}$$

with I the identity matrix. It is not necessary to calculate $Q_{2k}^{(k+1)}$ because it may be assumed that the singular solution is zero for fluid dynamic applications.

The $\{ X_j \}$ are solved for in the back-substitution stage by

$$X_{2k} = P_{2k}^{(k+1)}, \tag{40}$$

$$A^{(k)} R_0^{(k)} = Q_0^{(k)} - 2X_{2k} \tag{41}$$

$$X_0 = P_0^{(k)} + R_0^{(k)} \tag{42}$$

$$A^{(k)} R_m^{(k)} = Q_m^{(k)} - 2X_{2^k} \quad (43)$$

$$X_m = P_m^{(k)} + R_m^{(k)} . \quad (44)$$

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

$$A^{(r)} R_j^{(r)} = Q_j^{(r)} - (X_{j+2^r} + X_{j-2^r}) \quad (45)$$

$$X_j = P_j^{(r)} + R_j^{(r)} \quad (46)$$

with $j = 2^r, 3 \times 2^r, \ldots, 2^{k+1} - 2^r$. Note that the $\{ R_j^{(r)} \}$ are intermediate results.

It is clear from an examination of equations (31) through (46) that the major portion of the computational effort is expended in obtaining the $\{ R_j^{(r)} \}$. It is obvious that at each level, $r = 1,2,\ldots,k$, the equations for the $\{R_j^{(r)} \}$ can be solved in parallel for all j by using the PC's in the individual rows in parallel. The key to an efficient implementation of the Buneman algorithm on this array computer is then the efficient solution of a 1D set of equations in a row of PC's.

6.1 Solution of a 1D Problem

The typical problem is to solve the equation

$$BU = V \quad (47)$$

for $U = \{ u_i \}$, with $V = \{ v_i \}$ known and stored in a set of n+1 PC's. The matrix B is (n+1) by (n+1) and $n = 2^{s+1}$ with s an integer.

The most direct way of solving for U is by matrix multiplication. B^{-1} is precomputed and the ith row is stored in the ith PC. A total of n cell-to-cell and n^2 in cell transfers using the end around connection are required to move all of the $\{ v_i \}$ into all of the PC's and arrange them in sequential order. The calculation of U can then be done with n+1 multiplications and n additions. The cost of this method is

$$C_1 = 1 + 2^{s+1} (1.70 + 0.02 \times 2^{s+1}), \quad (48)$$

and n+1 words of permanent store are required in the ith PC for the ith row of B^{-1}.

In addition n+1 words of store are needed for the temporary storage of V in each PC.

An alternate method is to use the 1D version of Buneman's algorithm. The matrix B is, at the r^{th} level, $A^{(r)}$ which can be written as the product of 2^r tridiagonal matrices. Then U can be obtained by solving 2^r tridiagonal systems (see Buzbee, et al.[7] and Swartztrauber[18] for details).

The application of the Buneman algorithm gives rise to the same problem as the MG algorithm; initially all the data are stored in adjacent PC's, but after one level of reduction, the relevant data are stored in PC's whose indices differ by two, requiring two cell-to-cell transfers, etc. It can be shown that as for the MGA, the optimum strategy for large problems is to compress and expand the data array. Just as for the MGA, it can be shown that a total of $4(2^s-1)$ cell-to-cell transfers are required to compress and expand the P and Q vectors.

Apart from compression and expansion, it can be shown that the 1D version of equations (31) through (46) requires (4+14s) in cell transfers, (6s) cell-to-cell transfers, (5+10s) additions, and (2+2s) multiplications. Ths cost of using this algorithm is easily shown to be

$$C_2 = 2^r \left(2.40 \times 2^s + 6.88s + 0.18 \right) \tag{49}$$

The values of $2^r C_1/C_2$ increase from 1.15 to 17.6 as s increases from 4 to 9. It is therefore clear that for a single tridiagonal equation (r=0) the 1D Buneman algorithm is always less costly and, for large s, very much less costly than matrix multiplication on this array computer. The reason is that for large s the cost of the internal transfers necessary to rearrange the data for matrix multiplication comes to dominate the total cost. However, for any s, matrix multiplication is the less costly method for sufficiently large r because the 1D Buneman algorithm must be applied 2^r times.

The matrix method requires n+1 words of memory per PC for each level of reduction for which it is used. It appears to be worthwhile to allocate this storage because the reduction in the time cost is substantial.

The optimal strategy to solve the sequence of 1D problems is then to use the 1D Buneman algorithm for small r when it is the more efficient method and switch to matrix multiplication at large r for which this method is the more efficient. This strategy will be used in the application of the 2D Buneman algorithm.

6.2 Operation Counts and Time Cost of the 2D Buneman Algorithm

Once the operation counts and time costs for the solution of the 1D problem have been obtained it is a simple matter to find them for the 2D Buneman algorithm from Eq. (31) through (46). It is easily shown that just as for the 1D case, the least costly strategy is to compress the array of P and Q vectors during the reduction stage and expand it during the back-substitution stage.

The operation counts and time costs for the 1D problem, using both matrix multiplication and the Bureman algorithm, as well as for the 2D problem, are given in Table VII. The 1D problem has n+1 variables and the 2D problem has $(n+1)\cdot(M+1)$ variables, where $n = 2^{s+1}$ and $M = 2^{k+1}$.

In this table, the operation counts and costs for the solution of the 1D problem discussed in section 6.1 are given first. Note that the counts and cost for the 1D Buneman algorithm do not include the operation of compression and expansion. The operation counts and cost of the 2D Buneman algorithm do not include those of compression and expansion, and the solution of the 1D problems. Two sets of vectors, the P's and Q's must be compressed and expanded, and two 1D problems must be solved at each level from $r = 0$ to $r = k$.

In Table VIII the total cost of the direct solution using the Buneman algorithm of the Poisson equation for the cases $k = s = 4,5$ and 6 is listed. The basic cost unit is one multiplication time. These correspond to the relaxation solutions for $M = 5,6$ and 7. Also given in Table VIII are the cost of solving the 1D problems and the transfer cost. Both of these costs are given as per cent of the total cost. It can be seen that the transfer cost is always more than half of the total cost and that the cost of solving the 1D problems, including both arithmetic and transfers, is always in excess of 90% of the total cost.

7. COMPARISON OF RESULTS

The results of the analysis and simulations of sections 3 through 6 show quite clearly that the relative speed of the various algorithms depends very strongly on the architecture of the array computer, particularly the data transfer paths. The data transfer cost is minimum for the standard parallel Red-Black SOR for which it is only 40% of the total cost. It is always in excess of 50% for MG relaxation methods and the Buneman algorithm. Transfer cost is 2/3 to 3/4 of the cost of the solution of the larger problems for both the MGA and Buneman algorithm. If the transfer costs were ignored the cost of all of these algorithms would have been underestimated by a factor of 2 to 4. The overhead costs of the MGA are also substantial, in the range of 50% to 60% of the total cost.

The superiority of MG Red-Black SOR among the relaxation methods is also quite clear. It is more than twice as efficient as Single Weight Jacobi and more than five times as efficient as the standard Red-Black SOR. The next most efficient, or least costly, relaxation method, is the MG Five Weight Jacobi, but it still costs about 30% more than MG Red-Black SOR.

It is difficult to compare the cost of relaxation methods with direct methods because direct methods produce solutions which are exact (within round-off) in a finite number of steps which depend on the problem size, while the number of steps needed for convergence of any relaxation method depends not only on the problem size but on the convergence criterion. In order to make a comparison between direct and relaxation methods it will be assumed that the direct method cost can be compared to the cost of relaxing until the norm of the residuals is less than 10^{-6}. On this basis the MG Red-Black SOR is more efficient than the Buneman algorithm on this array computer. The Buneman algorithm costs almost two and one-half times as much as MG Red-Black SOR. As can be seen from Table VIII, the major portion of the cost of the direct method is incurred in solving the 1D problems at each stage of reduction and back-substitution, and most of that cost is due to data transfers. Bearing in mind that the value of 10^{-6} for the residual norm has been somewhat arbitrarily chosen as giving an accuracy equivalent to the direct method it seems clear that, on this array computer, the Buneman algorithm and the MG Red-Black SOR methods are competitive.

8. CONCLUSIONS

It can be concluded that, on this array computer; both the MG parallel relaxation methods and direct methods are much more efficient than standard parallel relaxation methods. The cost of solving the Poisson problem for the pressure field can be reduced by a factor of five by using one of these methods. The study of a fluid dynamic simulation[1,10] showed that about 80% of the total cost per time step was incurred in solving for the pressure field. The increase in efficiency of the Poisson solver makes possible a reduction by about a factor of three in the cost (time) per time step in a fluid dynamic simulation.

The transfer costs are always at least as important as arithmetic operation costs and for large problems, can be two or three times more than the arithmetic operation costs.

The overhead costs of applying the MGA are about half of the total costs.

I wish to thank Dr. A. Brandt for helpful discussions concerning the Multi-Grid Algorithm. I also wish to thank Dr. W. Lakin for a number of extensive discussions concerning this work and for reviewing the manuscript.

TABLE I

OPERATION COUNTS AND RELATIVE COSTS FOR VARIOUS GRID RELAXATION SCHEMES FOR 2D PROBLEMS

Operation	N_t	N_T	N_a	N_m	Cost
Jacobi Relaxation	0	4	4	2	4.8
Single Weight Jacobi Relaxation	0	4	5	3	5.9
Five Weight Jacobi Relaxation	0	8	9	4	9.7
Red-Black SOR Relaxation	0	4	6	2	5.0
Calculation of Residuals	0	4	6	4	7.0

TABLE II

RELATIVE DATA TRANSFER COST OF THE EXPANSION STRATEGY TO THE IN-PLACE STRATEGY, PER MG CYCLE FOR 2D AND 3D POISSON PROBLEMS ON THE ARRAY COMPUTER

$N_1 = N_2 = 2^M + 1$, $K = 2$

	2D, $L_0 = 4$	3D, $L_0 = 6$		
	L_{22}/L_1	L_{23}/L_1		
	$N_3 =$	32	64	128
M				
5	0.620	11.13	22.16	44.22
6	0.574	10.90	21.74	43.42
7	0.544	10.79	21.54	43.05

TABLE III

OPERATION COUNTS AND COSTS FOR VARIOUS OVERHEAD OPERATIONS USED IN APPLYING THE MGA

Operation	N_t	N_T	N_a	N_m	Cost
Compression (Level M $\rightarrow$ M-1)	1	2^M	0	0	0.6×2^M
Expansion (Level M-1 $\rightarrow$ M)	1	2^M	0	0	0.6×2^M
Injection	1	0	0	1	1.0
Weighted Injection	1	8	8	2	7.6
Linear Interpolation	4	6	4	1	5.1

TABLE IV

ERROR REDUCTION EFFICIENCY PER UNIT WORK
THERE ARE K ITERATIONS PER MG CYCLE ON A $2^M + 1$ BY $2^M + 1$ GRID

$\lambda \times 10^3$

Single Weight Jacobi ($\bar{\mu} = 0.60$)

M \ K	1	2	3	4
5	2.00	2.11	1.93	1.71
6	1.33	1.52	1.45	1.31
7	0.81	0.99	0.98	0.91

Five Weight Jacobi ($\bar{\mu} = 0.22$)

M \ K	1	2	3	4
5	3.39	2.62	1.99	1.58
6	2.25	1.85	1.45	1.18
7	1.45	1.29	1.05	0.87

Red-Black SOR ($\bar{\mu} = 0.50$)

M \ K	1	2	3	4
5	2.29	2.38	2.13	1.85
6	1.50	1.66	1.53	1.35
7	0.96	1.13	1.08	0.98

TABLE V

RESULTS OF SIMULATIONS OF SEVERAL MG
RELAXATION METHODS FOR PROBLEMS 1 AND 2
THE METHODS, PROBLEMS, AND SYMBOLS ARE EXPLAINED IN THE TEXT

Problem	M	$\tilde{\mu}$		$\lambda \times 10^3$	
		Theoretical	Observed	Theoretical	Observed
		Single Weight Jacobi ($\omega_0 = 1.0$)			
1	5	0.640	0.653	2.11	2.60
	6		0.691	1.52	1.85
	7		0.604	0.99	1.06
2	5		0.642	2.11	2.56
	6		0.631	1.52	1.69
	7		0.617	0.99	1.08
		Five Weight Jacobi ($\omega_1 = 48/41$, $\omega_2 = 8/41$)			
1	5	0.952	0.829	2.62	2.79
	6		0.821	1.85	1.91
	7		0.820	1.29	1.28
2	5		0.918	2.62	3.09
	6		0.903	1.85	2.10
	7		0.894	1.29	1.40
		Red-Black SOR ($\omega = 1.0$)			
1	5	0.750	0.919	2.38	3.04
	6		0.919	1.66	2.09
	7		0.920	1.13	1.41
2	5		0.952	2.38	3.21
	6		0.947	1.66	2.34
	7		0.954	1.13	1.54

TABLE VI

OVERALL PERFORMANCE OF A STANDARD PARALLEL RELAXATION METHOD AND MG RELAXATION METHODS FOR PROBLEM 1 AND 2. THE PROBLEMS AND SYMBOLS ARE EXPLAINED IN THE TEXT. THE UNIT OF COST IS ONE MULTIPLICATION TIME.

Problem	M	Transfers	Relaxations	Overhead Cost %	Transfer Cost %	Total Cost
		7-6-5-4-3-2				
		MG Single Weight Jacobi ($\omega_0 = 1.0$)				
		$\varepsilon = 10^{-2}$				
1	5	0 0 6 6 5 5	69	38	57	1441.9
	6	0 7 7 7 6 6	102	48	64	2544.3
	7	7 7 7 7 7 6	130	58	71	4039.1
		$\varepsilon = 10^{-4}$				
2	5	0 0 7 7 7 6	84	38	57	1747.5
	6	0 8 8 8 8 7	119	50	67	2851.5
	7	8 8 8 8 8 4	137	62	75	4312.4
		$\varepsilon = 10^{-6}$				
2	7	13 13 13 13 13 7	226	58	76	6860.5
		MG Five Weight Jacobi ($\omega_1 = 48/41$, $\omega_2 = 6/41$)				
		$\varepsilon = 10^{-2}$				
1	5	0 0 4 4 4 3	48	32	58	1176.3
	6	0 5 5 5 5 3	72	42	64	2077.1
	7	5 5 5 5 5 3	87	53	71	3126.1

TABLE VI CONT.

Problem	M	Transfers	Relaxations	Overhead Cost %	Transfer Cost %	Total Cost
			$\varepsilon = 10^{-4}$			
2	5	0 0 4 3 3 0	42	30	58	1002.4
	6	0 4 3 3 2 1	47	43	65	1368.2
	7	4 3 3 3 2 1	56	55	72	2074.4
			$\varepsilon = 10^{-6}$			
2	7	7 5 5 5 4 2	96	55	72	3569.2
			MG Red-Black SOR ($\omega = 1.0$)			
			$\varepsilon = 10^{-2}$			
1	5	0 0 3 3 3 2	44	40	59	878.9
	6	0 3 3 3 3 2	56	48	64	1291.4
	7	3 3 3 3 3 2	68	58	71	1934.3
			$\varepsilon = 10^{-4}$			
2	5	0 0 3 3 3 3	42	43	60	879.6
	6	0 3 3 3 2 1	49	50	65	1167.6
	7	3 3 3 3 2 1	60	60	72	1798.5
			$\varepsilon = 10^{-6}$			
2	7	5 5 5 5 4 3	106	54	80	2790.2
			Red-Black SOR ($\omega = 1.85$)			
			$\varepsilon = 10^{-2}$			
1	5		59	0	40	708.0
	6		267	0	40	3204.0
	7		1132	0	40	13584.0

TABLE VI CONT.

Problem	M	Transfers	Relaxations	Overhead Cost %	Transfer Cost %	Total Cost
			$\varepsilon = 10^{-4}$			
2	5		57	0	40	684.0
	6		185	0	40	2220.0
	7		674	0	40	8088.0
			$\varepsilon = 10^{-6}$			
2	7		1282	0	40	15384.0

TABLE VII

OPERATION COUNTS AND TIME COST FOR THE COMPONENTS OF THE DIRECT SOLUTION ALGORITHM FOR THE POISSON EQUATION

Operation	N_t	N_T	N_a	N_m	Cost
Compression & Expansion of One Vector[a]	0	$2(2^s-1)$	0	0	$1.2(2^s-1)$
Motrix Multiplication[a]	$(2^{s+1})^2$	2^{s+1}	2^{s+1}	$1+2^{s+1}$	$1+2^{s+1}(1.70 + 0.02\cdot2^{s+1})$
1D Buneman Algorithm[a]	$2^r(4+14s)$	$2^r(6s)$	$2^r(5+10s)$	$2^r(2+2s)$	$2^r(2.58+6.88s)$
Compression & Expansion of One Set of Vectors[b]	0	$2(2^k-1)$	0	0	$1.2(2^k-1)$
2D Buneman Algorithm[b]	$6+3k$	$8+6k$	$16+8k$	0	$6.52+4.46k$

[a]The 1D problem has a $(2^{s+1}+1)$ by $(2^{s+1}+1)$ matrix and $(2^{s+1}+1)$ element vectors. The 1D Buneman algorithm also requires compression and expansion of two vectors and solution of 2^r tridiagonal systems.

[b]The 2D problem has $(2^{s+1}+1)$ $(2^{k+1}+1)$ unknowns. The 2D Buneman algorithm also requires compression and expansion of two sets of vectors and the solution of two 1D problems at the levels $r = 0,1,2...,k$.

TABLE VIII

TOTAL COST OF SOLVING A 2D POISSON PROBLEM
USING THE BUNEMAN ALGORITHM, WITH
$k+1 = s+1 = M$

M	Cost of 1D Solutions (% of Total)	Transfer Cost (% of Total)	Total Cost
5	92.5	55.0	799.8
6	95.4	63.0	2243.0
7	97.6	68.5	6808.1

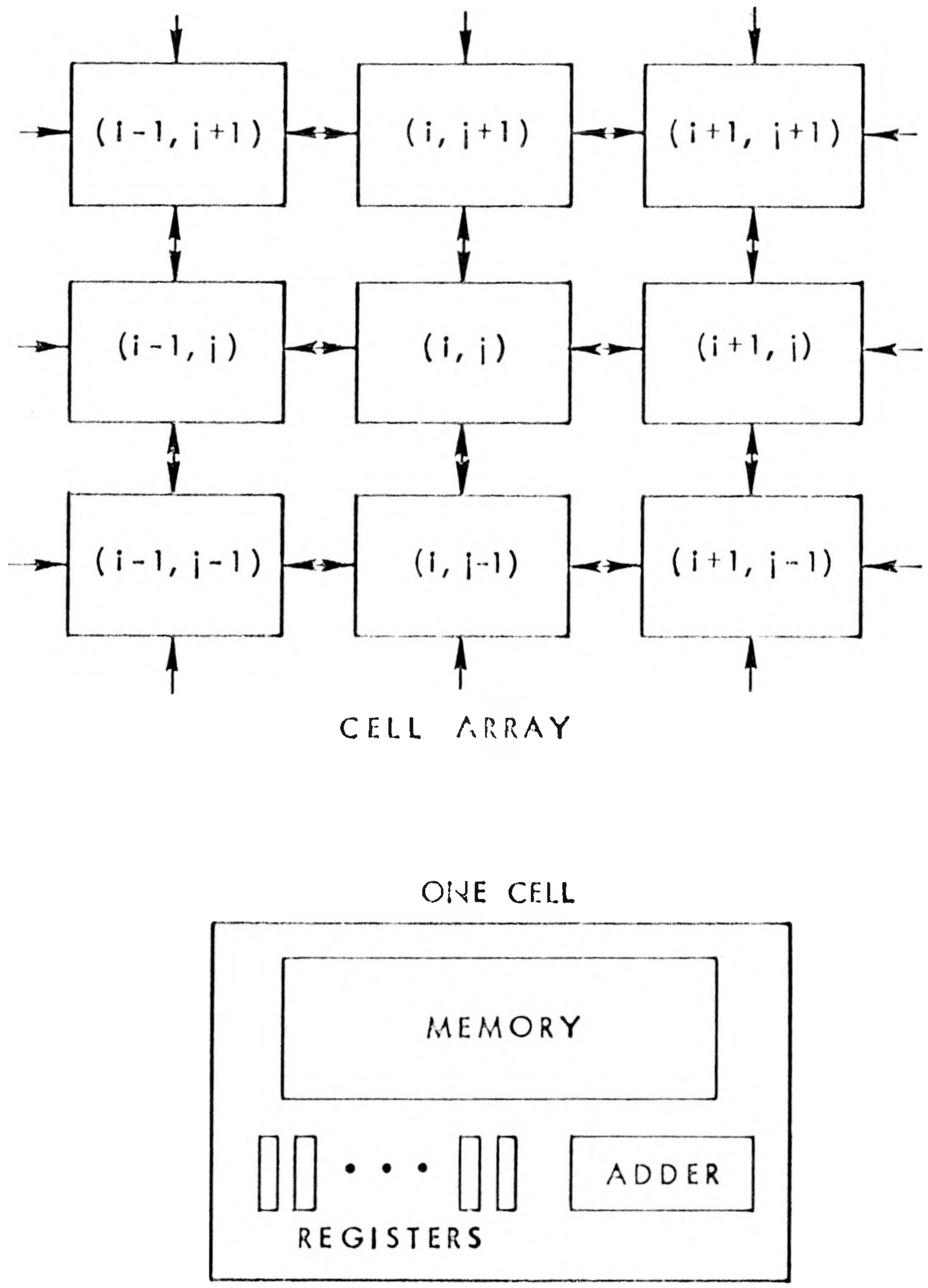


Figure 1. Schematic representation of the cell-array computer and a typical Processing Cell.

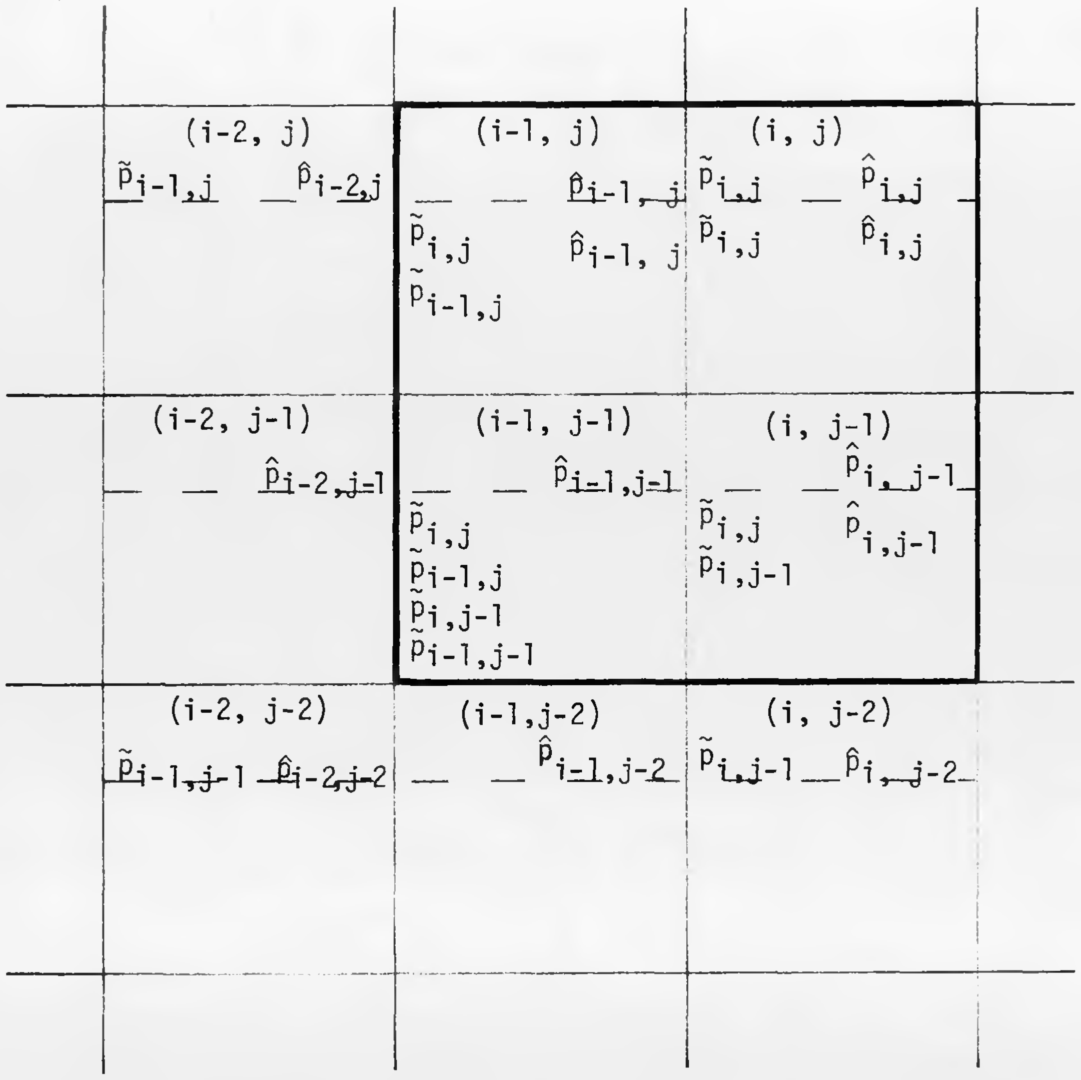


Figure 2. A section of the array. Each block represents one PC, which is labeled with the appropriate indices. The heavy lines deliminate the primitive correction block; all fine grid solutions in this block are corrected by the coarse grid solution and the correction algorithm is periodic, with period 2 in both the i and j indices.

REFERENCES

1. E. C. Gritton, W. S. King, I. Sutherland, R. S. Gaines, C. Gayley, Jr., C. Grosch, M. Juncosa, H. Peterson, "Feasibility of a Special-Purpose Computer To Solve the Navier-Stokes Equations," Rand Corp. report R-2183-RC (June 1977).

2. S. A. Orszag and M. Israeli, "Numerical Simulations of Viscous Incompressible Flows," Ann. Rev. Fluid Dyn., M. Van Dyke, W. G. Vincenti, and J. V. Wehausen, Ed. 6, 281-318 (1974).

3. S. A. Orszag and G. S. Patterson, Jr., "Numerical Simulation of Turbulence" Lecture Notes in Physics, Vol. 12. M. Rosenblatt and C. Van Atta, Eds. (Springer-Verlog, Berlin, 1972) pp. 127-147.

4. R. W. Hockney, "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis," J. Assoc. Comput. March 12, 95-113 (1965).

5. R. W. Hockney, "The Potential Calculation and Some Applications" Methods in Computational Physics, Vol. 9, B. Adler, S. Fernbach and M. Rotenberg, Eds. (Academic Press, New York, 1969) pp. 136-211.

6. B. L. Buzbee, G. H. Golub and C. W. Nielson, "On Direct Methods For Solving Poisson's Equation," SIAM J. Numer. Anal. 7, 627-656, (1970).

7. A. Brandt, "Multi-Level Adaptive Solutions to Boundary-Value Problems" Math. Comput., 31, 333-390 (1977).

8. R. A. Nicolaides, "On Multiple Grid and Related Techniques for Solving Discrete Elliptic Systems," J. Comput. Phys. 19, 418-431 (1975).

9. K. M. Case, F. J. Dyson, E. A. Frieman, C. E. Grosch, and F. W. Perkins, "Numerical Simulation of Turbulence," Stanford Res. Inst. Tech. Rept. JSR-73-3 (Nov. 1973).

10. C. F. R. Weiman and C. E. Grosch, "Parallel Processing Research in Computer Science: Relevance to the Design of a Navier-Stokes Computer," Proc. 1977 Int'l. Conf. Parallel Processing. (In press).

11. W. L. Miranker, "A Survey of Parallelism in Numerical Analysis," SIAM Rev. 13, 524-547 (1971).

12. J. F. Traub (Ed.)"Proceedings of Conference on Complexity of Sequential and Parallel Numerical Algorithms," (Academic Press, New York, 1973).

13. H. S. Stone, "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations," J. Assoc. Comput. Mach. 20, 27-38 (1973).

14. H. S. Stone, "Parallel Tridiagonal Equation Solvers," ACM Trans. Math. Software, 1, 289-307 (1975).

15. A. H. Sameh, S. C. Chen, and D. J. Kuck, "Parallel Poisson and Biharmonic Solvers," Computing, 17, 219-230 (1976).

16. J. M. Ortega and R. G. Voight, "Solution of Partial Differential Equations on Vector Computers," Inst. Computer Applications in Science and Engineering (ICASE) Rept. 77-7, NASA Langley Research Center (March, 1977).

17. W. F. Ames, "Numerical Methods for Partial Differential Equations," 2ed Ed. (Academic Press, New York, 1977), pp. 103-106.

18. P. N. Swarztrauber, "The Methods of Cyclic Reductions, Fourier Analysis and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle," SIAM Rev., 19, 490-501 (1977).

VECTORIZED FORTRAN SUBPROGRAMS FOR THE SYMMETRIC FAST FOURIER TRANSFORMS

by

Paul N. Swarztrauber
National Center for Atmospheric Research
Boulder, Colorado 80307

ABSTRACT

In addition to the usual real and complex periodic transforms, the package contains programs for fast sine, cosine and quarter wave transforms. The package has been used to solve constant coefficient elliptic partial differential equations using the Fourier method where the symmetric transforms are used in the event the boundary conditions are not periodic. For example, if the solution is specified on a boundary and its derivative is specified on the opposite boundary, then a quarter wave transform can be used and the application of the Fourier method is not restricted to problems with periodic boundary conditions. The symmetric transforms are performed by preprocessing the data into a complex periodic sequence. The coefficients for the symmetric transform can then be obtained by post processing the complex transform. The programs are written in standard FORTRAN in such a way that they are vectorized by the CRAY compiler. Sorting is not explicitly required in the complex transform which uses a data structure in which sorting is done implicitly. Timing comparisons are presented and several aspects of the implementation of the FFT on the CRAY-1 are discussed.

A PARALLEL ALGORITHM FOR SOLVING BONDED LINEAR SYSTEMS ARISING FROM SECOND ORDER PARTIAL DIFFERENTIAL EQUATIONS IN TWO DIMENSIONS

by

V. Ellel
D. Parkinson
International Computers Ltd.
London, England

ABSTRACT

A new iterative method is proposed for dealing with sparse linear systems typically arising from finite difference representations of second order partial differential equations in two dimensions. The time for a single iteration is proportional to $\log_2 N$ where N is the largest number of mesh points in either of the two dimensions.

The results of some experiments using the method on Poissons equation will be represented - although the method is quite general and can be applied to more general equations on non uniform rectangular meshes.

FINITE ELEMENT DYNAMIC ANALYSIS ON THE STAR-100

by

Jules J. Lambiotte, Jr.
NASA Langley Research Center
Hampton, Virginia 23665

ABSTRACT

Work is in progress at Langley Research Center to develop a model finite element code for the dynamic analysis of structures. Higher-order finite elements are used to describe the spatial behavior. The temporal behavior is approximated by using either the central difference explicit scheme or Newmark's implicit scheme. The total computational algorithm is broken up into a number of basic macro-operations. This paper will discuss the data organization and vectorization procedures used to implement these macro-operations on STAR. In particular, the procedures to generate the element stiffness and mass matrices, to decompose the global matrices, and to multiply the global stiffness matrix by a vector will be discussed and sample times given.

SOME VARIANTS OF METHODS FOR COMPUTING THE VARIANCE

by

T. Chan, G. H. Golub, R. LeVeque
Stanford University
Department of Computer Science
Stanford, California 94305

ABSTRACT

We consider the problem of computing the sample variance in a numerically stable manner. We describe several algorithms which can be implemented on a variety of computer architectures and give an estimate of the error. The results of numerical simulations will be presented.

A DIRECT POISSON SOLVER ON STAR

by

M. J. Kascic, Jr.*
Control Data Corporation
Arden Hills, MN 55112

1. Introduction:

The study of three dimensional turbulent channel flow with planar homogeneity involves the solution of the Poisson Equation

$$\nabla^2 f = g$$

on a rectangular parallelopiped

$$\left\{\begin{array}{ccccc} 0 & \leq & x & \leq & A_1 \\ 0 & \leq & y & \leq & A_2 \\ 0 & \leq & z & \leq & A_3 \end{array}\right\}$$

with the following types of mixed boundary conditions:

I.

$$\begin{aligned} f\{0,y,z\} &= f\{A_1,y,z\} \\ f\{x,0,z\} &= f\{x,A_2,z\} \\ f\{x,y,0\} &= f_0\{x,y\} \\ f\{x,y,A_3\} &= f_1\{x,y\} \end{aligned}$$

i.e. periodic in x and y and Dirichlet in z

and

II.

$$\begin{aligned} f\{0,y,z\} &= f\{A_1,y,z\} \\ f\{x,0,z\} &= f\{x,A_2,z\} \\ \frac{\partial f\{x,y,A_3\}}{\partial z} &= 0 = \frac{\partial f\{x,y,0\}}{\partial z} \end{aligned}$$

i.e. periodic in x and y and Neumann in z.

*This work was supported by a CDC Sponsored Research Award for "Research in Numerical Fluid Mechanics Using the CDC STAR Computer".

In this report we shall discuss the solution of the discretized Poisson Equation with boundary conditions I, using a cyclic reduction algorithm on the CDC STAR 100 and 100A computers. It is planned to study a gamut of Poisson solvers on STAR for such an application.

2. STAR Architecture

The interested reader is invited to consult the relevant literature {2}, {3}, {4}, {5} for details of the STAR architecture. We shall recall highlights.

The major components of the STAR we will consider are the memory, the stream unit and the floating point unit. We shall compare and contrast these for the STAR 100 and 100A.

Memory: The STAR 100 half million word machine utilizes a 1966 core memory that consists of 32 phased banks containing 16K words each. On the one hand the cycle time for an individual bank is 1.28 μsec. which implies that a random load/store busies a bank for 31 cycles. On the other hand, when streaming, i.e. when linear access is made to memory, four independent streams can be sustained to/from memory at a rate of 2 64 bit words/40 nsec. each. This implies a total bandwidth of 12.8 gigabits, a high rate even by today's standards.

It is quite obvious that efficient usage of the STAR 100 memory demands a "linear" or "vector" access mode.

The STAR-100A million word memory, by contrast, is LSI and consists of 128 phased banks containing 8K words each. Both access and cycle-time for this memory are 80 nsec.

Stream: For our purposes we shall only need to know that the stream unit contains the register file and the control with ancillary buffers to allow the memory {in either case} to perform as indicated above.

On the STAR 100, the register file consists of 256 programmable 64 bit registers with one read from and write to per 40 nsec. in scalar mode. On the STAR 100A, the register file also consists of 256 programmable 64 bit registers but two reads and one write may take place each 20 nsec. in scalar mode.

The STAR 100 has two general purpose segmented functional units that perform both scalar and vector floating point operations. When in vector mode, these units are kept busy processing operands which are streamed in and out of memory using the high bandwidth stream rate. Thus the effective megaflop rate is independent of the pipe length. The time to do a vector add/multiply of length N is

{2.1} $$2840 + 20N \;/\; 6360 + 40N \text{ nsec.}$$

By definition, a vector on STAR consists of contiguous elements.

In scalar mode the effective time to do an arithmetic operation depends on many factors such as 1. whether operands are in the register file 2. how many independent operations are to be done 3. the existence or lack thereof of register file conflicts, etc. We will illustrate these points later.

Naturally on the STAR 100, vector and scalar arithmetic cannot be done at the same time.

The STAR 100A has an independent scalar box with special purpose segmented units for add and multiply (as well as others we need not mention). Because of this independent unit, the STAR 100A can issue a vector instruction and go on to process scalar instructions that do not reference memory.

We will bring out other features of the architecture as they become important to the discussion of the problem at hand.

3. Discretized Poisson Equation

We shall solve the discretized Poisson equation on the rectangular parallelopiped:

$$\left\{\begin{array}{ccccc} 0 & \le & x & \le & A_1 \\ 0 & \le & y & \le & A_2 \\ 0 & \le & z & \le & A_3 \end{array}\right\}$$

The discretization consists in imposing a mesh of points with

$$\text{coordinates}\quad \left\{ (l-1)\Delta,\ (k-1)\Delta,\ j\Delta \ni\ l=1,L_1\ ,\ k=1,K_1\ ,\ j=0,J_1 \right.$$

$$\left.\text{with}\qquad \Delta = \frac{A_1}{L_1} = \frac{A_2}{K_1} = \frac{A_3}{J_1}\right\}$$

A typical point will be written (l,k,j) and any quantity f defined at that point will be written $f_{l,k,j}$

We now use a standard second order central difference approximation for the partial derivatives occurring in the equation, viz.

$$\left.\frac{\partial^2 f}{\partial x^2}\right)_{l,k,j} \sim \frac{f_{l-1,k,j} - 2 f_{l,k,j} + f_{l+1,k,j}}{\Delta^2} ,$$

the y and z directions treated mutatis mutandis.

Hence the Poisson equation is replaced by a set of simultaneous linear equations with a typical one having the form:

$$f_{l-1,k,j} + f_{l,k-1,j} + f_{l,k,j-1} - 6f_{l,k,j} + f_{l,k,j+1} + f_{l,k+1,j} + f_{l+1,k,j} = \Delta^2 g_{l,k,j}$$

Since our boundary conditions are periodic in x and y and Dirichlet in z we have

$$\left.\begin{aligned} f_{l,k,0} &= f_0(l\Delta, k\Delta) \\ f_{l,k,J_1} &= f_1(l\Delta, k\Delta) \end{aligned}\right\} \qquad \begin{aligned} l&=1,L_1 \\ k&=1,K_1 \end{aligned}$$

$$f_{L+1,k,j} = f_{0,k,j} \qquad k=1,K_1 \qquad j=1,J_1-1$$

$$f_{l,K+1,j} = f_{l,0,j} \qquad l=1,L_1 \qquad j=1,J_1-1 .$$

Hence the unknowns consist of the set

$$\left\{ f_{l,k,j} \ni l=1,L_1 \ , k=1,K_1 \ , j=1,J_1-1 \right\} .$$

In this paper we will fix our attention on the case

$J_1=64 , L_1=K_1=32.$

If we order the unknowns as follows:

$f_{1,1,1} \cdots f_{1,1,63}$, $f_{1,2,1} \cdots f_{1,2,63} \cdots f_{1,32,1} \cdots f_{1,32,63}$

$\cdots f_{32,1,1} \cdots f_{32,1,63} \cdots f_{32,32,1} \cdots f_{32,32,63}$,

the discretized Poisson equation becomes 32x32x63 = 64512 linear equations is 64512 unknowns which has the following block periodic tridiagonal form:

{3.2}
$$\begin{bmatrix} B & I & & & I \\ I & B & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & B & I \\ I & & & I & B \end{bmatrix}$$

with block dimension $L_1 = 32$ and each B has the same block periodic tridiagonal form:

{3.3}
$$\begin{bmatrix} A & I & & & I \\ I & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ I & & & I & A \end{bmatrix}$$

with block dimension K_1 = 32 and each A is 63x63 tridiagonal of the form:

$$\begin{bmatrix} -6 & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -6 \end{bmatrix} \tag{3.4}$$

In an attempt to set up a clear picture of this block structure, let us illustrate for $L_1 = K_1 = 4$

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
A & I & 0 & I & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 \\
I & A & I & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 \\
0 & I & A & I & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 \\
I & 0 & I & A & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I \\ \hline
I & 0 & 0 & 0 & A & I & 0 & I & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & I & 0 & 0 & I & A & I & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & I & A & I & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & I & I & 0 & I & A & 0 & 0 & 0 & I & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & I & 0 & 0 & 0 & A & I & 0 & I & I & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & I & 0 & 0 & I & A & I & 0 & 0 & I & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & I & A & I & 0 & 0 & I & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & I & I & 0 & I & A & 0 & 0 & 0 & I \\ \hline
I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & 0 & A & I & 0 & I \\
0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & I & A & I & 0 \\
0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & 0 & 0 & I & A & I \\
0 & 0 & 0 & I & 0 & 0 & 0 & 0 & 0 & 0 & 0 & I & I & 0 & I & A
\end{array}\right] \tag{3.5}$$

4. Block Cyclic Reduction

We shall now discuss the algorithm used to solve the blocked form of the discretized Poisson equation, Bunemann variant I of block cyclic reduction. A complete discussion of this algorithm including a proof of numerical stability can be found in {1}. We shall recall some necessary details using the notation of {1}.

Henceforth $L_1 = 2^{L+1}$ and $K_1 = 2^{K+1}$.

Consider three consecutive block equations using the block level of {3.2}

{4.1} $$\begin{aligned} x_{2n-2} + B\,x_{2n-1} + x_{2n} \qquad\qquad &= y_{2n-1} \\ x_{2n-1} + B\,x_{2n} + x_{2n+1} \qquad &= y_{2n} \\ x_{2n} + B\,x_{2n+1} + x_{2n+2} &= y_{2n+1} \end{aligned}$$

If we multiply the middle {even} equation by {-B} and add all three equations together, we get

{4.2} $$x_{2n-2} + \{2I-B^2\}\,x_{2n} + x_{2n+2} = y_{2n-1} - By_{2n} + y_{2n+1}$$

If we carry out this process for all the even equations, {4.2} becomes another block periodic tridiagonal system

{4.3} $$x_{2n-2} + B^{\{1\}} x_{2n} + x_{2n+1} = y_{2n}^{\{1\}}$$

in the even block variables with

{4.4} $$B^{\{1\}} = 2I - B^2 \qquad \text{and} \qquad y_{2n}^{\{1\}} = y_{2n-1} - By_{2n} + y_{2n+1}$$

Concomitantly, the odd equations taken together yield a block diagonal system that one can solve for the odd block variables given that the even variables are known:

$$B\, x_{2n-1} = y_{2n-1} - \{x_{2n-2} + x_{2n}\} \qquad \{4.5\}$$

This reduction phase can be repeated again and again until a 2 x 2 block system remains:

In carrying out the various stages of the cyclic reduction one generates at the {r+1}st stage:

$$B^{\{r+1\}} = 2I - B^{\{r\}2} \qquad \{4.6\}$$
$$y_n^{\{r+1\}} = \{y_{n-2^r}^{\{r\}} + y_{n+2^r}^{\{r\}}\} - B^{\{r\}}\, y_n^{\{r\}}$$

One can show { 1 }, that

$$B^{\{r\}} = -\prod_{j=1}^{2^r} \{ B + 2\cos \theta_j^r\, I \} \quad r \leq L+1 \qquad \{4.7\}$$

$$\text{where } \theta_j^r = \frac{\{2j-1\}\pi}{2^{r+1}} \qquad r \leq L$$

$$\text{and } \theta_j^{L+1} = \frac{j\pi}{2^L}$$

Direct usage of {4.6} to evaluate the right hand side for progressive stages of the reduction, however, is numerically unstable. The instability can be eliminated however, by the following method due to Bunemann.
Define $p_n^{\{r\}}$, $q_n^{\{r\}}$ such that

$$B^{\{r\}}\, p_n^{\{r\}} + q_n^{\{r\}} = y_n^{\{r\}} \qquad \{4.8\}$$

Thus at each stage of reduction, one has

{4.9} $$B^{\{r\}}x_n = B^{\{r\}}p_n^{\{r\}} + q_n^{\{r\}} - \{x_{n-2^r} + x_{n+2^r}\}$$

or $$B^{\{r\}}\{x_n - p_n^{\{r\}}\} = q_n^{\{r\}} - \{x_{n-2^r} + x_{n+2^r}\}$$

It is shown in { 1 } that the norms of the $q_n^{\{r\}}$ remain bounded and that the $p_n^{\{r\}}$ become "good" approximations to x_n.

5. Implementation

The algorithm described in section 4 is implemented on STAR to solve the problem discussed in section 3 as follows.

There are three levels of calculations:

Level 1 embodies cyclic reduction at the level of {3.2}.
Level 2 embodies cyclic reduction at the level of {3.3}.
Level 3 embodies the LU decomposed tridiagonal substitution at the level 3.4.

Levels 1 and 2 are written in STAR Fortran using explicit descriptors { 5 } to access STAR's vector hardware.

Level 3 is written in META {the assembly language for STAR}. The reason for this will be clear when we discuss this level.

Because STAR vector performance gets more efficient with longer vectors {allowing more operations to amortize startup}, besides the actual arithmetic, at levels 1 and 2, there will be data motion to rearrange elements into proper long vectors.

Let us outline the database requirements. The RHS of {3.2} resides in an array 65536 {= 32x32x64} words long. The 64512 {= 32x32x63} values are distributed according to the scheme outlined in section three for the f values with each 64th word blank. Thus, the two block sizes we shall work with are, 64 at level 2 and 64 x 2^{K+1} {=2048} at level 1.

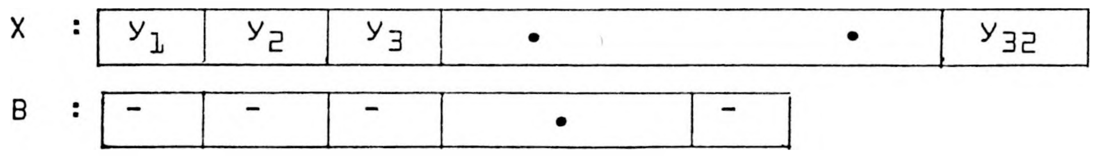
As the solution progresses, the p values will overwrite the RHS values, so that when we are done, the solution will fill the original array, a not uncommon practice to cut down memory utilization.

Notationally, we shall use singly subscripted variables, X_j, to refer to blocks of length 2048 at level one and doubly subscripted variables X_{jk} to refer to blocks of length 64 at level two, which will be subblocks of X_j.

An ancillary array B, of length 3/4 x 2^{K+1} x 2^{L+1} x 64 {= 49152} words will be used to carry the Bunemann $P_j^{\{r\}}$ vectors along. {The $q_j^{\{r\}}$ will reside as intermediate results in the X array.}

Level 1

Upon entering level 1, the following situation exists:



The level 1 blocks of the X {B array resp.} will be denoted by X_j {B_j resp}. Thus the above tableau has

$$X_j = y_j \qquad\qquad j = 1,32$$

Since the first step is to calculate $p_{2j}^{\{1\}}$ such that $Bp_{2j}^{\{1\}} = y_{2j}$ $j = 1,16$, we rearrange the X array sorting out even and odd blocks, viz.

X:								
	y_1	y_3	•	y_{31}	y_2	y_4	•	y_{32}

This is accomplished by 38 vector moves of length 2048.

Then level 2 is called to solve

{5.2} $$Bp_{2j}^{\{1\}} = y_{2j} \qquad j = 1,16$$

The $p_{2j}^{\{1\}}$'s overwrite the y_{2j}. Hence at this point:

X:							
	y_1	y_3	•	y_{31}	$p_2^{\{1\}}$	•	$p_{32}^{\{1\}}$

The formulae to calculate $q_{2j}^{\{1\}}$ are given in { 1 } as

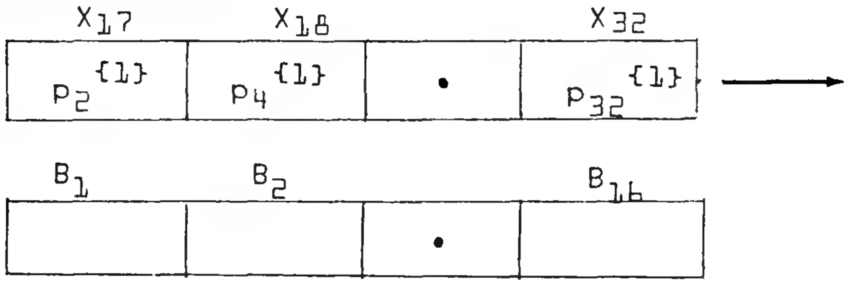
{5.3} $$\begin{aligned} q_2^{\{1\}} &= y_1 + y_3 - 2p_2^{\{1\}} \\ q_{2j}^{\{1\}} &= y_{2j-1} + y_{2j+1} - 2p_{2j}^{\{1\}} \qquad j = 2,15 \\ q_{32}^{\{1\}} &= y_{32} + y_1 - 2p_{32}^{\{1\}} \end{aligned}$$

The calculation of {5.3} involves 48 x 2048 {= 98304} arithmetic operations. Let us carefully outline how this particular kernel is calculated on STAR since comparable situations arise repeatedly in the solver.

A glancing acquaintance with STAR will yield the fact that {5.3} could be done in 3 x 16 {= 48} vector operations. However, {2.1} makes clear that for a fixed amount of arithmetic, we should strive to minimize the number of vector instructions to minimize startup time.

In the present case, the data is so arranged that five vector instructions accomplish {5.3}.

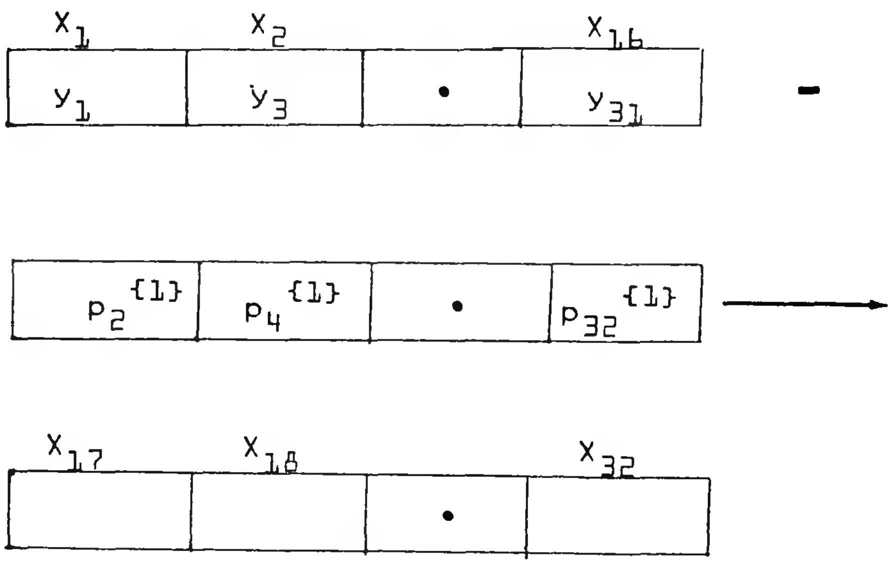
Instr. 1: Move



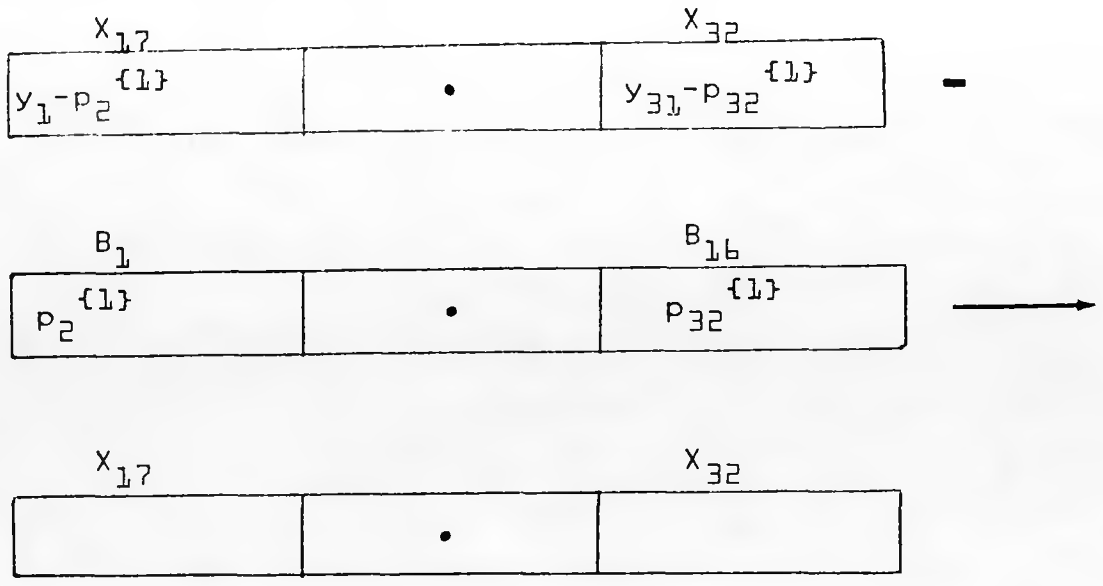
This is one vector instruction since all the blocks to be moved and the targets to be filled are contiguous.

Also note that X_{j+16} j=1,16 are now free.

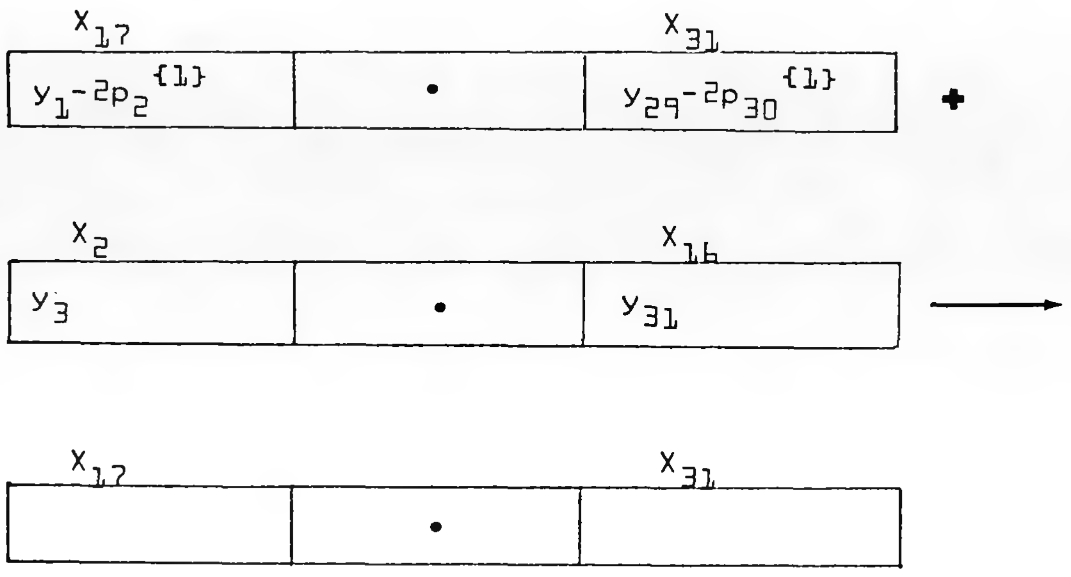
Instr. 2: Subtract



Instr. 3: Subtract



Instr. 4: Add



Instr. 5: Add

$$\overset{X_{32}}{\boxed{y_{31}-2p_{32}^{\{1\}}}} + \overset{X_{1}}{\boxed{y_{1}}} \longrightarrow \overset{X_{32}}{\boxed{\phantom{y_{31}-2p_{32}}}}$$

At this point we have

X_1	X_2		X_{16}	X_{17}		X_{32}
y_1	y_3	•	y_{31}	$q_2^{\{1\}}$	•	$q_{32}^{\{1\}}$

B_1	B_2		B_{16}	
$p_2^{\{1\}}$	$p_4^{\{1\}}$	•	$p_{32}^{\{1\}}$	•

A block sort of $X_{17} \cdots X_{32}$ and $B_1 \cdots B_{16}$ yields:

X_1	X_2		X_{16}	X_{17}	X_{18}		X_{24}	X_{25}		X_{32}
y_1	y_3	•	y_{31}	$q_2^{\{1\}}$	$q_6^{\{1\}}$	•	$q_{30}^{\{1\}}$	$q_4^{\{1\}}$	•	$q_{32}^{\{1\}}$

B_1	B_2		B_8	B_9		B_{16}
$p_2^{\{1\}}$	$p_6^{\{1\}}$	•	$p_{30}^{\{1\}}$	$p_4^{\{1\}}$	•	$p_{32}^{\{1\}}$

This completes the first stage of level 1. Effectively we have uncoupled variables

$X_2, X_4, \ldots, X_{32}$ from variables $X_1, X_3, \ldots, X_{31}$.

Basically the same process is repeated for stages two through five. One particular difference is that the RHS of {4.9} needs to be calculated before the call to level 2.

The steps are:

{5.4} {i} Two or three vector instructions to calculate the RHS
{ii} Call to level 2 to solve {4.9}
{iii} Four or five vector instructions to calculate $q_j^{\{r\}}$
{iv} Two block sorts to rearrange data.

At the conclusion of the fifth reduction stage of level 1 we have

X_1	X_2		X_{16}	X_{17}		X_{24}	X_{25}		X_{28}	X_{29}	X_{30}	X_{31}	X_{32}
y_1	y_3	•	y_{31}	$q_2^{\{1\}}$	•	$q_{30}^{\{1\}}$	$q_4^{\{2\}}$	•	$q_{28}^{\{2\}}$	$q_8^{\{3\}}$	$q_{24}^{\{3\}}$	$q_{16}^{\{4\}}$	$q_{16}^{\{5\}}$

B_1	B_2		B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	B_{16}
$p_2^{\{1\}}$	$p_6^{\{1\}}$	•	$p_{30}^{\{1\}}$	$p_4^{\{2\}}$	$p_{12}^{\{2\}}$	$p_{20}^{\{2\}}$	$p_{28}^{\{2\}}$	$p_8^{\{3\}}$	$p_{24}^{\{3\}}$	$p_{16}^{\{4\}}$	$p_{16}^{\{5\}}$

Now we begin to backsolve, starting with

$$B^{\{5\}} \{X_{16} - p_{16}^{\{5\}}\} = q_{16}^{\{5\}}$$

The outline for the successive substitutions is very similar to {5.4} except that we block merge at step {iv}.

To complete our discussion of level 1, we shall present an analysis of the execution time.

Except for negligible scalar overhead, level 1 consists of vector arithmetic and data motion. The vector arithmetic performs 475136 floating point operations in 9.66 msec. for a megaflop rate of 49.2. However, to be honest one must include the data motion execution time since it is an integral part of getting the vector performance. This raises the execution time to 17.58 msec. for an effective megaflop rate of 27.0.

Level 2

At various stages of level 1, it is necessary to solve equations of the form

$$B^{\{r\}}\{p_j^{\{r+1\}} - p_j^{\{r\}}\} = q_j^{\{r\}} - \{p_{j-2^r}^{\{r\}} + p_{j+2^r}^{\{r\}}\}$$

Each matrix $B + 2\cos\theta_j^{\{r\}}I$ in the decomposition of $B^{\{r\}}$ is block periodic tridiagonal of the form {3.3} except that the diagonal value will vary.

Hence level 2 must solve a recursive sequence of block periodic tridiagonal systems of the type solved in level 1.

A given call to level two demands 2^r recursive block tridiagonal solution with 2^s RHS's.

Call the RHS's Z_1, Z_2, ... Z_{2^s}. Note that each Z_j itself has a block structure; hence we have

{5.5} $$\underbrace{Z_{11}\ Z_{12}\ \cdots\ Z_{1\,32}}_{Z_1}\ \cdots\ \underbrace{Z_{2^s1}\ \ Z_{2^s2}\ \cdots\ Z_{2^s32}}_{Z_{2^s}}$$

Upon entering level 2 we first block transpose {5.5} allowing us to use a simultaneous solution approach to the 2^s RHS's. At this point we have

$$\underbrace{Z_{11}\ Z_{21}\ \cdots\ Z_{2^s1}}\ \underbrace{Z_{12}\ \cdots\ Z_{2^s2}}\ \cdots\ \underbrace{Z_{1\,32}\cdots\ Z_{2^s32}}$$

With the following exceptions in mind, the rest of the treatment of level 2 is identical to level 1:

{i} Several recursive systems are to be solved. This involves nothing more than an extra DO loop and some indexing arithmetic.

{ii} At each stage of level 2 reduction and back substitution, point tridiagonal systems will have to be solved by level 3. These point tridiagonal matrices have the form:

{5.6} $$\begin{bmatrix} -6 + 2\cos\theta_j^{\{r\}} + 2\cos\theta_k^{\{s\}} & 1 & \\ 1 & \ddots & \ddots \\ & \ddots & \ddots \end{bmatrix}$$

With the $\theta_j^{\{r\}}$ term arising from the particular stage of level 1 reached and the $\theta_k^{\{s\}}$ term arising from the particular stage of level 2 reached.

As in level one, let us summarize the execution rates achieved at level two.

A total of 2850816 floating point operations are accomplished in 71.43 msec. for a megaflop rate of 39.9. However, if we add in the data motion, the time rises to 181.93 msec. for an effective rate of 15.7 megaflops for level two.

Level 3

At level three we actually solve the point tridiagonal systems generated at level two.

The number of RHS's to be solved for, NRHS, and the number of recursive tridiagonal systems for each RHS, NTRI are related as follows:

For NB = 1,2,4,8,16

NRHS	NTRI
NB	32
NB	16
2NB	8
4NB	4
8NB	2
16NB	1

We now seek an algorithm to efficiently solve NTRI tridiagonal system with NRHS right hand sides efficiently on STAR.

One possibility is to simultaneously solve the various RHS's in vector form using the standard LU decomposition. This involves more data motion to line up the operands and only results in a mean vector length of 32. Thus it can be discarded.

One can also use straightforward point cyclic reduction. This may be a viable alternative on the STAR-100B with its improved merge instruction. However, compared to the algorithm used, it is marginal at best on the STAR-100 and certainly inferior on the STAR-100A.

Let us now discuss the algorithm of level three. There are 63 x 64/2 {= 2016} different 63 x 63 tridiagonal matrices each of the form

$$\begin{bmatrix} d & 1 & & \\ 1 & \ddots & \ddots & \\ & \ddots & \ddots & \ddots \\ & & \ddots & \ddots \end{bmatrix} \qquad \{5.7\}$$

if one examines the LU decomposition for such a matrix,

$$\begin{bmatrix} 1 & & & \\ a_2 & \ddots & & \\ & \ddots & \ddots & \\ & & a_{63} & 1 \end{bmatrix} \cdot \begin{bmatrix} b_1 & 1 & & \\ & \ddots & \ddots & \\ & & \ddots & 1 \\ & & & b_{63} \end{bmatrix} ,$$

two facts become apparent:

{i} $a_{j+1} = \frac{1}{b_j}$ $\qquad j = 1,62$

{ii} In the substitution phase one wishes to divide by the b_j's i.e. to multiply by $1/b_j$

Hence the LU decomposition of {5.7} can be "held" in 63 words.

In cases {such as the present} where the Poisson solver is used each time step in a transient problem, it becomes possible to once and for all produce the 2016 LU decompositions and store them in 127008 words.

This method has been utilized in the present case, and we store each LU decomposition in a 64 word block. Moreover, the present version actually stores all 63^2 decompositions in a 262144 word array. This allows smoother paging characteristics.

The total LU data occupy 4 large pages of STAR virtual memory. With the advent of the million word {16 large page} STAR-100A, and larger memories on the horizon, this subsidiary data base is no longer frightening.

Thus, level three carries out the substitution phase of the LU decomposed point tridiagonal systems generated by levels one and two.

Since this algorithm is recursive, it would seem to imply a tremendous degradation of performance from the standards attained by levels one and two. While it is true that STAR-100 performance of level three is not as good as levels one or two, it is still respectable. Moreover, the performance of the STAR-100A on level three is comparable to that of level two.

The main impediment to high performance scalar code on the STAR-100 is the load/store bottleneck e.g. a succession of loads causes a} memory bank conflicts and b} result address register overload.

We can circumvent this problem using the SWAP instruction. This instruction allows one to store the contents of a contiguous set of registers to a contiguous set of {virtual} memory locations while simultaneously loading the contents of some other contiguous set of memory locations to the register file. {This instruction demands that an even number of words be moved in either direction and the memory addresses that start both the load and store stream be on double word boundaries. Hence, our passion for 64 word blocks instead of 63.} After a 62 cycle startup, this instruction moves two words in and two words out of the register file each cycle.

The next problem we face is the recursion. Let us discuss two facets of STAR architecture that ameliorate the situation for recursion.

First, we need some further data about STAR.

STAR arithmetic scalar instructions are register to register i.e. two registers supply the inputs and a third accepts the output.

The META mnemonic for multiply and subtract are MPYS and SUBN.

There are four epochs of importance to a scalar arithmetic instruction a} begin issue b} issue c} shortstop d} register file arrival.

a} is the epoch at which one can consider the instruction to begin whatever processing is necessary for it to issue {such as procuring operands from the register file}.
b} is that epoch at which the instruction commences usage of the functional unit d} is that epoch at which the output is returned to the register file. As we go along, we will explore c}.

Suppose the 63 values of a_j are in registers L2...L63 with $1/b_{63}$ in L64 and the 63 values of a RHS are in registers P1...P63. The following instruction stream will execute the forward phase of the solution and deposit the solution in registers P1...P63

```
MPYS     L2, P1, TEMP
SUBN     P2, TEMP, P2
MPYS     L3, P2, TEMP
SUBN     P3, TEMP, P3
           .
           .
           .
MPYS     L63, P62, TEMP
SUBN     P63, TEMP, P63
```

If we time this instruction stream for the STAR-100, the first multiply begins issue at cycle 0, issues at cycle 2 and reaches the register file at cycle 17. However, at cycle 12 the output operand is available to be put right back into the functional unit if that is in line with the instruction stream. This is the shortstop capability. In this case the opportunity to use the shortstop operand occurs <u>for each instruction</u>. This opportunity continues right through the instruction stream for the back substitution.

We are not done yet, however. Notice that the cycle epochs for the first four instructions are:

a	b	c	d
0	2	12	17
10	12	18	23
16	18	28	33
26	28	34	39

There is a lot of dead time between these instructions, viz.,
we are only getting one floating point result each 8 cycles.

If we can squeeze in some other useful work, we can increase
the effective megaflop rate. What we can squeeze in is
another forward substitution. In most cases at level
three there are at least two RHS's. Since we have 256
registers, there is ample room to put another RHS in
register Q1...Q63. Then consider the instruction stream:

```
{5.8}  MPYS    L2, P1, TEMP         0    2   12   17
       MPYS    L2, Q1, TEMQ         2    4   14   19
       SUBN    P2, TEMP, P2        10   12   18   23
       SUBN    Q2, TEMQ, Q2        12   14   20   25
       MPYS    L3, P2, TEMP        16   18   28   33
       MPYS    L3, Q2, TEMQ        18   20   30   35
       SUBN    P3, TEMP, P3        26   28   34   39
       SUBN    Q3, TEMQ, Q3        28   30   36   41
```

It is easy to see we have increased the effective execution
rate to one floating point result each 4 cycles, a respectable
if not spectacular 6.25 megaflops.

In its present form, level three has an outer loop on pairs
of RHS's. At the beginning of each pass through the outer
loop we can simultaneously SWAP in two new RHS's and SWAP
out two solutions, a total of 256 words, in 126 cycles.

The inner loop begins by SWAPing in a new set of a's, a total of 64 words in 94 cycles. It then proceeds through the 498 arithmetic instructions to complete a forward and back substitution.

Level three performs 9179136 floating point operations in 1.86 sec. for a megaflop rate of 4.9.

In addition to the three components already discussed, viz., vector arithmetic, vector data motion and META register file scalar arithmetic, there is also overhead, to take care of such housekeeping duties as subroutine linkage, descriptor setup and integer index arithmetic. This overhead takes approximately .14 sec. To sum up, on the STAR 100, we have performed 12505088 floating point operations in 2.2 sec. for an overall 5.7 megaflop rate.

6. STAR 100A and Beyond

Let me briefly discuss the projected performance of this program on the STAR 100A which, at the time of this writing, is in final checkout.

The vector performance of levels one and two, of course, will not be affected. The improvement will come in the scalar arithmetic of level three and the overhead.

The four instruction kernel illustrated in {5.8} executes in 16 40 nsec. cycles on the STAR 100. On the STAR 100A this same kernel will execute in 10 20 nsec. cycles. Furthermore, the significantly improved load capability of the STAR 100A allows the inner loop SWAP to be replaced by loads sprinkled throughout the forward substitution. All told, it is projected that level three will execute in .58 sec. on the STAR 100A, which results in a 15.8 megaflop rate.

It is more difficult to tightly estimate the performance of the overhead on the STAR 100A. One of the main reasons for this uncertainty is that much of the overhead will be overlapped with vector arithmetic. We estimate that the overhead will execute in .04 sec.

With these estimates for level three and the overhead, one can project an execution time of .82 sec. on the STAR 100A which results in a 15.3 megaflop rate.

Future enhancements to STAR architecture will not only improve the megaflop rate of this algorithm, but they will also afford the opportunity to rethink the choices of implementation made at various levels. In particular, the STAR 100B with its significantly faster vector arithmetic and merge capability may bring vectorized point cyclic reduction back into the picture at level three.

However, in order to take the fullest advantage of this algorithm on STAR, one must look ahead to the point where the data motion and vector arithmetic of levels one and two can be done in parallel. Recall that the vector arithmetic of level one executes virtually at peak hardware rate even on the STAR 100.

7. Conclusion

The results we have presented can be tabularized as follows:

	FLOPS	STAR 100	
Level One	475136	.018 sec.	27.0 mflops
Level Two	2850816	.182 sec.	15.7 mflops
Level Three	9179136	1.86 sec.	4.9 mflops
Overhead	—	.14 sec.	—
Total	12505088	2.2 sec.	5.7 mflops

	FLOPS	STAR 100A	
Level One	475136	.018 sec.	27.0 mflops
Level Two	2850816	.182 sec.	15.7 mflops
Level Three	9179136	.58 sec.	15.6 mflops
Overhead	—	.04 sec.	—
Total	12505088	.82 sec.	15.3 mflops

These performance figures are "portal to portal", i.e., they include everything. They are not what Prof. B. Marschner of Colorado State has termed "EPA" computer ratings. In perusing the literature, all too often, one sees algorithm performance for a given architecture rated in terms of megaflops for selected kernels {presumably chosen to prove a point, be it positive or negative} or in terms of operation count per node without regard to the computational environment of these operations. It would seem that one of the simplest, most direct measures of an algorithm's performance is the total execution time. This is especially true for those who are more interested in solving real applications problems....

REFERENCES

1. B. L. Buzbee, G. H. Golub and E. W. Nielson: On Direct Methods for Solving Poisson's Equations. SIAM J. Numer. Anal. Vol 7 No. 4 December 1970.

2. Control Data Corporation: STAR 100 Computer Hardware Reference Manual No. 60256000.

3. Control Data Corporation: STAR 100 Features Manual No. 60425500.

4. Control Data Corporation: STAR 100A Computer Hardware Reference Manual No. 60256010.

5. M. J. Kascic, Jr.: Notes from the STAR seminar.

ADDENDUM

According to the great mathematician ANONYMOUS, "Sophistication is the ability to transform ten lines of simple logic into two lines of unreadable jargon".

Nowhere is the thrust of this principle better illustrated than in level one. Recall that level one executes at 27 megaflops. Of an evening, while perusing the completed manuscript, I chanced to ask myself an a posteriori very obvious question. If all the vector arithmetic of level one were done without data motion, what execution rate could be achieved? The initially embarassing (to the author, certainly not to STAR) answer is 46 megaflops.

The very simple fact is that at vector length 2048, STAR is operating at better than 40 megaflops. Thus, if level one were performed in "bits and pieces", it would run significantly faster than it does organized into longer vectors according to a "sophisticated" scheme.

Erasing much of the author's chagrin is the fact that since level one carries a small percentage of the computational load, the bottom line performance is unchanged.

One might ask whether the same situation exists at level two. The answer is no, since the extra descriptor setup time along with the limitation of vector length 64 leads to a maximum 14 megaflop rate. However, even this figure is misleading since at level three, the fact that all the 64 word blocks are contigious leads to a simple SWAPing scheme. At the very least, a more time consuming addressing scheme would be needed. Indeed, the present SWAPing scheme of level three depends, as a sine qua non, on having pairs of 64 word blocks contiguous.

VECTORIZATION OF BLOCK RELAXATION TECHNIQUES SOME NUMERICAL EXPERIMENTS

by

Daniel L. Boley*
University of California
Los Alamos Scientific Laboratory
Los Alamos, New Mexico

ABSTRACT

Carefully vectorized linear system solvers achieve performance levels of 40 to 90 million floating point operations per second on the CRAY-1. Since classical point successive over-relaxation is not easily vectorized, one way to exploit the potential performance of the CRAY-1 is block relaxation using the aforementioned linear system solvers. This paper compares the performance of three schemes of Block Successive Over-Relaxation on the Model Problem.

INTRODUCTION

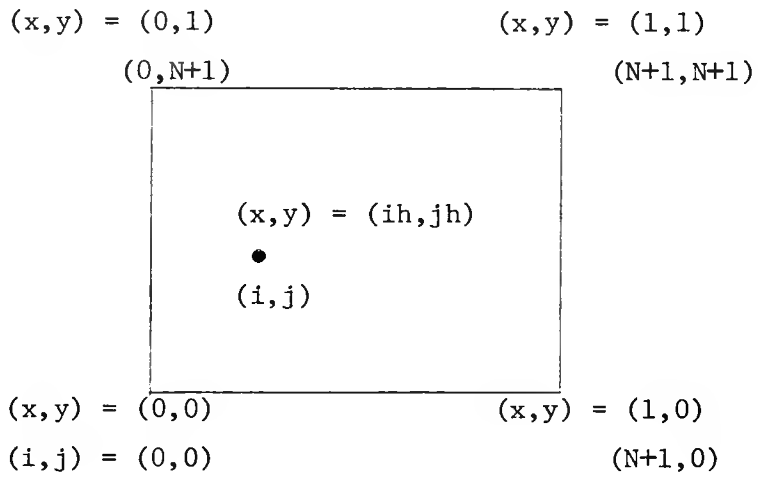
Classical Point Successive Over-Relaxation schemes using natural ordering and written in a language such as Fortran can typically run at a rate of ~ 2 million floating point operations per second (2 megaflops)[6] on a machine such as the CRAY-1 without taking advantage of the vector architecture. If we use a checkerboard ordering we can achieve up to 20 megaflops.[6]

However, by using a Block S.O.R. scheme we can use a linear system solver to do much of the work and thereby take advantage of the vector architecture. Such solvers can run from 40 megaflops for a tridiagonal system solver to 90 megaflops for a band solver when using the vector hardware of the CRAY-1.[4,5] We would like to compare the performance of three different schemes of Block Relaxation that utilize these kinds of linear system solvers. To do this we study solving the model problem; i.e., Laplace's equation $\Delta u=4$ on the unit

*Current address, Computer Science Department, Stanford University, Stanford, CA 94305.

square with boundary conditions $x^2 + y^2$ on all four boundaries. The solution is $u = x^2 + y^2$, and this solution also exactly satisfies the discrete equation derived from the 5-point formula. We consider solving the model problem using three different Block Successive Over-Relaxation schemes on the CRAY-1. The three schemes are: scheme L -- Line S.O.R., in which each block is a single column; scheme K -- K-line S.O.R., in which each block is a group of K consecutive columns; and scheme B -- square block S.O.R., in which each block is a K x K square set of grid points. Scheme L uses a tridiagonal solver that takes advantage of the vector architecture by solving for up to 64 systems simultaneously. Schemes K and B use a band solver.

Let us discretize the unit square so that grid point (i,j) corresponds to point (x,y) = (ih,jh) where h = 1/(N+1). Thus, the grid looks like



There are N^2 unknowns and 4N boundary values. We assume that the interpoint spacing h is the same in both the x and y directions.

We denote the spectral radius of the Jacobi method associated with each block scheme by ρ_L, ρ_K, ρ_B, the optimal ω in each case by $\omega_L, \omega_K, \omega_B$, and the spectral radius of the associated block S.O.R. iterator by $\lambda_L, \lambda_K, \lambda_B$. Then we have the following relationships (true for all three schemes, so the subscripts will be omitted):[1]

$$\omega = \frac{2}{1+\sqrt{1-\rho^2}} \quad ; \quad \lambda = \omega - 1 = \frac{1-\sqrt{1-\rho^2}}{1+\sqrt{1-\rho^2}} = 2\left(\frac{1-\sqrt{1-\rho^2}}{\rho^2}\right) - 1 \; . \quad (1)$$

Using the results from [Boley, Buzbee and Parter]^2 we have that

$$\rho_B = 1 - \frac{K}{2}(\pi h)^2 \qquad \rho_K = 1 - K(\pi h)^2 \tag{2}$$

and $\rho_L = 1 - (\pi h)^2$ is the same as ρ_K with $K = 1$.
Combining (1) and (2) gives:

$$\omega_B = \frac{2}{1 + \pi h\sqrt{K}} \qquad \omega_K = \frac{2}{1 + \sqrt{2}\ \pi h\sqrt{K}} \qquad \omega_L = \frac{2}{1 + \sqrt{2}\ \pi h} \tag{3a}$$

and

$$\left.\begin{aligned} \lambda_B &= 1 - 2\pi h\sqrt{K} + 2(\pi h)^2 \\ \lambda_K &= 1 - 2\sqrt{2}\ \pi h\sqrt{K} + 4(\pi h)^2 \\ \lambda_L &= 1 - 2\sqrt{2}\ \pi h + 4(\pi h)^2. \end{aligned}\right\} \tag{3b}$$

The number of iterations needed to reduce the initial error e_0 by a factor of h^2 is

$$I = \frac{-\log h^2}{-\log \lambda} \ .$$

In each of the three schemes this yields

$$I_B = \frac{C}{2\pi h\ \sqrt{K}} \ ; \quad I_K = \frac{C}{2\ \sqrt{2}\ \pi h\ \sqrt{K}} \ ; \quad I_L = \frac{C}{2\ \sqrt{2}\ \pi h} \tag{4}$$

where $C = -\log h^2$.
Note here that the number of iterations is smallest for the K-Line scheme.

Now we turn to the work required in each iteration. We are solving for N^2 unknowns in each sweep. We assume that the matrices are factored only once at the beginning, so we show only the cost of the back-substitutions.

In Scheme L each block has N unknowns and we have a symmetric tridiagonal system. The work in each block is 3N (see [Dahlquist and Bjork]^3), and there are N blocks so that the total work is $3N^2$.

In Scheme K, each block has K columns for a total of NK points. If we scan each block by rows, the matrix for each block becomes a symmetric banded system of order NK with K super-diagonals (half-bandwidth = K). The work to solve this system is (2K + 1)NK operations.[3] There are N/K blocks so the total work per sweep is $2KN^2 + N^2$.

In Scheme B, each block consists of K^2 points giving a symmetric banded system of order K^2 with half-bandwidth K. To solve this requires $K^2(2K + 1)$ operations.[3] There are $(N/K)^2$ blocks over the whole grid, so the total work per sweep is $2KN^2 + N^2$. Note that this is the same as for the K-line scheme.

We combine work per sweep with the estimated number of sweeps to arrive at Table I.

Table II tabulates the results of numerical experiments. In the experiments only the linear system solvers were vectorized. These were hand-coded by T. Jordan in CRAY Assembly Language (CAL)[4,5] and called each time as a subroutine. All the rest of the processing was in unvectorized Fortran. The iteration started with an initial guess of 0 and continued until the ∞-norm of the residual was less than $\frac{1}{2} h^2$. The storage requirements listed are just those for storing the matrix for 1 block. In the case of the Line scheme, unknowns in lines 1, 3, 5, 7 ... were solved for at once and then the unknowns in lines 2, 4, 6, 8 ...; we define this as odd-even ordering. This was done in order to

TABLE I

COMPARISON OF VARIOUS PARAMETERS BY SCHEME

Scheme	ρ	ω	λ	Work/ Sweep	Expected # of Sweeps	Total Expected Work
L	$1-(\pi h)^2$	$\frac{2}{1+\pi h\sqrt{2}}$	$1-2\sqrt{2}\pi h$	$3N^2$	$\frac{C}{2\sqrt{2}\pi h}$	$\frac{3}{2\sqrt{2}} \frac{C}{\pi} N^3 = .338\ CN^3$
K	$1-K(\pi h)^2$	$\frac{2}{1+\pi h\sqrt{K}\sqrt{2}}$	$1-2\sqrt{2}\pi h\sqrt{K}$	$2KN^2+N^2$	$\frac{C}{2\sqrt{2}\pi h\sqrt{K}}$	$\frac{2\ C\sqrt{K}\ N^3}{2\sqrt{2}\ \pi} + \frac{CN^3}{2\sqrt{2}\pi\sqrt{K}} = .225\ CN^3\sqrt{K} + .113\ \frac{CN^3}{\sqrt{K}}$
B	$1-\frac{K}{2}(\pi h)^2$	$\frac{2}{1+\pi h\sqrt{K}}$	$1-2\pi h\sqrt{K}$	$2KN^2+N^2$	$\frac{C}{2\pi h\sqrt{K}}$	$\frac{C\sqrt{K}\ N^3}{\pi} + \frac{C\ N^3}{2\pi\sqrt{K}} = .318\ CN^3\sqrt{K} + .159\ \frac{CN^3}{\sqrt{K}}$

where $C = -\log h^2 = \log(N+1)^2$

TABLE II

VECTORIZED S.O.R. CONVERGENCE DATA

(Numbers in parenthesis are the grid size N)

Storage required for matrix

Scheme L ... $3(N/2)\cdot N = 6144_{(64)}$ $24576_{(128)}$

K	Scheme K (64x64)	Scheme B (64x64)	Scheme K (128x128)	Scheme B (128x128)
2	$(K+1)KN = 384$	$(K+1)K^2 = 12$	768	12
4	1280	80	2560	80
8	4608	576	9216	576
16	17408	4352	34816	4352
32	67584	33792	135168	33792
64	266240	266240	532480	266240

Iteration Count

Scheme L ... $143_{(64)}$ $314_{(128)}$

K	Scheme K (64)	ratio $\sim \sqrt{2}$	Scheme B (64)	ratio $\sim \sqrt{2}$	Scheme K (128)	ratio $\sim \sqrt{2}$	Scheme B (128)	ratio $\sim \sqrt{2}$
2	113	1.41	159	1.43	245	1.43	346	1.44
4	80	1.33	111	1.39	171	1.39	240	1.41
8	60	1.20	81	1.29	123	1.32	170	1.36
16	50	.98	63	1.09	93	1.19	125	1.28
32	51	3.60	58	2.32	78	.96	98	1.07
64	17		25		81		92	

Iteration Times in Seconds

Scheme L ... $1.636_{(64)}$ $14.201_{(128)}$

K	Scheme K (64)	ratio $\sim \sqrt{2}$	Scheme B (64)	ratio $\sim \sqrt{2}$	Scheme K (128)	ratio $\sim \sqrt{2}$	Scheme B (128)	ratio $\sim \sqrt{2}$
2	2.592	1.48	5.503	2.68	22.265	1.50	47.884	2.26
4	1.749	1.33	2.659	1.52	14.856	1.38	23.303	1.69
8	1.316	1.14	1.748	1.26	10.741	1.25	14.674	1.34
16	1.159	.84	1.383	.93	8.598	1.01	10.972	1.09
32	1.388	2.21	1.487	1.69	8.476	.71	10.052	.78
64	.627		.881		11.945		12.960	

effectively vectorize the solver. The storage in the line case reflects the need to store the matrices for half the blocks at once. The other two schemes used a vectorized band solver.

The number of iterations required diminishes as K increases in proportion to $\sqrt{K}$. Comparing the number of iterations between the K-line and the square block cases, the ratio is $\sqrt{2}$ as expected. These properties do not continue as K approaches N. This is also expected since the analysis assumed that N was much larger than K (cf. [Boley, Buzbee and Parter]^2).

The third part of the table is the work required to iterate until the residual is reduced to $1/2\ h^2$. The most unusual feature in this table is that the times go through a minimum as K increases. This is partly explained by the failure of the analysis as K approaches N, and partly by the overhead involved in the subroutine calls. This latter property is exhibited by Table III.

The times in Table III should increase linearly in K. For small K the times are actually decreasing, showing that much of the time is spent in calling the linear system solver and in the overhead of operating with short vectors. The work done by the solver per subroutine call increases as K increases, lessening the effect of overhead in the linkage.

TABLE III

SECONDS PER ITERATION IN SQUARE BLOCK SCHEME

K	Seconds	Ratio	
2	.1412	.699	
4	.0988	.888	
8	.0877	1.02	128 x 128 grid
16	.0891	1.18	
32	.1039	1.36	
64	.1422		

Returning to Table II, Scheme L compares quite favorably with schemes K and B, and takes a lot less storage. If we went to larger problems then Schemes K and B would probably beat Scheme L by a more substantial margin, but the storage required to achieve such speeds would be impossibly large.

In Table IV we show the times for doing the exact same problem with no vectorization. Except for K=2 when the overhead is large, the times are monotonic increasing. When K is small enough for the analysis to be valid ($4 \le K \le N/4$) the ratios in the times even approach $\sqrt{2}$. One can tell when the analysis begins to break down by noticing when the iteration counts stop decreasing like $\sqrt{K}$.

TABLE IV

UNVECTORIZED S.O.R. CONVERGENCE DATA

Iteration Count

K	Scheme K (64)	Scheme B (64)	Scheme K (128)	Scheme B (128)
2	113	159	245	346
4	80	111	171	240
8	60	81	123	170
16	50	63	93	125
32	51	58	78	98
64	17	25	81	92

(Above exactly matches vectorized case as expected.)

Times in Seconds

K	Scheme K (64)	1/ratio	Scheme B (64)	1/ratio	Scheme K (128)	1/ratio	Scheme B (128)	1/ratio
2	5.052	0.95	6.699	0.91	43.812	0.94	58.300	0.91
4	4.784	1.15	6.118	1.15	40.961	1.11	52.908	1.12
8	5.523	1.43	7.036	1.37	45.391	1.30	59.066	1.29
16	7.880	1.87	9.612	1.72	58.802	1.54	76.284	1.47
32	14.740	0.64	16.532	0.85	90.525	1.99	111.730	1.81
64	9.393		13.760		179.674		202.548	

The improvement achieved in using vectorization is 2 to 14-fold, with the speedup around 7-fold at the optimal values of K for the vectorized cases (K=N/8). Vectorizing some of the system preparation and convergence tests, or writing the entire inner loop in CAL would make the speedup even more marked, and less sensitive to K.

Conclusions: On a vector machine such as the CRAY-1, it is possible to vectorize the relaxation iteration schemes. As long as the block size $K < \frac{1}{4} N$, where N is the grid size, the work diminishes, but at a prohibitive cost in storage. The old standard single-line case with odd-even ordering seems to be the best compromise in most situations.

ACKNOWLEDGMENTS

I am indebted to Thomas Jordan for providing the vectorized kernels, and to B. L. Buzbee and S. V. Parter for their ideas and guidance. This work was performed under the auspices of the Los Alamos Scientific Laboratory when I was there during the summer of 1978.

REFERENCES

1. J. M. Ortega, Numerical Analysis, A Second Course, (Academic Press, New York, 1972).

2. D. Boley, B. Buzbee and S. Parter, "On Block Relaxation Techniques," University of Wisconsin, Math Research Center Report #1860 (June 1978).

3. Dahlquist and Bjork, Numerical Methods, translated by Ned Anderson (Prentice Hall, Englewood Cliffs, NJ, 1974), p. 166.

4. T. L. Jordan, private communication, Los Alamos Scientific Laboratory, (July/August 1978).

5. K. Fong and T. L. Jordan, "Some Linear Algebraic Algorithms and Their Performance on the CRAY-1," Los Alamos Scientific Laboratory report LA-6774 (June 1977).

6. J. J. Dongarra, "LINPACK Working Note #11: Some LINPACK Timings on the CRAY-1," Los Alamos Scientific Laboratory report LA-7389-MS (August 1978).

VECTORAL: A VECTOR ALGORITHMIC LANGUAGE FOR ILLIAC

by

Alan A. Wray
NASA-Ames Research Center
Moffett Field, California

ABSTRACT

Though a FORTRAN-like language, CFD, has been in use on ILLIAC IV for several years, the need has arisen for a more flexible and powerful language on that ancestral maxicomputer. Several goals may be given for this project:

(1) provide modern control structures to enable and encourage the writing of well-structured programs;

(2) provide an optimizer to enhance machine productivity and to allow source programs to be more human engineered without loss of machine efficiency;

(3) provide language features to exploit all of the important hardware features;

(4) provide a variety of useful language constructs to ease the implementation of the very large codes generally run on ILLIAC.

Some examples of the language features available in VECTORAL and their syntax in the above four areas are as follows:

(1) A two-way Boolean branch is provided by IF ... THEN ... ELSE, a multi-way branch by DO CASE <integer expression>, and a looping Boolean branch by WHILE; statement brackets are used with these control structures to permit the logical grouping of several statements into one object statement. Simple GO TO is allowed, but labels are identifier-like strings of higher mnemonic value than numeric labels. Nested procedure declarations and corresponding levels of nomenclature are provided.

(2) The optimizer features folding of constant expressions, redundant operation elimination, movement of invariant code from loops, reduction of strength of multiplications and routes by iteration counters, elimination of

redundant routes, and various optimizations of the use of Control Unit memory, as well as localized machine-dependent optimizations.

(3) Several useful hardware features are given a high-level language implementation not available in CFD: vector and scalar bit string operations (and, or, exclusive or, bit setting, resetting, complementing, and testing, shifting, and rotating) and the single precision (32 bit) mode of operation to provide a 128 PE machine.

(4) A large number of "small" but nevertheless useful features are provided to make the programmer's life easier, e.g., identifier names of arbitrary length, no significance to card boundaries or columns, nested iterative data specifications, recursive procedures, scalar as well as vector functions, EBCDIC and ASCII character strings, no restrictions on expression complexity, up to 15 indices, arbitrarily complex Boolean expressions, use of the "natural" relational symbols (< , > , = rather than .LT. , .GT. , .EQ.), brackets and braces allowed as well as parentheses.

An Introduction to VECTRAN and Its Use in Scientific Applications Programming

by

George Paul
M. Wayne Wilson

IBM T.J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT

VECTRAN is an upward compatible extension to FORTRAN which permits the problem programmer/scientist to incorporate and utilize natural vector/matrix problem formulation in his program. VECTRAN minimally extends the syntax of FORTRAN while greatly extending its semantics.

VECTRAN contains IBM FORTRAN IV as a proper subset.

This paper will introduce the basic concepts and new syntax provided in the VECTRAN extensions, and will demonstrate the utilization of VECTRAN in a variety of numerical algorithms.

I. INTRODUCTION

VECTRAN [1] is an experimental language extension to IBM FORTRAN IV [2] developed within IBM to study and facilitate the application of vector/array processing algorithms. An experimental prototype compiler has been written for VECTRAN.

Since VECTRAN was developed as a tool both to study the potential applicability and extent of vector processing in scientific computation and to define the prerequisite language support required, several design prerequisites and functional requirements were specified for the design/implementation of the language/compiler.

Among the functional requirements specified for VECTRAN, the authors felt that to be viable the language must go beyond the use of unsubscripted array names in expressions and the simple semantic extension of current operators and library functions to operate distributively, element by element upon the scalar elements of arrays. Consequently the following five requirements were set forth for the language. The language must provide:

i) a convenient means to specify rectangular subarrays and general sections of arrays
ii) a direct facility to manipulate both arrays and subarrays as entities on either side of assignment statements
iii) array-valued functions and the use of array-valued expressions as arguments to subprograms
iv) an array oriented operator set including matrix and reduction operators
v) facilities to support sparse matrix algorithms employing both logical arrays and index vectors as means to specify the structure and compressed storage addressing mechanisms.

Among the design prerequisites specified, the authors felt that two considerations -- performance and compatibility, should dictate the language and compiler design. Consequently the following guidelines were set forth. The language/compiler should:

i) maintain an object-time orientation and provide for early binding (compilation) wherever possible and avoid constructs which would require late binding (interpretation) at object-time
ii) provide for subset containment, i.e., the new features where not utilized should not adversely affect performance
iii) be upward compatible with existing FORTRAN and be immediately comprehensible as an extension to FORTRAN
iv) be notationally analogous to 'standard' mathematical convention wherever possible within the constraints of the character set, etc.
v) as an operational and conversion consideration, should allow mixed FORTRAN and VECTRAN object modules.

II. NEW CONCEPTS

The FORTRAN concept of an array is limited only to view arrays as a set or aggregate of data items (scalars) identified by a single symbolic name. This notion allows only the use of individual elements of an array in expressions, and expressions must always represent a single scalar value. In order to broaden this narrow view of arrays, VECTRAN introduces the concept of an array value and its ancillary notions of atomicity, range and conformability and the concepts of elemental and transformational operators and functions.

A. Array Values and Atomicity

VECTRAN introduces the concept of an array value and defines this concept in a manner analogous to the relationship defined in FORTRAN between arrays and scalar data items. An array value is defined to be an aggregate of scalar values. In VECTRAN entire arrays, subarrays and/or subsets of arrays, as well as elements of arrays may be used in expressions. Furthermore an expression may represent either an array value or a scalar value.

As a consequence of this concept VECTRAN maintains a sharp distinction between the storage (memory allocation) associated with an array name and the array value represented by that name. This point will be elaborated upon further as we proceed.

Individual scalar values bear the same relation to the array value they compose as subscripted variables do to the array name. However, in place of the explicit subscript quantities used to specify subscripted variables, we refer in VECTRAN to the 'implied free indices' of an array value. These indices are analogous to the indices of tensor analysis, and imply a rectilinear structure to array values.

The implied free indices which exist in VECTRAN for all array values from simple array names to complicated array-valued expressions, are implicit and independent of explicit subscripts which may or may not exist. The implied free indices are independent of storage locations or addresses. All array values in VECTRAN are rectilinear. Each free index conceptually running in unit steps from one to a specified upper limit value.

In VECTRAN an array value is a single atomic entity, as it is conceptually in mathematics. The use of an array name in an expression refers to the entire array value, and it is not simply an abbreviation for a sequence of scalar values. In an assignment statement involving array values, the array value specified by the right-hand-side expression is conceptually totally evaluated prior to assignment. Hence the atomicity of array-valued expressions is maintained.

B. Range and Conformability

The bounds on the implied free indices of an array value are defined by the range associated with the array value. The range of an array-valued entity or expression is a list of N entries, where N is the maximum rank (number of dimensions) allowed for arrays in the compiler implementation. (In the current implementations of IBM FORTRAN IV and VECTRAN, N is equal to seven.) Each entry may be either a positive integer constant (the value representing its name) or the name of an unsubscripted integer variable. Range defines at compile-time the structural characteristics (shape and bounds) of the array value represented at object-time. The actual number of scalar values contained or represented by the array value remains a dynamic function of the values represented by the range entries at object-time, and the range of the array is independent of the storage allocation (DIMENSION) associated with the array name.

Array-valued entities or expressions are judged to be conformable if they have identical ranges. A scalar entity or expression is defined to be conformable with an entity of any range. Note, two arrays may be conformable even if their rank (dimensionality) are unequal, and on the other hand, two arrays at object-time may have identical shapes even though they are not conformable. Conformability is purely a compile-time concept, and allows the compiler to determine the legality of expressions at that time.

Rank, range and conformability are compile-time concepts analogous to the concepts of length, type and mode in FORTRAN. Rank and range, like mode, propagate through expressions according to rules associated with each operator in VECTRAN.

C. Elemental and Transformational Operators/Functions

An operator or function is said to be elemental if it operates distributively on each element of its array-valued argument(s) independently, and if its result has the same range as its arguments.

An operator or function which is not elemental is said to be transformational. Typically the elements of the result of a transformational operator or function are each computationally dependent on several or all elements of the operands, and/or the array-valued result has a range which is different from its operands. Matrix multiply, reduction operators and matrix transposition are examples of transformationals.

III. ARRAY IDENTIFIERS

The range of an array-valued data item may be specified by the user by means of the RANGE statement. The RANGE statement is a new declaration statement which may be used in lieu of, or in conjunction with the DIMENSION statement.

The syntax of the RANGE statement is illustrated below:

General Form

RANGE $/r_1/a_{1,1}(k_{1,1}), \ldots, a_{1,N}(k_{1,N}) \ldots /r_M/a_{M,1}(k_{M,1}), \ldots$

Where:

Each $a_{i,j}$ is an array name, or the name of an array-valued function subprogram supplied by the user.

Each $k_{i,j}$ is optional, and is composed of one through seven unsigned integer constants, separated by commas, representing the maximum value of each subscript in the array. Each $k_{i,j}$ may contain integer variables only when the RANGE statement in which they appear is in a subprogram and the corresponding $a_{i,j}$ is a dummy argument of that subprogram.

Each r_i is composed of one through seven unsigned integer constants or integer variables (not array elements), separated by commas, representing the range list to be associated with the list of array names which follows it. Note, the number of entries in each range list determines the rank specified for the associated array names, and this number should be equal to the number of entries in the dimension list for these arrays.

If an explicit RANGE statement does not appear for an array, its range list is determined from the dimension data provided for the array. Thus preserving upward compatibility with FORTRAN.

The use of a variable in a range list in no way restricts its normal usage in the program. Consequently, the active range of an array at object-time may be varied dynamically by assigning new values to the variables referred to in the range list.

A. Unsubscripted Array Identifiers

Array names in VECTRAN may be used without subscripts in arithmetic or logical expressions to designate array-valued entities. When utilized in this manner, the unsubscripted array name references only those elements in the array which lie within the currently defined range. These elements are referred to as belonging to the 'principle subarray' of the referenced array.

For example:

```
RANGE/N,M/ A(10,10),B(15,25)
          .
          .
          .
N = 5
M = N+2
          .
          .
          .
A = 2.5*A+B
          .
          .
          .
```

In this example, the dimensioned storage allocations for A and B are [10x10] and [15x25] respectively. Both A and B are defined to have range NxM. The integer variable N is assigned the value 5, and M is assigned the value 7 (5+2). The statement

```
A = 2.5*A+B
```

thus refers to the [5x7] principle subarrays of A and B.

B. Sections of Arrays

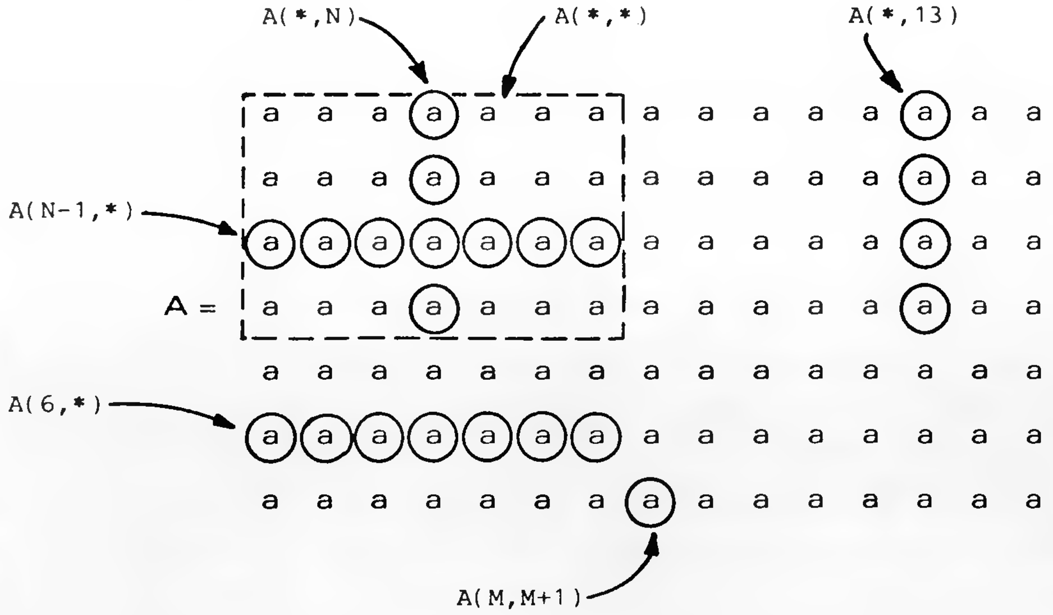
In addition to unsubscripted array names, VECTRAN allows the use of sections of arrays as array-valued data items. (The concept of array sections in VECTRAN is somewhat analogous to that of PL/I.)

In VECTRAN a section may for example be that part of any row, column, etc. of an array which lies within the active range of the array. A section is designated by the use of an asterisk (the section selector symbol) as a subscript selector. The asterisk implies that all values of the designated dimension within the object-time range are to be used in the expression. For example, G(*,K) implies that all elements (rows) within the active range of the K'th column of the matrix G are to be used; G(J,*), that all elements (columns) within the active range of the J'th row are to be used and G(*,*), the entire active range of the matrix is to be used. The latter is entirely equivalent to using the array name G unsubscripted.

The reader should carefully note the distinction between PL/I and VECTRAN sections:

The range of a VECTRAN section is determined by selecting the ordered subset of the range list entries of the parent name, specified by corresponding section selector symbols. Consequently, the sections retain the active range of the parent array within the dimension(s) of their free subscripts. The rank of the section is the number of section selector symbols, i.e., the number of free indices. (PL/I sections apply to the entire declared dimension of the array.) Note, this characteristic is retained even though the 'fixed' subscript quantities of the section may lie outside of the active range designated for their respective dimensions. Refer to the examples below.

```
RANGE/N,M/A(7,15)
M = 7
N = M/2 + 1
```



Sections may be used in VECTRAN in the same manner as any other array-valued entity in the language.

VECTRAN further extends the section concept by defining shifted sections. We have thus far considered only designation of contiguous arrays and subarrays defined by the active range or by sections. In either case these subarrays consisted of the elements referenced by the lowest valued subscript quantities in each dimension, i.e. the origin of these arrays or sections began with the first array element in each dimension. We shall now consider the designation of internal and other general rectilinear subarrays.

The concept of the shifted section allows the user to specify an independent shift in the origin of each dimension of the section. The shift may be any valid real or integer scalar-valued expression. The shift in origin may be either positive or negative. Furthermore, the user may specify the direction or sense in which the subscript quantities of the section are to be incremented.

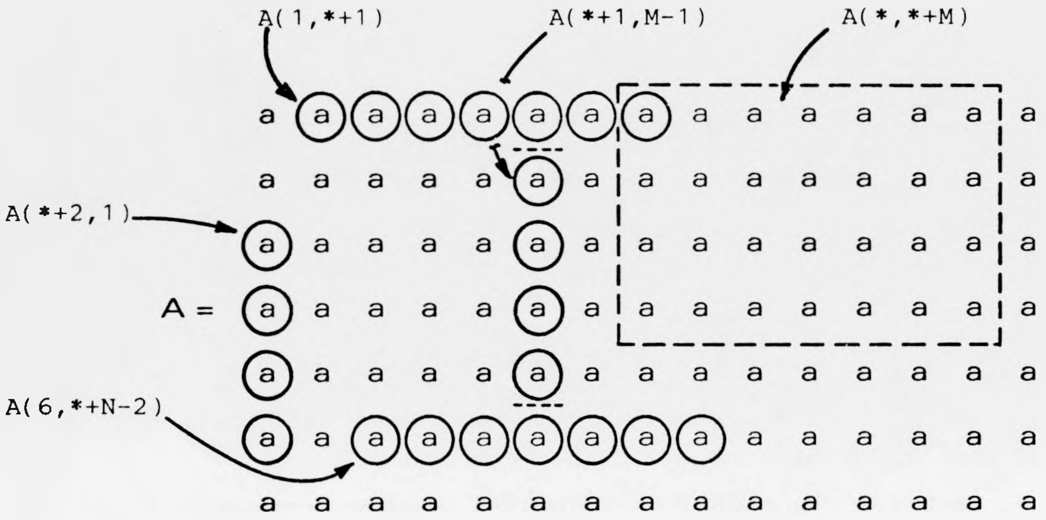
A shifted section is specified by placing an additive expression to the immediate right of the asterisk, subscript selector, specifying the section. This expression must begin with either a plus sign or a minus sign immediately following the asterisk. The shifted section designation is read to mean that the origin (first element of the section in the specified dimension) of the section is to be the first element of the array in the specified dimension of the array, plus or minus the value of the expression. If the expression is real-valued, it is converted (truncated) to integer.

The direction in which the section is to be taken is specified by either a plus or minus sign immediately to the left of the asterisk designating the section. If no sign appears, VECTRAN assumes the positive direction. The user should carefully note the precedence in which the shifted section is determined. Specifically:

1. The section is first determined.
2. The subscript quantities of the section are incremented in the reverse order.

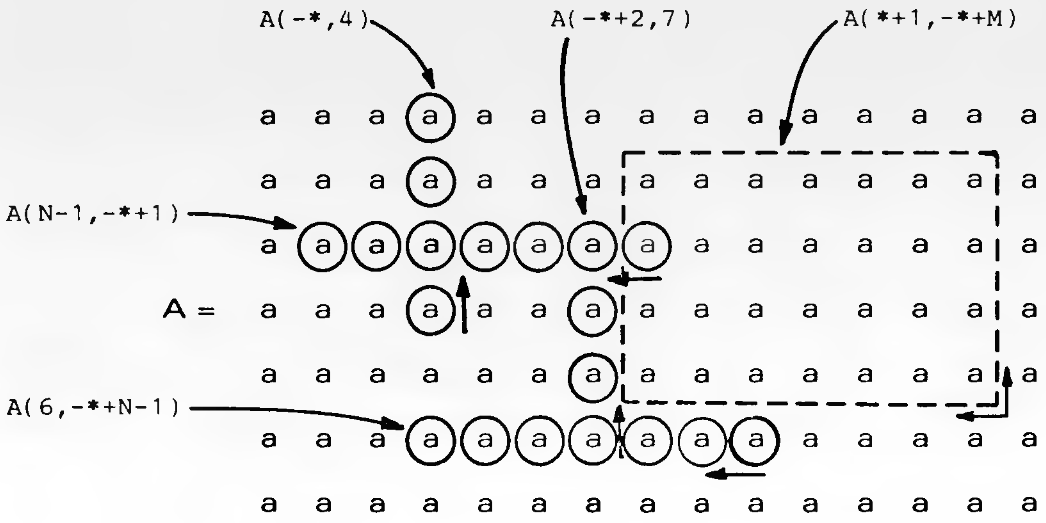
The range of the shifted section, and consequently the object-time range, and the rank of the section are determined in the identical manner as ordinary sections. That is, shifting the origin of the section does not affect rank, range (or object-time range), but only the specific elements of the parent array which are referenced. Again, consider the example above.

```
RANGE/N,M/A(7,15)
M = 7
N = M/2 + 1
```



Now, we shall consider shifted sections in the negative or converse direction. Let us again consider our example.

```
RANGE/N,M/A(7,15)
M = 7
N = M/2 + 1
```



The reader should once again note that the shifted sections illustrated above in each case retain the active range of the matrix A in the free dimension(s) of the section.

VECTRAN does not test to determine whether a shifted section remains within the storage associated with the referenced array.

The concepts of sections and shifted sections combined with the range concept provide the VECTRAN user with a very powerful notational tool by which general rectilinear contiguous subarrays may be directly specified and operated upon. VECTRAN EXPRESSIONS ARE FULLY EVALUATED PRIOR TO ASSIGNMENT OF THE COMPUTED RESULT. Consequently operations between arrays or sections may be carried out, and array-valued results assigned without ambiguity or loss of data integrity.

The following program excerpt is taken from a subroutine which performs the Levinson algorithm [3] for the solution of a system of linear equations in which the matrix of coefficient values has the Toeplitz form. This algorithm is commonly used to compute Weiner match filters.

```
             SUBROUTINE LEVIN(M,F,G,R)
             RANGE /K1/ A(100),R(1),F(1),G(1)
                          .
                          .
                          .
             M1 = M-1
       C     SOLVE LOOP
             DO 30 K = 1,M1
             K1 = K-1
             H = -BETA/ALPHA
             IF(K1) 20,20,10
➡      10    A(*+1) = A(*+1) + H*A(-*+1)
       20    A(K+1) = H*A(1)
             ALPHA = ALPHA + H*BETA
             K1 = K+1
➡            BETA = A .,. R(-*+1)
             Q = (G(K1) - GAMMA)/ALPHA
             F(K1) = Q*A(1)
             K1 = K
➡            F = F + Q*A(-*+1)
             K1 = K+1
➡            GAMMA = F .,. R(-*+1)
       30    CONTINUE
                          .
                          .
                          .
```

The four lines indicated by the solid right arrows utilize either unsubscripted array names, shifted sections, converse sections or some combination of these array identifiers. (Note, these arrows are utilized purely for illustration herein, and are not part of the input syntax.) The first such line, line 10, utilizes both the shifted section A(*+1) and its converse section A(-*+1) in the same expression, and stores the resulting vector back into A(*+1). This line offers both an excellent example of the use of shifted and converse sections and of the importance of maintaining atomicity in the evaluation of array-valued expressions. In this example, a vector temporary is created to preserve data integrity. The second and fourth lines utilize the vector inner product operator (denoted symbolically by .,.) and involve both unsubscripted array names and converse shifted sections. (The inner product operator is described below.) The third line, like the first, computes the sum of a vector plus a scalar times a vector. These examples not only illustrate the general utility of these array identifiers in computation, but also illustrate the dynamic flexibility allowed by the range concept. In this example the reader should note that not only is the value of the range of these vectors being changed with each pass through the loop, but that it is being re-evaluated during the execution of the loop as well. (The value of the range is specified by the variable K1.) Furthermore, the concept of range allows the conformability of these expressions to be determined fully at compile-time.

C. Vector-Valued Subscripts

Vector-valued subscript quantities are legitimate subscript selectors in VECTRAN. Vector-valued subscripts are incorporated into VECTRAN to facilitate sparse matrix manipulation and to provide a convenient means of indirectly addressing and manipulating arrays via tables.

Vector-valued subscripts are utilized in much the same manner as scalar-valued subscripts. Vector-valued subscripts, however, denote the selection of a set of array elements (in the order specified by the vector subscript selector) rather than a specific individual array element. Note, an array subscripted by vector-valued subscript quantities cannot be used directly as an argument to a subprogram because of the indexing procedures invoked to address the array. Expressions containing arrays with vector-valued subscripts may be utilized as arguments to subprograms, and both the array name and vector subscripts may be passed as separate arguments to a subprogram which may then utilize the vector subscripts to reference the array.

Suppose Z is a matrix of active range 5x7, and U and V are vectors of active range 3 and 4, respectively. Furthermore, assume:

```
U = 1 3 2
V = 2 1 1 3
```

then, Z(3,V) consists of the elements of the third row of Z in the order:

```
Z(3,2) Z(3,1) Z(3,1) Z(3,3),
```

and Z(U,2) consists of the column elements:

```
Z(1,2) Z(3,2) Z(2,2),
```

and finally Z(U,V), of the elements:

```
Z(1,2) Z(1,1) Z(1,1) Z(1,3)
Z(3,2) Z(3,1) Z(3,1) Z(3,3)
Z(2,2) Z(2,1) Z(2,1) Z(2,3).
```

Section selector symbols may also be used with vector subscript selectors, for example, Z(U,*) consists of the elements:

```
Z(1,1) Z(1,2) Z(1,3) Z(1,4) Z(1,5) Z(1,6) Z(1,7)
Z(3,1) Z(3,2) Z(3,3) Z(3,4) Z(3,5) Z(3,6) Z(3,7)
Z(2,1) Z(2,2) Z(2,3) Z(2,4) Z(2,5) Z(2,6) Z(2,7).
```

When a vector-valued subscript is used in a subscripted array, the range list entry which corresponds to that dimension in the subscripted array is the first entry of the range of the vector subscript entity. The corresponding entry in the range of the parent array is ignored. The range list entries of the parent array and the vector subscript quantity need not be named identically. The active range of the vector subscript 'overrides' the active range of the referenced array name.

For example, the array specified by Z(V,U) has active range 4x3, and consists of the elements:

```
Z(2,1) Z(2,3) Z(2,2)
Z(1,1) Z(1,3) Z(1,2)
Z(1,1) Z(1,3) Z(1,2)
Z(3,1) Z(3,3) Z(3,2),
```

whereas the array Z had active range 5x7. In this example the reader should also note that the second and third rows of the matrix Z(V,U) are identical replications as specified by the second and third elements of the vector V.

By the above examples, the reader can easily see that vector-valued subscripts permit the user to access the elements of arrays in any desired order, and indeed the ordering specified need not be one to one with the array referenced. Consequently, the use of vector-valued subscripts permit the referenced array to be augmented by any specified repetition --in any sequence-- of the array elements.

IV. IDENTIFICATION OF SECONDARY ARRAYS

We have thus far considered only designation of contiguous arrays or subarrays and selection of ordered subarrays using subscript selectors. VECTRAN further extends the user's capability to define and manipulate sections of arrays which are regularly ordered, but which may be composed of elements which are non-contiguous or which may belong to skewed sections of the array, (e.g. the diagonal elements of a matrix, etc.), by means of the IDENTIFY statement. The IDENTIFY statement may also be utilized to facilitate implementation of algorithms involving storage mappings for banded matrices, etc.

A. Dynamic Equivalence and Virtual Arrays

The IDENTIFY statement IS AN EXECUTABLE STATEMENT which allows the user to dynamically equivalence an identified secondary ('virtual') array with selected elements of a primary ('real') array or with the elements of another identified secondary array.

This statement is perhaps the most powerful semantic extension to FORTRAN in the VECTRAN language. IDENTIFY allows the user to readily manipulate non-contiguous data, skewed sections and other similar subscripted arrays directly by array name in the same manner as any explicitly declared array.

The IDENTIFY statement consists of four principal parts:

i) A range list declaration for the identified secondary ('virtual') array.
ii) The variable name to be associated with the identified ('virtual') array.
iii) The variable name of the host array, ('real' or 'virtual').
iv) A selection mapping which specifies which elements of the host array are to be equivalenced with the virtual array name, and the subscript order in which these elements are to be referenced in the virtual array.

It is important to note, that neither execution of the IDENTIFY statement nor subsequent reference to the identified array cause duplication or creation of a new array. However, identified array elements may be selected from the host array and stored into any other conformable array via the normal assignment statement. An identified array is an equivalenced array and program reference to either the host array name or the identified array name may be used to change the value in main storage of the array elements.

The IDENTIFY statement differs in two important aspects from the EQUIVALENCE statement.

i) The EQUIVALENCE statement specifies a static equivalence relation which is in effect throughout the program execution. The IDENTIFY statement dynamically establishes at its point of execution an equivalence relation which may be altered through the course of the program execution. Indeed reference to an identified array invokes the mapping parameters last associated with the identified array name in order of program execution. (The rank of an identified array, however, must remain the same throughout the program unit.)

ii) The EQUIVALENCE statement establishes an equivalence between arrays, array elements or scalar variables with respect to a 'fixed point' or element of the array. All other elements are equivalence by virtue of their position relative to this element. The IDENTIFY statement may be utilized to identify the virtual array name with selected elements of the host array. The host array may be an explicitly declared array in storage or another identified array.

B. Index Mapping Parameters and the Subscript Mapping Mechanism

Prior to defining the IDENTIFY statement, it is convenient to first define and discuss addressing procedures invoked by VECTRAN to reference arrays [4] . VECTRAN stores arrays in ascending storage locations in main storage with the value of the first subscript quantity of an array increasing most rapidly, and the value of the last subscript quantity increasing least rapidly.

Consequently given a primary array A of rank S, dimensioned $N_1 x N_2 x \ldots x N_S$, and given that the length of each element of A is L bytes; the address of the array element $A(I_1, I_2, \ldots, I_S)$ is given by:

$$\begin{aligned} \text{ADDR}\ (A(I_1, I_2, \ldots, I_S)) &= BA + L^*((I_1-1) + (I_2-1)^*N_1 + (I_3-1)^*N_1{}^*N_2 + \ldots) \\ &= BA + (I_1{}^*M_1 + I_2{}^*M_2 + I_3{}^*M_3 + \ldots + I_S{}^*M_S) \\ &\quad - (M_1 + M_2 + \ldots + M_S) \end{aligned}$$

Where:

BA is the byte address of the first element of A.

$(M_1, M_2, \ldots, M_S)$ is a set of multipliers determined from the storage allocation (dimension) of A, and the length in bytes of each element of A, (i.e., $M_1 = L$, $M_2 = L^*N_1$, $M_3 = L^*N_1{}^*N_2$, ..., etc.).

The byte address, BA, and the set of multipliers $(M_1, M_2, \ldots, M_S)$ are the index mapping parameters for the array A. The function ADDR is the subscript mapping function for A.

Let B be a secondary array of rank R, which is identified as a 'regular', rectilinear section of A. (By 'regular', we mean that for each dimension, the elements of B have equal spacing.) The array B may also be described by a subscript mapping function:

$$\begin{aligned} \text{ADDR}\ (B(J_1, J_2, \ldots, J_R)) &= BA' + (J_1{}^*M_1' + J_2{}^*M_2' + \ldots + J_R{}^*M_R') \\ &\quad - (M_1' + M_2' + \ldots + M_R') \end{aligned}$$

The multipliers M_j' are no longer as simply related to dimension data as in the former case however.

Upon execution of the IDENTIFY statement, the selection mapping specifying the equivalenced elements is examined and the index mapping parameters for the identified array are evaluated, stored and associated with the virtual array name. Subsequent program references to the identified array name invoke the above array indexing procedures using these parameters. Clearly, the host array may in turn be an identified array as this process is simply a transformation of the index mapping parameters.

Note, however, from an implementation point of view, an explicit vector of multipliers need only exist at object-time for certain types of arrays (clearly for identified arrays, although the vector need not exist as a contiguous entity). In general, index mapping vectors need only exist for array-valued parameters, array-valued function names and identified arrays.

C. The IDENTIFY Statement

General Form

IDENTIFY $/n_1,n_2,...,n_N/$ $v(i_1,i_2,...,i_N) = r(m_1,m_2,...,m_M)$

Where:

v is the identified or 'virtual' array name and has rank N.

r is the host array name and has rank M.

$n_1,n_2, ...,n_N$ is a set of one through seven unsigned integer constants or integer variables, (not array elements), separated by commas, representing the range list to be associated with the 'virtual' array v.

$i_1,i_2, ...,i_N$ is a set of one through seven distinct integer variables, (not array elements), separated by commas, representing the list of dummy subscript variables defining the ordinal order of the mapping of v onto a subset of r.

$m_1,m_2, ...,m_M$ is a set of one through seven valid scalar subscript quantities, separated by commas, defining the mapping of v onto r in terms of the set of dummy subscript variables, $i_1,i_2, ...,i_N$.

Only scalar-valued subscript selectors are valid subscript selectors in the IDENTIFY statement. Vector-valued subscripts and section selector symbols are invalid. (Sections and replications may be specified by scalar-valued subscripts.)

The identified array name, v, must not have been previously declared as an array name except possibly in another IDENTIFY statement in the program unit, nor may v and r be the same name. The identified array may be of a different type than the host array, however, the host array and the identified array must agree in element length.

The identified array name may appear undimensioned in a type declaration or RANGE statement, but must not appear in a DIMENSION, COMMON, EQUIVALENCE or DATA statement. The first appearance of an identified array name in an executable statement must be on the left-hand-side of the equals sign in an IDENTIFY statement.

The first occurrence (in statement order) of the identified array name in an IDENTIFY statement defines its rank throughout the program. The rank is implicitly declared by the number of entries in the range list, $(n_1,n_2, ...,n_N)$. Note, the identified array name may appear in a RANGE statement (undimensioned) only after its first appearance in an IDENTIFY statement. The rank declared for v in any subsequent use must equal the initial declared rank of v, although any or all of r, $(m_1,m_2, ...,m_M)$, $(n_1,n_2, ...,n_N)$ and $(i_1,i_2, ...,i_N)$ may be different in any subsequent IDENTIFY statement. Further, the subscripts $(i_1,i_2, ...,i_N)$ are utilized only in a formal sense; values assigned to the variables, (names), elsewhere in the program unit are unaltered by the occurrence of these variables, (names), in the IDENTIFY statement.

It should be carefully noted that the scalar-valued subscript quantities appearing in $(m_1,m_2, ...,m_M)$ should be linear expressions of the implicit dummy subscript variables, $(i_1,i_2, ...,i_N)$. This linearity is assumed over the integers after evaluation (conversion) of expressions appearing as subscript quantities. If linearity is not maintained, the resulting identified array may not be the array desired.

The host array, r, may be an explicitly declared 'real' array or a previously identified 'virtual' array.

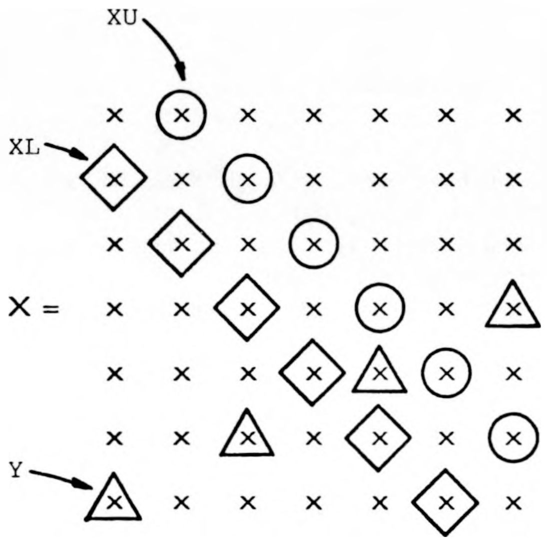
The array r may have at most rank seven; similarly, the rank of v may be at most seven.

The scalar subscript quantities, $(m_1,m_2, ...,m_M)$, along with the dummy subscript variables, $(i_1,i_2, ...,i_N)$, specify the selection mapping of the identified array. The order in which the 'dummy subscript variables' appear in the list, $(i_1,i_2, ...,i_N)$, defines the specific ordering in which the virtual array v is to be defined and referenced. If a dummy subscript variable included in the list $(i_1,i_2, ...,i_N)$ is not utilized in the list of subscript quantities $(m_1,m_2, ...,m_M)$, in general, a replication of the real array r is implied.

Once defined by an IDENTIFY statement, identified arrays are treated the same as any other array in the program unit.

Consider the following examples. The following statements dynamically equivalence the upper and lower diagonals of the [7x7] matrix X as the vectors XU and XL respectively, and equivalence the skew diagonal indicated as the vector Y.

```
DIMENSION X(7,7)
        .
        .
        .
IDENTIFY /K/ XU(I) = X(I,I+1)
IDENTIFY /K/ XL(I) = X(I+1,I)
IDENTIFY /L/ Y(J)  = X(8-J,2*J-1)
        .
        .
        .
```



The following program excerpt is taken from a subroutine which performs the odd-even reduction algorithm [5] for the solution of a tridiagonal system of equations.

```
          SUBROUTINE ODDEVN ( IDIM,N,U,D,L,B,X,IFAIL )
          DIMENSION U( 1 ),D( 1 ),L( 1 ),X( 1 ),B( 1 )
          INTEGER*4 VCTL,VPTR,UPTR,DPRT,LPTR,BPTR,RPRT,XPTR
          REAL*8 L,LOD,LEV,LRM
                         .
                         .
                         .
          VPRT = 0
          RPRT = N
      C   START A NEW ITERATION
      10  UPRT = VPRT
          DPRT = VPRT
          LPRT = VPRT
          IDENTIFY /VCTL/ UOD( K ) = U( UPTR+2*K-1 )
          IDENTIFY /VCTL/ DOD( K ) = D( DPTR+2*K-1 )
          IDENTIFY /VCTL/ LOD( K ) = L( VPTR+2*K+1 )
          IDENTIFY /VCTL/ BOD( K ) = B( VPTR+2*K-1 )
          IDENTIFY /VCTL/ BDD( K ) = B( VPTR+2*K+1 )
          IDENTIFY /VCTL/ UEV( K ) = U( VPTR+2*K )
          IDENTIFY /VCTL/ DEV( K ) = D( VPTR+2*K )
          IDENTIFY /VCTL/ LEV( K ) = L( LPTR+2*K )
          IDENTIFY /VCTL/ BEV( K ) = B( VPTR+2*K )
          IDENTIFY /VCTL/ URM( K ) = U( RPRT+K )
          IDENTIFY /VCTL/ DRM( K ) = D( RPRT+K )
          IDENTIFY /VCTL/ LRM( K ) = L( RPRT+K+1 )
          IDENTIFY /VCTL/ BRM( K ) = B( RPRT+K )
      C   DIVIDE ALL THE ODD-NUMBERED ROWS BY THE PIVOTS
  ➡       BOD = BOD/DOD
          VCTL = VCTL - 1
  ➡       UOD = UOD/DOD
          DPRT = DPRT + 2
          IDENTIFY /VCTL/ DOD( K ) = D( DPTR+2*K-1 )
  ➡       LOD = LOD/DOD
      C   UPDATE EVEN-NUMBERED DIAGONAL ELEMENTS
  ➡       DRM = DEV - UOD*LEV - UEV*LOD
      C   UPDATE EVEN-NUMBERED ELEMENTS IN B
  ➡       BRM = BEV - BOD*LEV - UEV*BDD
      C   CHECK IF VCTL IS EQUAL TO 1
          IF ( VCTL .EQ. 1 ) GO TO 50
          VCTL = VCTL - 1
      C   CREATE NEW SUPER-DIAGONAL
          UPRT = UPRT + 2
          IDENTIFY /VCTL/ UOD( K ) = U( UPTR+2*K-1 )
  ➡       URM = UOD * UEV
      C   CREATE NEW SUB-DIAGONAL
          LPRT = LPRT + 2
          IDENTIFY /VCTL/ LEV( K ) = L( LPTR+2*K )
  ➡       LRM = LOD * LEV
      C   RESET ALL VECTOR POINTERS; CHECK AND RESET LENGTH
          VPRT = RPRT
          RPRT = VPRT + VCTL + 1
          LCTL = VCTL/2
          IF ( VCTL .GT. 2*LCTL ) GO TO 40
          VCTL = LCTL + 1
          GO TO 10
                         .
                         .
                         .
```

In this example the IDENTIFY statement is utilized to define the odd and even elements of the main, upper and lower diagonals at each iteration. With these definitions, the vector structure of the algorithm is readily recognized and coded. The lines designated by the solid right arrows again specify vector operations. Note, the program assumes the input vectors are dimensioned two N in length. The results of each iteration are stored end to end as the pointer variables and IDENTIFY statements are updated.

V. OPERATORS AND FUNCTIONS

As indicated above among the functional requirements set forth for VECTRAN was to provide array-valued functions and operators and to extend the operator set and library functions of FORTRAN to provide operators/functions for matrix computation and reductions. As a first step in this direction VECTRAN extends the semantics of all existing FORTRAN operators and library functions (e.g., SIN, COS, LOG, etc.) to imply that the operation or function is applied distributively element by element to array-valued arguments. That is, all existing FORTRAN operators and library functions are defined to be elemental in VECTRAN.

VECTRAN goes beyond this first step, however, and provides a number of new operators and library functions for:

i) arithmetic and logical operations between array-valued arguments
ii) manipulation of arrays within expressions
iii) reduction operations
iv) utility functions

as well as allowing the user to define array-valued function subprograms with scalar and/or array-valued arguments.

A. New Arithmetic Operators and Functions

Among the new array arithmetic operators provided in VECTRAN are the matrix multiply operator (denoted .*.) and Hermitian vector inner product operator (denoted .,.). These new operators have been defined and implemented to accumulate all intermediate inner product evaluations in twice the arithmetic precision of the greater of the two input operand data types (lengths) to preserve numerical accuracy. The general syntax of FORTRAN, however, requires that the length of the result of a dyadic operation be equal to the greater of the length of its input operand data types; therefore upon completion, these operators convert the resulting inner product values to the precision of the maximum operand length.

Since it is frequently desirable numerically to preserve 'double precision accumulation' of an inner product (matrix product) with the sum or difference of an initial value, VECTRAN provides library functions for these operations as well. These functions GMXP, (generalized matrix multiply), and GHIP, (generalized Hermitian inner product), respectively, allow initial values as third operands and provide not only 'double precision accumulation' with the initial value, but also afford the user a more flexible control of both the intermediate computational precision and the final precision of the result.

In addition to these functions, VECTRAN provides a third library function, GLAP (generalized linear algebraic product). Most frequently in numerical linear algebraic algorithms involving complex matrices, the true Hermitian inner product of complex vectors is not desired. Rather, the sum reduction of the element by element complex products of the vector arguments is desired. (The Hermitian inner product takes the complex conjugate of one of the complex operands to preserve the metric of the inner product operator over the complex vector space.) Refer to Wilkinson [6] . VECTRAN accomodates the numerical linear algebraist by providing an alternate entry point to GHIP, GLAP, which avoids the taking of the complex conjugate twice in this application. GLAP and GHIP are identical for INTEGER and REAL arguments.

B. New Manipulation Operators and Functions

Among the new array manipulation operators and functions provided in VECTRAN are operators/functions for matrix transposition, vector/subarray extraction and the creation of index vectors.

Operators are provided for both transposition and Hermitian transposition of real and complex matrices. These operators (denoted .T. and .H. , respectively) are suffix operators and do not recopy data when used in expressions unless an explicit assignment of the transposed array value is made. Library function equivalents are also provided for these operators.

Several array extraction library functions are provided including ROW, COL and SECT. ROW and COL provide for row (column) extraction from matrix-valued expressions. SECT provides for general section extraction from array-valued expressions. These functions are primarily useful when the user wishes to preserve (compute) only the array value specified by the extracted section from an array-valued expression for which he does not wish to explicitly allocate permanent storage. These functions supplement the section concept, allowing the user to use the specified array-valued result abstracted from an array-valued expression (input argument) for which the compiler dynamically allocates temporary storage.

The VECTRAN library function INDEX is provided as a convenience to the user; it may be used to generate index vectors with an arbitrary increment between successive elements. This function is somewhat analogous to the APL iota function, but allows for arbitrary starting, ending and increment values.

C. New Reduction Operators and Functions

Among the new reduction operators and functions are arithmetic and logical reduction operators and functions to determine algebraic maximum/minimum values and the locations of these values within arrays.

The arithmetic reduction operators are addition (denoted +/), subtraction (denoted -/), multiplication (denoted */) and division (denoted //). The logical reduction operators are AND (denoted .AND./), OR (denoted .OR./) and Exclusive OR (denoted .XOR./). Each of these operators operate on arrays of arbitrary rank producing a single scalar-valued result.

VECTRAN provides the library functions RMAX and RMIN to determine the algebraic maximum (minimum) scalar value contained within an array-valued argument.

The VECTRAN library functions MAXEL and MINEL are provided to determine the location of the algebraic maximum (minimum) value contained within an array-valued argument. Both MAXEL and MINEL return a vector-valued result, a subscript vector locating the first occurrence of the maximum (minimum) element in the array.

D. New Utility Functions

Two new utility functions are provided in VECTRAN, namely LIMIT and RANGE.

LIMIT allows the user to dynamically determine at object-time the current values of the range variables associated with an array identifier or array-valued expression.

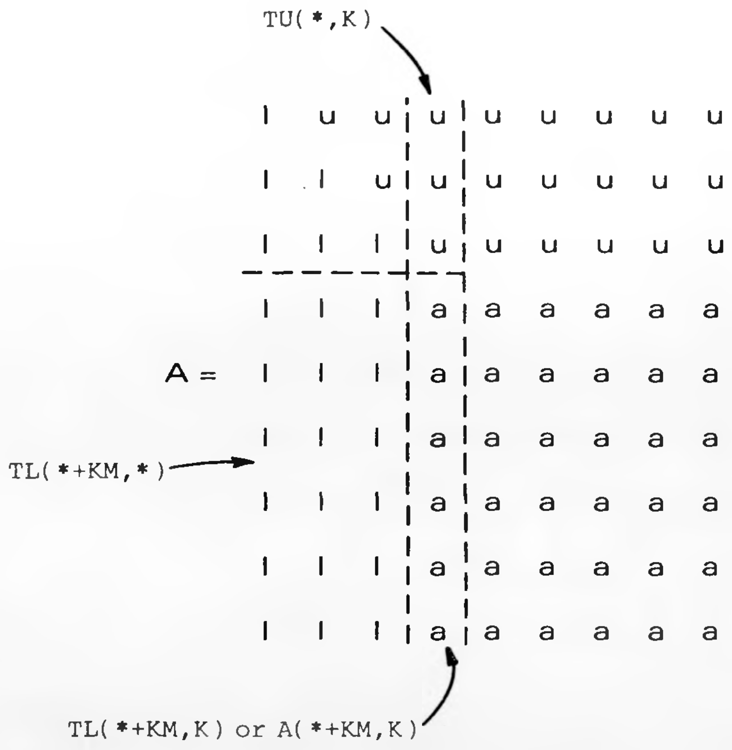
RANGE allows the user to override the range of its array-valued argument to force conformability within a particular expression.

These two functions are analogous to the FORTRAN functions which specify mode conversion, e.g., DBLE, REAL, etc.; LIMIT and RANGE, however, specify the range of 'shape' data.

The following program excerpt is taken from a subroutine which computes the LU matrix decomposition of an asymmetric matrix using the Crout algorithm [7]. This program illustrates the use of several of the operators and library functions described above.

```
          SUBROUTINE DECOMP ( IDIM,N,A,CHG,DET,IED,IFAIL )
          RANGE /L1,L2/ A( IDIM,1 )  /L1/ CHG( 1 )
                               ⋮
      C   TL ( TU ) IS THE LOWER ( UPPER ) TRIANGULAR FACTOR RESPECTIVELY.
      C   VIRTUAL ARRAY TL IS CREATED ONLY FOR CLARITY IN EXPRESSIONS.
      C   TU IS NEEDED TO PROVIDE CONFORMABILITY IN RANGE.
          IDENTIFY /L1,L2/ TL( I,J ) = A( I,J )
          IDENTIFY /L2,L1/ TU( I,J ) = A( I,J )
                               ⋮
      C   FACTORIZE LOOP
          DO 80 K=1,N
          KM = K-1
          L1 = NP-K
          L2 = KM
      C   FORM K'TH COLUMN OF MATRIX L.  THE FIRST COLUMN OF L IS
      C   THE FIRST COLUMN OF A.  THE PIVOTAL ELEMENT IS A( L,K ).
          IF ( K .EQ. 1 ) GO TO 25
➡         TL( *+KM,K ) = A( *+KM,K ) - TL( *+KM,* ) .*. TU( *,K )
      C   LOOK FOR EQUILIBRATED PIVOT.
➡     25  L = KM + MAXEL( ABS( CHG( *+KM ) * TL( *+KM,K ) ) )
      C   PERFORM INTERCHANGE IF NECESSARY.
          CT = CHG( L )
          IF ( L .EQ. K ) GO TO 30
          DET = -DET
          L2 = N
➡         A( K,* ) == A( L,* )
          CHG( L ) = CHG( K )
      30  CHG( K ) = L
      C   CHECK FOR SINGULARITY AND UPDATE DETERMINANT.
          D = ABS ( TL( K,K ) )
          IF ( D*CT .LT. DEL ) GO TO 100
          DET = SCALE( D*DET, IED )
      C   FORM K'TH ROW OF U
          IF ( K .EQ. N ) GO TO 80
          L1 = N-K
          IF ( K .GT. 1 ) GO TO 70
➡         TU( 1,* ) = ( 1./D ) * RANGE( A( 1,* ),/L1/ )
          GO TO 80
      70  L2 = KM
      C   THIS EXPRESSION INVOLVES TRANSPOSES BECAUSE 1-DIMENSIONAL
      C   SECTIONS ARE INTERPRETED TO BE COLUMN VECTORS.
➡         TU( K,*+K ) = ( RANGE( A( K,*+K ),/L1/ ) - TU( *,*+K ).T. .*. TL( K,* ) ) / D
      80  CONTINUE
                               ⋮
```

The following figure illustrates the computation of the 4'th column of the lower triangular factor for the case in which A is an [9x9] matrix. This calculation is performed by the first line desigated by a solid arrow in subroutine DECOMP. Note, in this case K = 4 and KM = 3. Note the use of the matrix multiply operator.

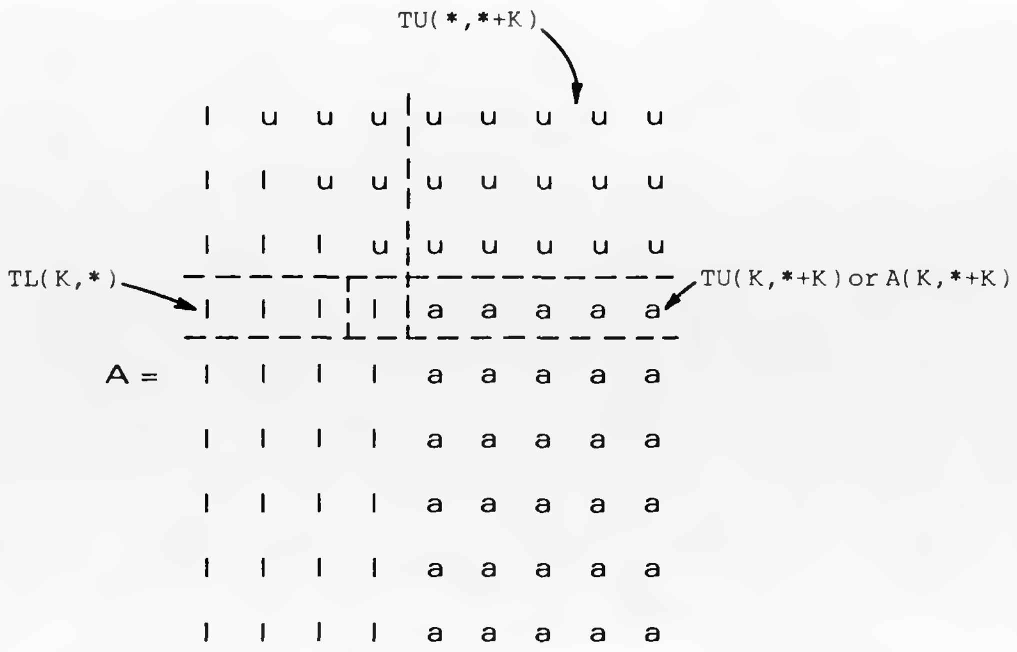


The second line designated by an arrow illustrates the use of MAXEL to determine the location of the maximum element of its vector-valued expression.

The third line designated by an arrow is a new type of assignment statement provided by VECTRAN called an exchange statement, (denoted by the double equal sign). The exchange statement causes reciprocal assignment to take place with data conversion in both directions if indicated.

The fourth and fifth lines designated by arrows are alternate expressions for the computation of K'th row of the upper triangular matrix, TU. The first expression is used only for the case K = 1. These statements employ the RANGE function to override the declared range of A, because all one dimensional sections in VECTRAN are defined to be column vectors. Note also the use of the suffix transpose operator in the latter expression. The matrix operator (.*.) is used in the examples presented herein primarily for clarity of exposition; the GMXP function would provide better numerical accuracy, particularly in DECOMP.

The following figure illustrates the calculation of the 4'th row of the upper triangular factor for the case in which A is an [9x9] matrix. This figure illustrates the computation performed by the last line designated by an arrow; for the case shown, K = 4 and KM = 3.



VI. ASSIGNMENT-LIKE STATEMENTS

VECTRAN provides several new statement types which are not present in FORTRAN. The first class of statements, called 'conditional assignment statements,' provide the user with a convenient way of assigning values to specified subsets of elements of an array within the context of array manipulations and computations. The second class of statements provides a direct means for the dynamic compression/expansion, merging/splitting and restructuring of arrays. These instructions are particularly useful in the manipulation of sparse arrays via logical truth tables or logical-valued expressions.

Together these two classes make up a new general type of statement which we shall call "assignment-'like' statements" because of their similarity and relationship to the assignment statement.

A. Conditional Assignment Statements

There are two new statement types in this class, the WHEN statement and the AT statement. These statements may be thought of as extensions of the logical IF statement. Since array-valued expressions may, or may not be elemental by nature, two basic forms are suggested. The WHEN statement is defined to handle the more general case involving non-elemental or transformational expressions. The AT statement is defined to handle the simpler case of elemental expressions. The AT statement is not strickly required, but is provided as a user convenience and for performance reasons. As we shall see, however, its presence uniquely extends the power of VECTRAN.

The syntax of the WHEN statement is illustrated below.

General Form

WHEN (logexpr) a = [expr_1] [, OR [b] = expr_2]

Where:

a and b are the names of arrays, or subscripted arrays. b is optional, but if present in the WHEN OR form, b must be conformable with a. Note, a and b may be the same entity.

expr_1 and expr_2 are arithmetic or logical expressions conformable with a. expr_1 is optional in the WHEN OR form, but if omitted, the equal sign following a must be followed by the comma.

Note, both expr_1 and b may not concomitantly be omitted in the same statement.

logexpr is a logical expression conformable with array a.

In the execution of this statement:

The entire expression logexpr is first evaluated.

Then, in the event the 'OR' condition is present, the expressions expr_1 and expr_2 are entirely evaluated, and for each element of logexpr which has the value .TRUE. (.FALSE.) the corresponding element of the result of expr_1 (expr_2) is assigned to the corresponding element of a (b).

Or, in the event the 'OR' condition is not present, the expression expr_1 is entirely evaluated, and for each element of logexpr which has the value .TRUE., the corresponding element of expr_1 is assigned to the corresponding element of a. The elements of a corresponding to elements of logexpr which have the value .FALSE. are left unchanged.

If expr_1 (expr_2) is a logical expression, then a (b) must be a LOGICAL array. If expr_1 (expr_2) is an arithmetic expression, then a (b) must be an INTEGER, REAL or COMPLEX array. Assignment of the values of expr_1 (expr_2) into the array a (b) follows the same rules of conversion as an assignment statement.

If logexpr is a scalar variable, or an array or subscripted array, its value(s) may not be changed by any function reference during the course of the evaluation of the expression expr_1 (expr_2), or in the evaluation of subscript quantities of a (b).

The WHEN statement should only be used when one or more of the expressions logexpr, expr_1 or expr_2 cannot be evaluated on an element by element basis. That is, when the value of each element of the result of logexpr or expr_1 or expr_2 may not be evaluated independently of the computation required to compute the values of the other elements of the result.

Now consider the AT statement.

General Form

AT (logexpr) a = [expr_1] [, OR [b] = expr_2]

Where:

a and b are the names of arrays or subscripted arrays. b is optional, but if present in the AT OR form, b must be conformable with a. Note, a and b may be the same entity. expr_1 and expr_2 are arithmetic or logical scalar-valued expressions, or elemental array-valued expressions conformable with a. expr_1 is optional in the AT OR form, but if omitted, the equal sign following a must be followed by the comma.

Note, both expr_1 and b may not concomitantly be omitted in the same statement. logexpr is a logical scalar-valued expression, or an elemental array-valued expression conformable with a.

In the execution of this statement:

In the event the 'OR' condition is present, the expression logexpr is evaluated element by element in subscript order, and the expression expr_1 (expr_2) is evaluated and assigned to the corresponding array position of a (b) if the element of logexpr has the value .TRUE. (.FALSE.).

Or, in the event the 'OR' condition is not present, the expression logexpr is evaluated element by element in subscript order, and for each element of logexpr which has the value .TRUE., the expression expr_1 is evaluated and assigned to the corresponding array position of a, otherwise the element of a remains unchanged and no evaluation of expr_1 is carried out.

If expr_1 (expr_2) is a logical expression, then a (b) must be a LOGICAL array. If expr_1 (expr_2) is an arithmetic expression, then a (b) must be an INTEGER, REAL or COMPLEX array. Assignment of the values of expr_1 (expr_2) into the array a (b) follows the same rules of conversion as an assignment statement.

This statement differs from the WHEN statement in that although the assignment of each element of expr_1 (expr_2) is conditional on the corresponding value of logexpr, expr_1 (expr_2) is evaluated only for those elements corresponding to true (false) values of logexpr. Consequently, expr_1 (expr_2) must be such that evaluation of each element of their respective results may be individually and independently evaluated. Similarly, the expression logexpr must also be computable element by element.

To illustrate the difference between the WHEN and the AT statements note the differences in the order of the resulting data storage illustrated by the following example. Assume V is a vector of eight elements, and the range of V is five. Initially V consist of:

$$v_1\ v_2\ v_3\ v_4\ v_5\ v_6\ v_7\ v_8$$

Then the statement:

```
WHEN ( .TRUE. ) V( *+1 ) = V
```

implies that upon completion of execution of the statement, V consist of:

$$v_1\ v_1\ v_2\ v_3\ v_4\ v_5\ v_7\ v_8$$

Whereas, the statement:

```
AT ( .TRUE. ) V( *+1 ) = V
```

implies that upon completion of execution of the statement, V consist of:

$$v_1\ v_1\ v_1\ v_1\ v_1\ v_1\ v_7\ v_8$$

Thus data integrity is preserved on an array basis in the execution of the WHEN statement, and on a scalar basis in the execution of the AT statement.

As an immediate consequence of element by element evaluation and storage of array-valued expressions by the AT statement, iterative evaluation of expressions with data feedback is possible within a single VECTRAN statement. This is illustrated by the following program excerpt taken from a subroutine which performs the forward and backward sweeps of a decomposed tridiagonal system of equations using the standard Gaussian algorithm.

```
      SUBROUTINE TRISOL ( AA,X,B,N )
      RANGE /M/ X( 1 ),B( 1 )  /M,3/ AA( *,* )
C     ASSUME
C     AA( *,1 ) = LOWER DIAGONAL OF MATRIX L, ( AA( 1,1 ) = 0.0 )
C     AA( *,2 ) = MAIN DIAGONAL OF MATRIX L
C     AA( *,3 ) = UPPER DIAGONAL OF MATRIX U, ( AA( N,3 ) = 0.0 )
                      .
                      .
                      .
      M = N-1
C     PERFORM FORWARD SUBSTITUTION
      X( 1 ) = B( 1 )/AA( 1,1 )
      AT ( .TRUE. ) X( *+1 ) = ( B( *+1 ) - X( * ) * AA( *+1,1 ) ) / AA( *+1,2 )
                      .
                      .
                      .
C     PERFORM BACKWARD SUBSTITUTION
      AT ( .TRUE. ) X( -* ) = X( -* ) - X( -*+1 ) * AA( -*,3 )
                      .
                      .
                      .
```

In this example the first AT statement performs the forward solution of the lower triangular factor of the tridiagonal matrix. The use of shifted sections coupled with the element by element evaluation and assignment causes the computed results to be fed back iteratively in the calculation. The second AT statement similarly performs the backward solution by utilizing converse sections.

The reader should carefully note that although the array syntax of VECTRAN is utilized in this example, that the computation is performed in scalar mode.

B. Assignment with Array Expansion/Compression

As mentioned above, the second class of "Assignment 'Like' Statements" are those statements for expanding/compressing, spliting/merging and restructuring arrays under logical control. There are two new statements types in this class: PACK and UNPACK. Each of these new statements has conditional forms related to the WHEN and AT statements defined above.

The syntax of the PACK statement forms are illustrated below.

General Form

```
PACK /k/ a = expr1
PACK /k/ WHEN (logexpr) a = [expr1] [, OR [b] = expr2]
PACK /k/ AT     (logexpr) a = [expr1] [, OR [b] = expr2]
```

Where:

a and b are the names of arrays of rank one, or vector-valued sections of arrays. b is optional, but if present in the PACK WHEN OR or PACK AT OR form, b must be conformable with a. Note, a and b may be the same entity.

$expr_1$ and $expr_2$ are conformable arithmetic or logical scalar or array-valued expressions of arbitrary rank. $expr_1$ is optional, but if omitted in the PACK WHEN OR or PACK AT OR forms, the equal sign following a, must be followed by the comma.

logexpr is a logical scalar or array-valued expression conformable with $expr_1$, ($expr_2$).

In the case of the PACK AT form of this statement logexpr, $expr_1$ and $expr_2$ must be either scalar-valued or elemental array-valued expressions.

k is the name of an integer scalar variable or integer subscripted variable. The count of the number of elements assigned to a is assigned to k upon completion of execution of the PACK statement. Note, k is not the range of a unless so declared by another statement such as the RANGE statement.

If $expr_1$ ($expr_2$) is a logical expression, then a (b) must be a LOGICAL array. If $expr_1$ ($expr_2$) is an arithmetic expression, then a (b) must be an INTEGER, REAL or COMPLEX array. Assignment of the values of $expr_1$ ($expr_2$) into the array a (b) follows the same rules of conversion as an assignment statement.

The unconditional form of the PACK statement causes the evaluation of the array-valued expression $expr_1$, and the values of $expr_1$ to be assigned in subscript order to the vector a. If one of the conditional forms of the statement is used, then the logical expression logexpr is evaluated and governs the assignm^nt of the values of $expr_1$ and $expr_2$ into the arrays a and b in a manner analogous to the WHEN and AT statements. Values of $expr_1$ corresponding to true values of logexpr are assigned to a, and in the event the 'OR' condition is present -- values of $expr_2$ corresponding to false values of logexpr are assigned to b. Values of $expr_1$ ($expr_2$) corresponding to false (true) values of logexpr are ignored. In either case, the number of values assigned to a is counted, and upon completion of the PACK statement this count is assigned to the variable k.

Consider the following example, to merge selected elements from two arrays into a single vector. Suppose the matrices IA and IB are as shown:

$$IA = \begin{matrix} 7 & 1 & 4 & 3 \\ 3 & 1 & 2 & 6 \\ 2 & 8 & 5 & 9 \end{matrix}$$

$$IB = \begin{matrix} 4 & 2 & 3 & 6 \\ 8 & 1 & 5 & 4 \\ 1 & 7 & 2 & 8 \end{matrix}$$

The statement

```
PACK/K/AT(IA.GT.4)IV=IA,OR=IB
```

yields the results:

IV = 7 8 1 2 1 8 3 5 5 6 6 9

K = 5

Now consider the forms of the UNPACK statement.

General Form

```
UNPACK a = expr₁
UNPACK WHEN (logexpr) a = [expr₁] [, OR [b] = expr₂]
```

Where:

a and b are the names of arrays or subscripted arrays of arbitrary rank. b is optional, but if present in the UNPACK WHEN OR form, b must be conformable with a. Note, a and b may be the same entity.

$expr_1$ and $expr_2$ are arithmetic or logical array-valued expressions of rank one. $expr_1$ is optional, but if omitted in the UNPACK WHEN OR form, the equal sign following a, must be followed by the comma.

logexpr is a logical array-valued expression conformable with a.

If $expr_1$ ($expr_2$) is a logical expression, a (b) must be a LOGICAL array. If $expr_1$ ($expr_2$) is an arithmetic expression, then a (b) must be an INTEGER, REAL or COMPLEX array. Assignment of the values of $expr_1$ ($expr_2$) into the array a (b) follows the same rules of conversion as an assignment statement.

The unconditional form of the UNPACK statement allows the user to dynamically expand a packed vector onto an array of one or more dimensions. This statement complements the PACK statement. That is, the unconditional form of the UNPACK statement causes the vector-valued expression $expr_1$ to be evaluated and the values assigned to the array a in the subscript order indicated by the range of a. If a conditional form of the UNPACK statement is used, then the expression logexpr is evaluated and governs the assignment of the values of $expr_1$ and $expr_2$ into the arrays a and b analogous to the WHEN statement. Each sequential value of $expr_1$ is assigned -- in subscript order -- to the element of a corresponding to the next true value of logexpr, and in the event the 'OR' condition is present -- each sequential value of $expr_2$ is assigned in subscript order to the element of b corresponding to the next false value of logexpr. Otherwise values of a (b) corresponding to false (true) values of logexpr are left unchanged. Thus, this statement may be used to merge values from two vectors into a common array or to store selected elements of a vector into two arrays under logical control.

We shall consider two examples. First consider, expanding the packed vector IV onto the matrix IU according to the LOGICAL array L.

Assume:

$$L = \begin{matrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{matrix}$$

$$IU = \begin{matrix} x & x & x & x & x & x \\ x & x & x & x & x & x \end{matrix}$$

$$IV = \begin{matrix} 3 & 4 & 7 & 1 & 5 & 2 & 6 & 3 & 8 \end{matrix}$$

Where the x's in IU simply denote original values of IU. Then the statement,

```
UNPACK WHEN ( L ) IU = IV
```

yields the result:

$$IU = \begin{matrix} 3 & x & 4 & x & x & x \\ x & x & 7 & 1 & x & 5 \end{matrix}$$

Second, consider the use of the UNPACK statement to accomplish a threaded merge of two source vectors under logical control.

Assume,

$$L = \begin{matrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{matrix}$$

$$IA = \begin{matrix} 7 & 4 & 3 & 1 & 2 & 8 & 6 & 5 & 0 & 9 \end{matrix}$$

$$IB = \begin{matrix} 3 & 0 & 2 & 7 & 4 & 6 & 5 & 1 & 8 & 9 & 3 & 7 \end{matrix}$$

then the statement,

```
UNPACK WHEN ( L ) IU = IA, OR = IB
```

yields the result:

$$IU = \begin{matrix} 7 & 2 & 3 & 2 & 1 \\ 3 & 7 & 1 & 6 & 8 \\ 0 & 4 & 4 & 5 & 6 \end{matrix}$$

From these examples we note that the PACK and UNPACK statements like the WHEN and AT statements may have any combination of one or two target arrays and one or two source expressions.

VII. SUBPROGRAMS

As stated in the introduction, one of the functional objectives of VECTRAN was to provide the user with a means of defining array-valued function subprograms and to allow the user to utilize array-valued expressions as arguments to subprograms. Let us first consider array-valued arguments.

A. Array-Valued Arguments

In general array identifiers (with the exception of vector-subscripted arrays) and array-valued expressions may be used as arguments to function or subroutine subprograms except for statement functions. This includes all array names (primary or identified) and their sections. Although vector-subscripted arrays may not be utilized as an actual argument, they may be a component of an array-valued expression which is an actual argument, and both the array name and the vector defining the subscripted array may be passed as actual arguments -- thus allowing the subscripted array to be 'recreated' within the subprogram.

B. Extensions to User-Supplied Functions

VECTRAN extends FORTRAN function definitions in two ways:

Firstly, VECTRAN provides a new declaration statement, namely the ELEMENTAL statement, which allows the user to declare that his function subprogram may be applied distributively element by element to array-valued arguments and that the 'result' of this function is an 'array' conformable with its arguments. Note, elemental function subprograms are actually defined as scalar-valued functions which may be applied distributively over conformable array-valued arguments.

Secondly, VECTRAN extends the FUNCTION statement to include the definition of array-valued functions by extending its syntax to include the explicit specification of the assigned range for the array-valued function. Note, when using user-defined array-valued functions, the name of the function must appear in a RANGE statement in the calling program prior to its first usage therein.

C. Extensions to Formal Parameter Specification

Although a VECTRAN-compiled subprogram may generally be referenced by either VECTRAN or FORTRAN program units, some distinct advantages accrue in overhead if the programmer provides structural information regarding array dummy arguments. (FORTRAN subprograms may always be referenced by VECTRAN program units.)

Firstly, in a subprogram if the programmer can guarantee that it will be referenced only by VECTRAN program units and that the actual argument will be an array or an array-valued expression and not a scalar element, then the dimensioning data for array-valued dummy arguments may be replaced by asterisks, indicating only rank. Consider the following example.

```
REAL FUNCTION /M,M/ SIMXFM(X,Y,M)
DIMENSION X(*,*)
RANGE /N,M/ X  /N,N/ Y(*,*)
COMMON N
              .
              .
              .
SIMXFM=X.T. .*. Y .*. X
              .
              .
              .
```

In this example, which applies a similarity transformation to the the matrix Y, note firstly the inclusion of the range information in the FUNCTION statement and secondly the use of asterisks in the DIMENSION and RANGE statements. In the example the defined function SIMXFM will determine the dimensioned storage allocation of the dummy array-valued arguments X and Y at object-time from the calling program. Note, the presence of these asterisks preclude the subprogram being called by a FORTRAN program unit. This mechanism is defined as 'call by mapping only.'

Secondly, in a subprogram if the programmer knows that, for a particular dummy argument, any actual argument will be only the name of an array defined by dimensioning data, (i.e., the array is contiguous), or an array element of such an array, then he may convey this information to the compiler by placing slashes around the dummy argument in the FUNCTION or SUBROUTINE statement thereby specifying 'call by location only.' Information on the storage structure associated with the array is then determined solely from the dimensioning data provided in the subprogram unit, and must consequently correspond to the dimensioning data of the array used as an actual argument.

Lastly, and the more normal situation, a formal parameter may be dimensioned in the usual FORTRAN manner within the subprogram (without slashes surrounding the dummy argument). In this case the mapping information generated internally is utilized when the subprogram is called by FORTRAN program, but this information is overridden when supplied mapping data is passed with the array-valued actual parameter from a VECTRAN calling program.

VIII. CONCLUSION

In conclusion the authors believe VECTRAN not only offers a general extension to FORTRAN without greatly expanding its syntax, but does so while maintaining the general philosophy of FORTRAN and its object-time performance orientation.

The VECTRAN extensions are designed to be implementable on a broad range of machine architectures and do not specifically reflect any particular machine design or preference. VECTRAN may be implemented on scalar, vector or parallel type machines.

The VECTRAN extensions complement automatic vectorization approaches and offer more information about the underlying problem structure to enable higher optimization and vectorization at lower compile-time cost.

Finally, VECTRAN provides for increased programmer productivity for new application development as well as ease of program conversion for existing FORTRAN applications.

Only the major features of VECTRAN are presented herein, for a more complete and detailed exposition of VECTRAN the reader is referred to reference [1], and in particular to Chapters II, IV and V, and Appendix D. The authors would welcome any criticism or remarks concerning VECTRAN, its function, or its applicability.

ACKNOWLEDGMENTS

The authors wish to acknowledge W.A. Murray, L. Manzanera and J.W. Huang who originally wrote the various subroutines used as examples in the paper.

REFERENCES

1. G. Paul and M. Wayne Wilson, "The VECTRAN Language: An Experimental Language for Vector/Matrix Array Processing," IBM Palo Alto Scientific Center report G320-3334 (August 1975).

2. IBM System/360 and System/370 FORTRAN IV Language, form GC28-6515.

3. N. Weiner, Extrapolation, Interpolation and Smoothing of Stationary Time Series, Appendix B by Norman Levinson, John Wiley & Sons, Inc., New York (1949), pp. 129-139.

4. M. Wayne Wilson, "Flexible Subarray Facilities for Classical Programming Languages," IBM Houston Scientific Center report G320-2426 (November 1973).

5. J.J. Lambiotte, Jr., "The Solution of Linear Systems of Equations on a Vector Computer," PhD Dissertation -- University of Virginia (May 1975), pp. 112-152.

6. J.H. Wilkinson, The Algebraic Eigenvalue Problem, Oxford University Press, London (1965), p. 4.

7. J.H. Wilkinson and C. Reinsch, Linear Algebra, Handbook for Automatic Computation Volume II, Contribution I/7 "Solution of Real and Complex Systems of Linear Equations," by H.J. Bowdler et al., Springer-Verlag, Berlin (1971), pp. 93-110.

REGISTER ALLOCATION IN THE SL/1 COMPILER

by

Douglas D. Dunlop*
Computer Sciences Corporation
Hampton, Virginia

and

John C. Knight
NASA Langley Research Center
Hampton, Virginia

ABSTRACT

The problem of allocating the 256 general purpose registers of the CDC STAR-100 is discussed. The algorithm employed by the SL/1 compiler developed at NASA Langley Research Center is described. The algorithm is based on estimation of the effects of different allocations on execution time of a program. Preliminary measurements of the algorithm's performance are given.

I. INTRODUCTION

One important characteristic of the CDC STAR-100 is that it has 256 general purpose registers. All scalar operations take place out of registers and vector descriptors for vector instructions are held in registers. A vector descriptor is just the length and beginning address of a vector. The number of available registers typically exceeds the number required for temporary results and special addresses, and this leads to the idea of keeping frequently used data permanently in the register file. All references to these data items are then made directly to the register file, thus eliminating the need to load and store from memory. These savings can be very important in view of the relatively slow main storage access time of the STAR-100.

*Work performed under NASA Contract Number NAS1-14900.

In a high level language program, a choice has to be made of the most appropriate data items to allocate to registers. One solution is to introduce into the language the notion of the register file through a register declaration or compiler directive. This solution was rejected for SL/1 because it is contrary to the spirit of a high level language, and it has been found that through careful program analysis, the compiler can make an effective selection. In the SL/1 compiler, this selection is based on a "cost" function which, for each candidate estimates the benefits of keeping it in a register. The selection process then maximizes these estimated benefits.

II. THE COST FUNCTION

Let P be an SL/1 program and D be the set of data items in P that are candidates for being permanently allocated to a register. The cost function for a data item $d \in D$ is denoted C(d) and is defined as the cost, in machine cycles, of running P with d stored in main memory less the cost of running P with d stored in a register. A positive value of C(d) indicates that P runs more efficiently with d stored in a register.

Given the C(d) values for all $d \in D$, one could allocate storage optimally as follows:

(1) allocate to a register the $d \in D$ with the largest positive C(d),
(2) delete this item from the set being considered,
(3) repeat (1) and (2) until all available registers are used or there are no more candidates.

The other members of D should be allocated to main storage. Strictly speaking, this algorithm is unrealizable in that the C(d) values can only be known after the program executes. The cost function depends on the program's run-time flow of control which cannot be determined, in general, prior to the execution of the program. The C(d) values may be estimated, however, by a compiler through careful study of the source program. The SL/1 compiler utilizes a variety of techniques to obtain cost function predictions and allocates storage based on these results.

III. ESTIMATING THE COST FUNCTION

An SL/1 program consists of a set of global declarations followed by a set of non-nested procedure definitions. Procedures may contain local declarations. A wide variety of control structures are available in SL/1 and all of them (except the GO TO) operate on a sequence of statements. This leaves the programer with very little need for the GO TO statement which is important because the mechanisms for estimating the cost function rely on detecting program loops by the presence of explicit iterative statements rather than using control flow graphs. Implicit loops constructed by the programer using GO TO statements are not detected.

The cost function is based largely on estimations of statement execution counts. These are predictions of how often a particular SL/1 statement will be executed in a given program. Initially, each procedure is analyzed separately and it is assumed that the procedure will only execute once. In addition, all the statements inside a procedure are given initial statement execution counts of 1. As explicit loops are encountered in the procedure, the execution counts of all of the statements controlled by the loop are multiplied by a factor f.

The factor f is the compiler's estimate of the number of iterations, on average, that the loop will execute. Occasionally, f may be calculated exactly based on the constant nature of the termination criterion and the absence of exits from the loop body. In more general cases, the compiler simply assumes that the loop will iterate a constant number of times. This constant is the same for all loops.

The statement execution count estimations are modified by estimating procedure execution counts. Let V be the set of procedure definitions in the program being compiled. The compiler constructs a directed graph containing elements of V as nodes and arcs of the form $\langle x,y \rangle$ where $x,y \in V$ in the event that procedure x contains a call to procedure y. Associated with each arc $\langle x,y \rangle$ is an estimate of the number of times x will call y. This estimate (written $E(\langle x,y \rangle)$) is the sum of the statement execution counts of the CALL statements calling y from x. For $y \in V$, the procedure execution count of y (written R(y)) is calculated using

$$R(y) = \sum_{x \in SV} (E(\langle x,y \rangle)*R(x))$$

where SV is the subset of V containing procedures with calls to y. The R values can always be calculated provided the graph is acyclic. If the compiler detects a cycle in the graph, a diagnostic is issued to alert the programer to a possibly recursive procedure call (illegal in SL/1) and an R value of 1 is given to the offending procedure. Finally, the statement execution counts of all of the statements in a procedure are multiplied by the R value of that procedure, thus incorporating the R information into the statement execution counts.

Once the statement execution counts have been estimated, the cost function C(d) where $d \in D$ can be calculated. All C(d) values are initially zero. Each reference to a $d \in D$ in the program is examined and the difference in cost between d permanently residing in main storage and permanently residing in a register at that point in the execution is estimated. If the reference to d requires d to be in a register (for example, if it is a scalar involved in an arithmetic operation) C(d) is increased by the cost in machine cycles of loading d into a register multiplied by the statement execution count of the statement involved. Similarly, if the reference to d requires d to be in main storage (for example, if it is a variable passed as a parameter), C(d) is decreased by the cost of storing and reloading d multiplied by the appropriate execution count. The cost of loading d into a register depends on the type of d. For example, a scalar variable must be loaded from main storage which is a relatively slow operation whereas a constant can be created from data in the instruction stream which is relatively fast.

When this process is complete, the cost function in terms of machine cycles has been computed for all quantities in the program which could reside in registers. This information is then used for register allocation.

IV. APPLYING THE ESTIMATED COST FUNCTION

Variables declared globally in an SL/1 program are known to every procedure in the program. Those declared locally are known only to the procedure in which they are declared. Consequently the SL/1 compiler divides the portion of the register file that is available for permanently holding data items into a global section and a local section. The procedure entry code saves the caller's local register section. Local data items of the callee which are permanently allocated to registers are loaded into the local section from memory on procedure entry and are stored back to memory when exiting the procedure. This mass loading and storing of portions of the register file is made relatively efficient by the STAR-100's SWAP instruction.

The existence of the global and local register sections has a complicating influence on the register allocation technique. Each register in the local section can be used to contain a local data item for each procedure. Assume the set of global data items and the sets of local data items for each procedure are sorted in order of decreasing C(d) values. As each register is being allocated, two possibilities exist. If the next global data item has a positive C(d) value which exceeds the sum of the C(d) values of the next local data item from each of the procedures, the global data item is allocated to the register; otherwise, the next local data item with positive C(d) values from each procedure is allocated to the register. The process terminates when the available register set is full or when no data items with positive C(d) values exist. The remaining data items, if any, are allocated storage in main memory. Actual register numbers are allocated following this selection process to ensure that global and local sets are contiguous.

V. DIFFICULTIES WITH REGISTER ALLOCATION

In allocating registers, several problems arise for which there is no easy solution. Four of the more significant are described here. The SL/1 compiler presently makes no attempt to handle them in an optimal way. Temporary "ad hoc" solutions have been implemented and work is continuing to find more effective algorithms.

Certain computations can be recognized at compile time as being constant (for example, the address of an array reference with constant subscripts). These quantities are candidates for residence in a register, but the cost difference between register and memory residence cannot be determined. If the constant computation is not allocated to a register, it can be made available when needed either by loading from memory or by recomputing it. Recomputation is preferred if the necessary operands are in registers, but if not, it is faster to load it from memory. Thus, the choice, which is part of register allocation, cannot be made properly until after register allocation has been completed.

A portion of the register set must be reserved for holding temporary values, etc., during execution. These temporary values include scalar subexpression results and data items not residing permanently in registers. The number of registers required depends upon the maximum number of temporary values that are needed at any one time. This number can be estimated at compile time, but its precise value is influenced by what is permanently residing in registers. Hence, the way in which the register set should be divided up in an optimal way cannot be determined until the registers have been allocated.

In computing C(d) it is not sufficient to process the program linearly, incrementing C(d) for each reference to d. If several references to d occur in one basic block, it may not be necessary to reload d for each reference since it may still be residing in a temporary register from a previous reference.

The way in which d is used is also a factor since if it appears on the left hand side of an assignment, the new value will probably have to be stored even if it is between references in a basic block. Whether the store is needed or not depends on the existence of other assignments in the basic block and how d is managed between assignments. Thus, in order to estimate C(d) as accurately as possible, the program structure in terms of basic blocks, references to d and assignments to d must be known as the estimate is computed.

The STAR-100 is capable of a high degree of parallelism in the execution of scalar instructions. If the unit which executes load and store instructions is not busy and there are no data dependencies, a load or store instruction can be issued and immediately afterwards subsequent scalar instructions can be issued. They will be executed in parallel with the load or store. Clearly, a compiler should take advantage of this by reorganizing instruction sequences in an optimal way. The SL/1 compiler currently does not do this although the situation can easily occur by chance. This complicates the computation of C(d) considerably since what appears to be a cost of keeping d in memory could be partially or completely masked by the parallelism of the hardware.

VI. PRELIMINARY EVALUATION OF THE ALGORITHM

In order to assess the performance of this algorithm, the SL/1 compiler has been modified to output its estimate of the cost functions and to generate instructions which measure the actual cost functions at execution time. Instructions are also produced to count the number of times each line of a program executes.

One measure of the algorithm's efficiency is the proportion of the optimal set of data items which it actually allocated to registers. This can be measured by examining the actual cost data after execution, using this to determine the optimal allocation, and then comparing it with the compiler's selection. The proportion of the optimal set actually allocated is then the efficiency measure.

Clearly, the performance of this algorithm is program dependent and even data dependent for a given program. In order to draw any general conclusions, it is necessary to measure the algorithm's performance over a wide range of programs and average the results. At the time of writing, a complete set of data had been obtained for only one program. The program is 1100 lines long and contains 350 global quantities which could be selected for permanent allocation to a register. There are an insignificant number of local variables. The actual number of registers available is 113. If registers were allocated randomly, the efficiency measure would be 0.32, and this is a lower bound against which the algorithm can be judged. Table I shows the actual efficiencies which were obtained for a range of values of the constant which the compiler assumes for number of iterations of explicit loops.

TABLE I

LOOP CONSTANT

	4	16	64	10000
Efficiency Measure	0.67	0.59	0.59	0.59

TABLE II

ALLOCATION TYPE

	1	2	3
Processor Time (Seconds)	6.15	6.03	6.69

The algorithm will never achieve an efficiency of 1 except by chance, but it is important to examine the compile time estimates to determine how they can be improved. The problem which occurred with the program described above is that it contains a loop which is executed many more times than any of the others. Several variables appear just a few times in the program, but all of the appearances are inside this single critical loop. These variables do not seem important to the algorithm, and so they are incorrectly allocated to memory.

Perhaps the most important measure of the algorithm is its effect on total program running time. The program described above was compiled and executed three different ways. They were:

(1) normal operation of the algorithm,

(2) optimal allocation performed by reading in the actual cost data from a previous run,

(3) nothing permanently allocated to registers. A quantity will remain in a register for the duration of a basic block only.

The measured processor times are shown in Table II. The difference between cases (1) and (3) is only about 10% which is initially surprising. The reasons are that the program is well written, it uses mostly vector instructions, and it uses long vectors. These have the effect of largely swamping the overhead introduced by the additional load and store operations. This effect will vary considerably from program to program. It can be seen from Table II that of the benefit to be gained in this case by register allocation, most has been obtained by the practical algorithm.

VII. CONCLUSION

An algorithm has been presented for management of the 256 general purpose registers of the STAR-100. The algorithm has been implemented in the SL/1 compiler and is in production. Visual inspection of its actions and early results of experiments indicate that it is performing well.

The algorithm is quite complicated, requires considerable code in the compiler, and adds a small, but non-trivial, amount to the compile time of a program. Is it worth the effort? It is reasonable to consider algorithms which are simpler and use much less information about the program in making the allocation decision. It may transpire that near optimal register allocations can be made this way. Carrying this a step further, random allocation or allocation of the first variables declared may be efficient.

In the other direction, there may be cases where greater analysis of the program than described here will make better choices. Very small changes in the set of data allocated to registers may have a significant effect on programs with complicated loop structures performing many iterations.

As well as being appropriate to the STAR-100, it is felt that these techniques may have relevance to machines with similar structures. For example, allocation of the B and T registers on the CRAY-1 is a very similar problem to that described for the STAR-100.

ACTUS: A LANGUAGE FOR SIMD ARCHITECTURES

by

Ron Perrott
Department of Computer Science
The Queen's University
Belfast, N. Ireland

David Stevenson
Institute for Advanced Computation
NASA-Ames Research Center
Moffett Field, California 94035

ABSTRACT

A new language has been defined at the Institute for Advanced Computation, NASA-Ames Research Center, aimed specifically for lock-step parallel computers such as the Illiac IV, STAR-100 and Cray-1. The two major dsign goals in defining the language were i) that it would represent the state-of-the-art in language design with respect to support for developing reliable programs and ii) that it would permit codes developed on one parallel architecture to be moved to a different parallel architecture without undue loss of efficiency. This latter goal is possible only when algorithms are expressed at an appropriate level of generality, since the actual implementation of algorithmic consructs may differ considerably for efficient utilization of different architectures.

I. INTRODUCTION

During the last two decades at least three generations of hardware components have been developed with a resulting increase in efficiency and reliability of computing machines. For high level programming languages, only the second generation has just been reached; and it is not yet widely disseminated among the scientific community which uses the special purpose computer architectures to solve their problems. The result is that most of the programmers and researchers using high performance machines are expected to tackle a task on a machine of the latest hardware technology using a comparatively inferior software tool.

This mismatch of technologies has led to cumbersome project implementations, usually delivered late and operating inefficiently. Also, the size and the complexity of the projects that a software engineer is being asked to implement have increased with the available processing power and are now almost beyond the power and features of the programming languages being used to tackle them.

There is every reason to believe that the size of the projects being undertaken in the next decade will substantially increase: the volume of data being generated by current satellites is already swamping existing computers, and this volume will increase dramatically in the next decade. Hence, from this single application area one can expect an increasing reliance on or demand for the special architectures of today's high speed computers which give greater performance over conventional computers through the exploitation of parallelism.

Most of the high level languages being used to program the existing parallel computers are extensions of languages designed many years ago for conventional sequential machine architectures: STAR FORTRAN for the STAR-100, CFT for the CRAY-1 and IVTRAN and CFD (FORTRAN-like languages) and Glypnir (an Algol-like language) for the ILLIAC IV. Each of these languages have been oriented toward one particular computer with the result that transporting codes between these machines, or to other parallel computers, is difficult if not imprudent.

It is now apparent that these parallel computers require a language created in their own generation using, as far as possible, the experience accumulated in language design and implementation techniques and incorporating the new approaches that are necessary in writing algorithms for these special architecture computers. SL/1 is one such approach being undertaken at Langley, although it is explicitly designed for the STAR-100.

A language designed for parallel computers requires a means of expressing the parallel nature of a problem for these parallel computers. This is achieved in ACTUS by means of the data declarations. In addition, new control methods are introduced for controlling the extent of parallelism in the program execution. As a final point, the language was explicitly designed to enable a compiler to generate efficient object code for the different parallel computers.

II. REVIEW OF CURRNT ILLIAC IV LANGUAGES

The currently available languages for scientific computing (primarily FORTRAN and its dialects) appear in many situations to have been overtaken by the range of applications which they are expected to handle. These applications require a greater number of instructions to be executed per second and the only way that current hardware can do this is by duplicating the existing execution units. Programming languages are, therefore, urgently required to handle parallel computation on such machines as the STAR-100 and ILLIAC IV; this, in turn, will lead the way to the development of languages for the new parallel processors of the 1980s. The existing languages for today's parallel machines have been based on FORTRAN and Algol; their major weakness is the limited capabilities of the compiler to detect the inherent parallelism which has been hidden by the syntax of a sequentially oriented programming language.

More specifically, in the case of the ILLIAC IV, the available high level languages are CFD, IVTRAN, (both FORTRAN-like) and Glypnir (Algol-like). The languages CFD and Glypnir force the user to think in terms of a fixed number of processors. The user has to be continually aware that the ILLIAC IV consists of a central processing unit with limited arithmetic capability and 64 processing units executing in lockstep. The data declarations then indicate on which processor or processors execution is to be performed. In effect, the user is aware that there is a single instruction stream with the central processor selecting those instructions that it is capable of executing. IVTRAN adopts a different philosophy by extracting the parallelism within sequential FORTRAN DO loops; the extraction is somewwat restricted and often the code generated is executed in one processing unit only. In such situations, to get the compiler to generate more efficient code, the user must restructure the program. IVTRAN also requires some knowledge of the ILLIAC IV architecture for the allocation of arrays and the alignment of operands.

Hence, the notation or framework provided by all three languages often forces the programmer to think and construct a solution to his problem in a manner which is not the most natural or straightforward. It is possible, however, to look to the past and adopt those software engineering principles which have led to the successful design and implementation of languages for the construction of reliable programs for sequential machines. A programming language for today's high performance computers should, therefore,

i) try to hide the idiosyncrasies of the hardware as much as possible from the user,

ii) enable the user to express the parallelism of the problem directly,

iii) enable the user to think in terms of a varying rather than a fixed extent of parallel processing, and

iv) enable control of the processing both explicitly and through the data, as applicable.

III. DESCRIPTION OF ACTUS

The major features of the new language are the explicit expression of parallelism in the syntax of the program and the implicit and explicit control of the extent of parallel processing during the execution of the program. Another important feature is the alignment of parallel operands. These features will be briefly described and an example program will be given.

A. Expression of Parallelism

Since today's high speed computers were developed as a means of performing the same operation on independent data, it is the data which should indicate the extent of the parallelism. To represent this parallelism in a high level language, each data declaration has associated with it the maximum extent of parallelism that can be applied to that data type.

For example, if the user wishes to operate on a three dimensional (mxnxp) array 'a' of integers, the data declaration can be used to indicate in which direction the parallel processing is to be applied. For example,

```
  var a: array[1..m; 1..n; 1:p] of integer;
```

The parallel dots ':' indicate that this is the index which is to be processed in parallel, that is, spread across the processors in an array processor or stored contiguously in a vector processor.

B. Implicit Control

The language constructs representing sequencing, selection and iteration can then be applied to such declarations. This, among other things, enables control by means of the data values. For example,

```
          var t: array[1:q] of real;
     while t[1:q]>0.0 do
     begin
           'statements'
     end
```

causes those components of 't' which are positive to determine which processors, in an array processor, are enabled for computing or which values are presented for processing in a vector processor; in both cases execution continues until all the components of 't' are negative or zero.

Another example of implicit control is the 'for' loop. The index, start and termination variables of a for loop construct can be either scalar or parallel variables; the extent of parallelism must be the same for all parallel variables in the loop. Such parallel control variables are represented by means of one-dimensional arrays. The semantic meaning of a parallel 'for' loop is that each component of the extent of parallelism has associated with it its own start, increment and termination values, but that each component can be processed in parallel. This construct is an example of a program construct that will be compiled into different machine-level primitives for different architectures. On an array processor, each processor would have its own start, increment and termination variables. When the termination condition is met in a processor, that processor will be disabled until all processors have finished executing their 'for' loop. On a vector computer, each pass through the 'for' loop will be preceded by the constrction of vectors containing the components to be processed in parallel. (For simplicity, this discussion omits the issue of data alignment.) In current high level languages for the different parallel architectures, this general construct is lacking and constructs reflecting the machine-level primitives are supplied instead (mode bits in an array processor and vector compress operations in a vector processor), thus making codes developed on one architecture prohibitively expensive to transport to another architecture. The parallel 'for' construct is used in the example at the end of this section.

C. Explicit Control

A control value can be declared to modify the extent of the parallelism. For array processors, the extent of parallelism is defined to be the number of processors that could logically compute upon a particular data structure at the same time; the definition of extent of parallelism for vector machines is similar. The members of the control value are integer values which identify the parts of an array which are to be processed. For example,

```
control inb = (2:m-1);
```

For an array processor this indicates that only processors 2 to m-1 are to be enabled. When used with the above declared array 'b' as b[inb;j] it would cause all but the perimeter of the rectangle to be processed. In a vector processor, the control value indicates which elements of the vector are to be presented for processing.

Control values can be manipulated by the following set of operators: union, intersection and difference. These operators facilitate computation on various parts of parallel data structures.

D. Dynamic Control

In those situations where the extent of parallelism does not change for a group of statements, the 'within' construct can be used to avoid repeated specification, as follows.

```
  within i:j do
begin
  t[#]:=2.0;
  b[#;1]:=b[#;3]+x[#];
end;
```

The '#' abbreviation is used to represent the extent of parallelism indicated by the 'within' specifier, in this example from 'i' to 'j' for each of the statements.

The extent of parallelism will not be re-evaluated until either the construct is exited, or until another 'within' construct or extent setting construct is encountered. The extent of parallelism can be varied dynamically by changing the values of 'i' and 'j' (in the above example) in a loop construct.

E. Alignment

It is also necessary to be able to align the elements of one parallel data structure with (any of) the elements of another parallel data structure; this is called operand alignment. In this situation contiguous elements are to be aligned with respect to other contiguous elements in a shift or wrap-around fashion.

The notation for achieving operand alignment is by using a positive integer (move data from right to left) or a negative integer (move data from left to right) as one of the operands for an alignment operator; the other operand is a control value or explicit identification of the extent of parallelism. The abbreviations shf and rtn are used to represent the shifting and rotation (wraparound) movement of data; the movement is effected on the maximum declared extent of parallelism for the variable concerned. The indices are calculated before the operands are fetched and manipulated. In particular, all index calculation and alignment actions are performed before the operations indicated by the statement are performed. For example,

```
     while t[inb]>0.0 do
begin
 t[inb]:=(t[inb shf - 1] + t[inb shf + 1])/2
end ;
```

will cause the components of 't' (excluding the extremities) to be updated using the values of its neighbors.

F. Example

The following example is the inner loop of a feature extraction algorithm that illustrates some of the features described above (from "Texture Measurement on the Illiac IV using a Maxmin Algorithm, IAC T.M. 5632). The array L contains the pixel values; 255 is used for an undefined pixel value. C is an array to count the number of threshold crossings detected on each scan line and d is the hysteresis matrix: an entry of d is 1 if the pixel value passed through the top of the hysteresis range and 0 if it passed through the bottom of the range. The variable T contains the value of the threshold parameter, the array a contains the value of the top of the threshold range and b the bottom of the threshold range.

```
within 1:j do
begin
  c[#]:=0;
  d[#,1]:=1;
  if L[#;1]=255 then L[#;1]:=0;
  a[#;1]:=L[#;1];
  b[#;1]:=L[#;1]-T;
  for i[#]=2 to S[#] do
    begin
    a[#;i[#]]:=a[#;i[#]-1];
    b[#;i[#]]:=b[#;i[#]-1];
    d[#;i[#]]:=d[#;i[#]-1];
    end;
  if (L[#;i[#]]<255) and
         (l[#;i[#]]>a[#;i[#]-1]) then
    begin
    a[#;i[#]]:=L[#;i[#]];
    b[#;i[#]]:=L[#;i[#]]-T;
    d[#;i[#]]:=1;
    c[#]:=c[#]+1-d[#;i[#]-1];
     end;
  if (L[#;i[#]]>=0) and
         (L[#;i[#]]<b[#;i[#]-1]) then
    begin
    a[#;i[#]]:=L[#;i[#]]+T;
    b[#;i[#]]:=L[#;i[#]];
    c[#]:=c[#]+d[#;i[#]-1];
    end;
  if (L[#;i[#]]<=a[#;i[#]-1]) and
         (L[#;i[#]-1>=b[#;i[#]-1) then
    begin
    a[#,i[#]]:=a[#,i[#]-1];
    b[#,i[#]]:=b[#;i[#]-1];
    c[#]:=d[#;i[#]-1];
    end;
 end;
```

THE VECTORIZER SYSTEM: CURRENT AND PROPOSED CAPABILITIES

by

Mathew Myszewski
Massachusetts Computer Associates, Inc.
Wakefield, Massachusetts 01880

ABSTRACT

The Vectorizer system, developed by Massachusetts Computer Associates, Inc., is a powerful tool for converting FORTRAN programs to use the vector operations available on large scale computer systems. The Vectorizer system addresses both the safety and the efficiency of the conversion. A machine-independent analyzer determines which FORTRAN constructs can be safely converted to vector operations. Each of the machine-dependent generators addresses the issue of producing efficient code for a particular target system. Examples will be drawn from over a year's experience with user codes.

AUTOMATIC STACKLIB FACILITIES IN STAR*FORTRAN

by

Anil K. Lakhwara
Control Data Corporation
Sunnyvale, California 94086

ABSTRACT

STAR*FORTRAN as implemented for the STAR-100 and the STAR-100A computer systems contains a variety of compile time options. One of the options which can be invoked is automatic vectorization. This option will cause the generation of vector hardware instructions in place of conventional scalar instructions. The STAR*FORTRAN automatic vectorization process has recently been expanded to include the generation of very efficient stacklib routines for a selected set of FORTRAN sub-routine constructs not otherwise vectorizable {due primarily to recursion requirements}. The examples which will be discussed include the inner product, the first sum and two second-order recursions. The method is quite general with respect to both the syntax in FORTRAN and with respect to implementation of additional coded constructs.

VECTORIZING FORTRAN

by

Lee Higbie
Cray Research, Inc.
7850 Metro Parkway, Suite 213
Minneapolis, Minnesota 55420

ABSTRACT

This paper discusses techniques that will increase the vectorizability of programs written in Fortran. The general approach is from the viewpoint of modifying existing programs so that they will run faster on the CRAY-1, but the tricks that are presented are ones that will be generally useful for any vectorizing Fortran compiler. The more fundamental techniques are ones that can be considered essential for virtually any program that is to run on a vector machine, others are stop-gap measures that will not hurt in the long run but will only be necessary in the near future for CFT, Cray Fortran. Topics that are not discussed are vector extensions to Fortran and vector functions. The topic is rewriting standard Fortran to increase vectorizability, within or close to the confines of standard Fortran.

COMPUTATIONAL FLUID DYNAMICS, ILLIAC IV, AND BEYOND

by

K. G. Stevens, Jr.

NASA-Ames Research Center
Moffett Field, CA 94035

ABSTRACT

This paper summarizes the computational fluid dynamics work being done at Ames Research Center utilizing the ILLIAC IV. Both the physical nature of the problems and computational requirements are discussed with utilization of the Illiac Disk System being highlighted. Finally, results of studies for an even more powerful computational resource to be incorporated in a proposed Numerical Aerodynamics Simulation Facility are also discussed with respect to the Ames computational fluid dynamics effort.

I. INTRODUCTION

At present the Computational Fluid Dynamics Branch at Ames Research Center is developing several codes for use on the ILLIAC IV. These codes are the fore-runners for the eventual development of three dimensional Reynolds-averaged Navier-Stokes codes which can be used for aircraft design. This paper briefly discusses three of these codes and their use of the ILLIAC IV. It then discusses the Numerical Aerodynamics Simulation Facility (NASF) which is an effort to develop a computational facility of sufficient magnitude to allow economic solution of these codes once they are developed.

II. TRANSONIC AILERON BUZZ

Transonic aileron buzz was first encountered by World War II pilots and is described as a one degree of freedom flutter where shock wave motion causes a phase difference in the response of the hinge moment to aileron movement. Tests were conducted from 1947 to 1949 on an F-80 wing mounted in the Ames 16-foot wind tunnel. The wing tip was supported to prevent bending, and the aileron was allowed to move freely about the hinge line.

A recently developed viscous flow airfoil code[1] has been used to simulate transonic aileron buzz. The thin-layer Navier-Stokes equations are solved with the turbulence modeled by a two-layer algebraic eddy viscosity model. The calculated results, which were obtained on the ILLIAC IV, are in essential agreement with the wind tunnel data (see Figure 1.) For an angle of attack of -1 degree and Mach numbers of 0.76 and 0.79, the aileron failed to buzz. at a Mach number of 0.82, buzz occurred and the calculated aileron deflection history nearly follows the experimental data.

The ILLIAC IV code uses up to a 64 x 128 grid. This code runs about three times faster on the ILLIAC IV than on the CDC 7600.

III. SIMULATION OF TURBULENCE

Turbulence and transition phenomena are being simulated on the ILLIAC IV, which has the capacity for handling 500,000 grid points[2]. This permits spacial resolution much finer than possible before with serial processors. The compressible Navier-Stokes equations have been solved for several three dimensional geometries including a circular jet[3]. The development of the jet is traced from the orifice, through roll-up into "smoke rings" and into transisition to turbulent flow (See Figure 2.) The mean velocity profile and turbulent intensities in the resulting turbulent jet are similar to those observed in subsonic jets. More detailed comparisons with experimentally measured shear stresses and temporal correlations are planned as are computations of the noise produced by the turbulence as determined by the contained quadrupole sources and also by the pressure and normal velocity data on a surrounding control surface.

These classes of codes use FFTs and finite difference methods on grids as large as 128 x 64 x 64 and make heavy use of the Illiac Disk system. This problem would be impractical on a CDC 7600 due to the slow disk transfer rates. The FFTs used by these codes run six times faster that those on the CDC 7600.

IV. VISCOUS SEPARATED FLOW IN THREE DIMENSIONS

A three dimensional Reynolds-averaged Navier-Stokes code has been developed for the ILLIAC IV[4]. The current code uses a 40 x 40 x 40 grid. A laminar viscous flow calculation is shown in Figure 3 for a hemisphere-cylinder with an angle of attack of 19 degrees and free stream Mach number of 1.2. The calculation compares well with the experimental profiles from Hsieh[5]. On the leeward side streamwise separation occurs at the nose. Points of streamwise seperation and reattachment predicted by numerical calculations are denoted by S and R in Figure 3. Also shown in Figure 3 are numerical results for the windward side which remains attached and crossflow velocity vectors which show the separation. This code is now being modified by Lomax, Pulliam and Steger to simulate the flow around a wing using a 80 x 40 x 48 grid. This code is a path finder on the way to engineering use of the three dimensional Navier-Stokes solutions. It is this code which uses an implicit method and another three dimensional code which uses an explicit method to solve the Reynolds-averaged Navier-Stokes equations that are serving as the aerodynamics benchmarks for the NASF study. This code runs from 2 to 5 times faster on the ILLIAC IV than the CDC 7600 depending on the grid size.

V. THREE DIMENSIONAL CODES AND THE ILLIAC IV.

The ILLIAC IV currently has a small "vector" access memory and a large block addressable memory which is logically a drum. Because of this memory organization considerable effort has gone into developing buffering schemes for the ILLIAC IV. One of the schemes is discussed by Lomax[6]. This scheme works well when the numerical method can be factored into a sequence of one dimensional operators. This factoring allows the data to be brought into memory in a series of x, y, and z columns so that one dimensional operators can be performed in parallel across the cross section of the column as shown in Figure 4. Also shown in this figure is the manner in which the subblocks are skewed on the drum to allow equally fast access to all three directions. (Note: The data within the subblocks are stored in the usual FORTRAN way. This implies actual transposes of the data must be performed if contiguous vectors are to be accessed or it implies noncontiguous memory access for vectors for two of the column accesses.) This scheme for data buffering will be applicable to the new Charged Coupled Device (CCD) memories and allows easy identification of "vectors" provided the numerical method can be split into a sequence of one dimensional operators.

VI. NUMERICAL AERODYNAMICS SIMULATION FACILITY (NASF).

Even with improvments in numerical methods it is anticipated that a machine 40 times more powerful than the ILLIAC IV will be required to allow the use of three dimensional Reynolds-averaged Navier-Stokes solutions as an engineering tool. This is a goal of NASF and corresponds to solution times of less that ten minutes per case which translates to the rate of one billion useful floating point calculations per second. An Ames survey of the computer industry as well as one by Los Alamos Scientific Laboratory and other surveys predict that a machine of this magnitude will not exist in the mid-1980's unless its development is sponsored by a user orginization. Ames further determined that there were no technology constLaints which precluded such a machine in that time frame.

With this in mind Ames has awarded two feasibility studies to determine if the fabrication of a NASF is possible in the mid-1980's. These studies have been awarded to Burroughs and Control Data Corporations. The remainder of this paper covers the concepts which these corporations feel indicate that NASF is feasible.

VII. BURROUGHS CORPORATION CONCEPT OF NASF

The Burroughs concept[7] is shown in Figure 5. The Support Processor System (SPS) for the Burroughs NASF concept is conventional and is built around hardware and software similar to a B7700. This SPS concept is quite similar to the way the BSP is front-ended. However, the Flow Model Processor (FMP) shown in the system is quite different from the BSP and ILLIAC IV. The major points to notice about the Burroughs FMP concept are

1. 512 Processors with their own instruction issue unit
2. 521 Memory modules
3. Transposition Network (TN)
4. CCD staging memory.

The fact that each processor has its own instruction stream makes this concept quite different from other lock-step parallel designs. Most of the problems with data dependent branches are gone as are the problems with "scalar" code. This would be a nearly optimal approach if we could afford a full cross-bar switch as the TN. Memory conflicts are greatly reduced by having a prime number of memory banks. However, a full access TN is too large and expensive to be practical. It's here where Burroughs has spent much of its time. The current Burroughs concept for the FMP incorporates a double Omega network with one layer having inverted priorities as the TN. This network seems to be a good compromise between throughput, cost, and reliability. Although normal access to the memory is for planes of data from three dimensional arrays, the machine's multiple instruction stream concept would, over time, randomize the "vectors" to be fetched. If the TN were not a cause of delay the Burroughs FMP would run at 1.7 Billion Floating Point Operations per Second (GFLOPS) on the benchmark aerodynamics codes. When the TN is included the estimated speed on a real aerodynamics code is estimated to be one GFLOPS.

VIII. CONTROL DATA CORPORATION CONCEPT OF NASF

The Control Data Corporation (CDC) concept[8] is shown in Figure 6. The front end hardware is all connected to high speed network trunks. This allows many different types of devices to be accessed by the FMP. The CDC proposed SPS includes two CYBER computers, an 819 disk farm, 38500 archival storage systems and high speed graphics.

The CDC FMP concept is similar to the CDC STAR 100. The major difference being the size of memory--eight million words, and the number of pipes--nine, as can be seen in Figure 6. Special care is being taken to minimize unoverlapped startup times for the pipelines. The FMP pipes allow linking of operations similar to the chaining allowed by the CRAY-1. If a 16 nanosecond clock is used, the CDC FMP has a maximum speed of three GFLOPS. This speed is reduced for real problems since vectors must be gathered and control vectors must be used instead of IF statements. It is presently estimated that this FMP concept will execute the benchmark aerodynamics codes at a rate of about one GFLOPS.

IX. CONCLUSION

It appears at this point that it is possible to build the NASF in the mid-1980's and that with continued code development it will be possible to use three dimensional Reynolds-averaged Navier-Stokes solutions as engineering design tools at that time. Ames is, therefore, seeking funding beginning in FY80 for design and construction of the NASF.

REFERENCES

1. J. L. Steger and H. E. Bailey, "Simple Calculations of Transonic Aileron Buzz," submitted to the AIAA 17th Aerospace Science Meeting, 1978.
2. R. S. Rogallo, "An ILLIAC Program for the Numerical Simulation of Homogeneous Incompressible Turbulence," NASA TM-73,203, November 1977.
3. A. Wray, Ames Research Center, Moffett Field, California, private communication, 1978.
4. T. H. Pulliam and J. L. Steger, "On Implicit Finite-Difference Simulations of Three Dimensional Flow," AIAA Paper 78-10, 1978.
5. T. Hsieh, "An Investigation of Separated Flow About a Hemisphere-Cylinder at 0- to 90-deg Incidence in the Mach Number Range from 0.6 to 1.5," AEDC-TR-76-112, July 1976.
6. H. Lomax, "Three-dimensional Computational Aerodynamics in the 1980's," NASA CP-2032, February 1978.
7. Burroughs Corporation, "Final Report Numerical Aerodynamics

Simulation Facility Preliminary Study Extension," NASA Contract No. NAS2-9456, February 1978.
8. N. R. Lincoln, et. al., "Preliminary Study for a Numerical Aerodynamics Simulation Facility Final Report -- Phase 1 Extension," NASA Contract No. NAS2-9457, February 1978.

CALCULATION OF AILERON BUZZ

STEGER/H. BAILEY, 1978

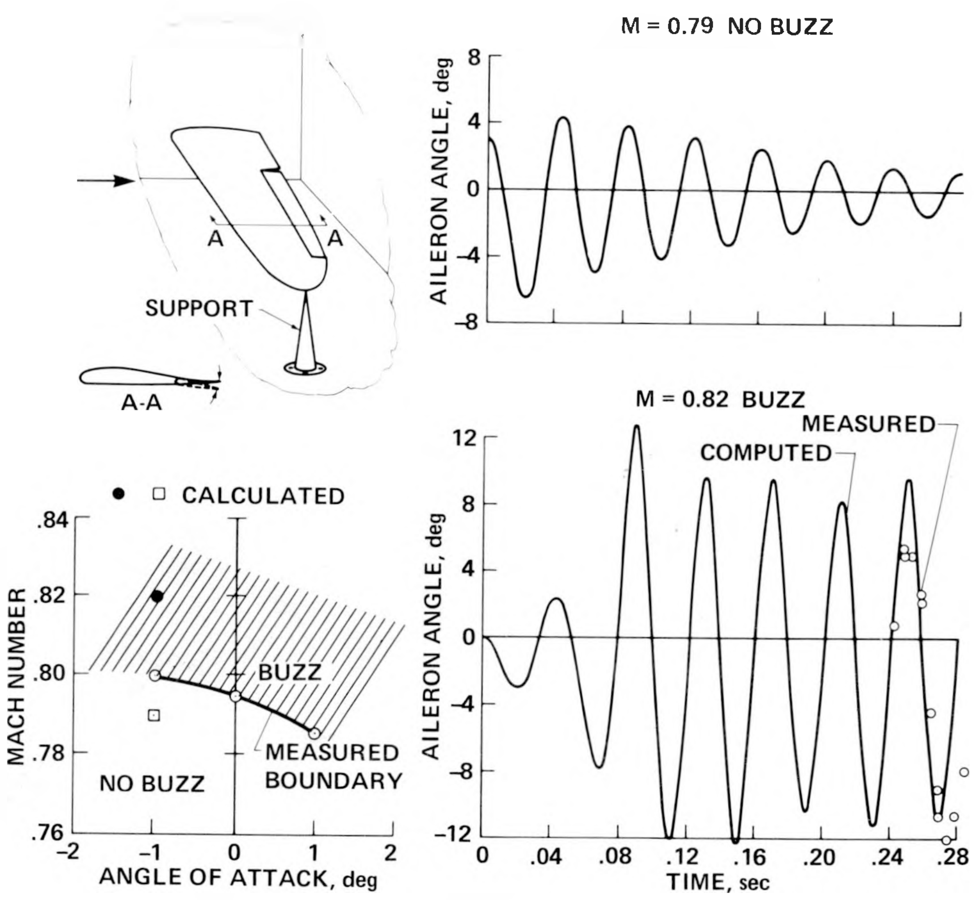
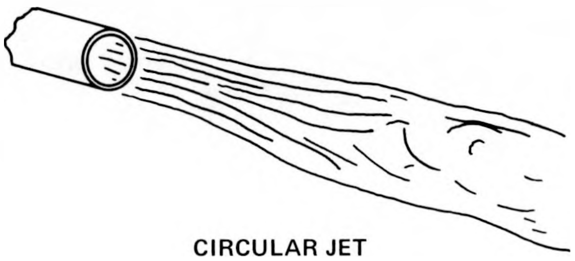


FIGURE 1.

SIMULATION OF TURBULENCE



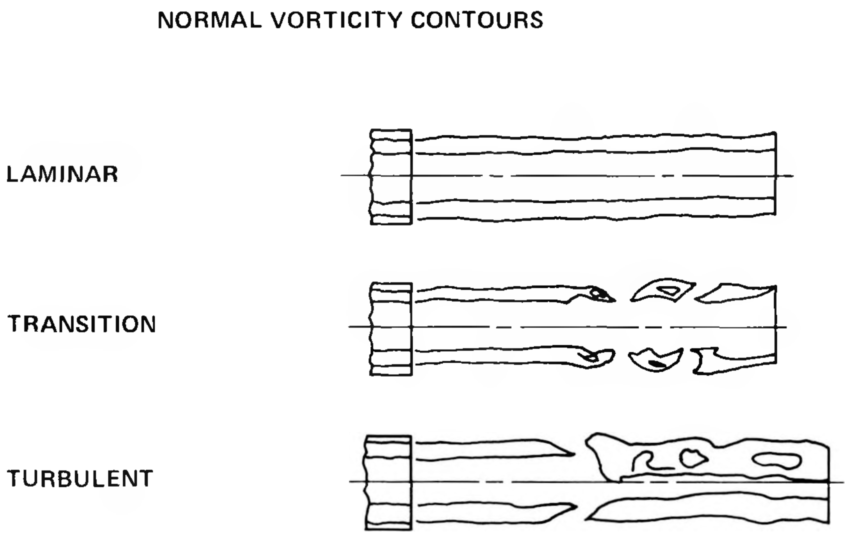
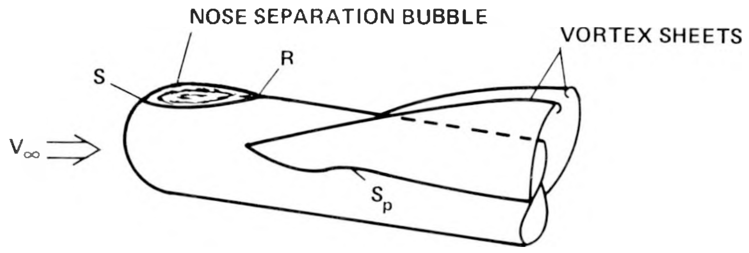
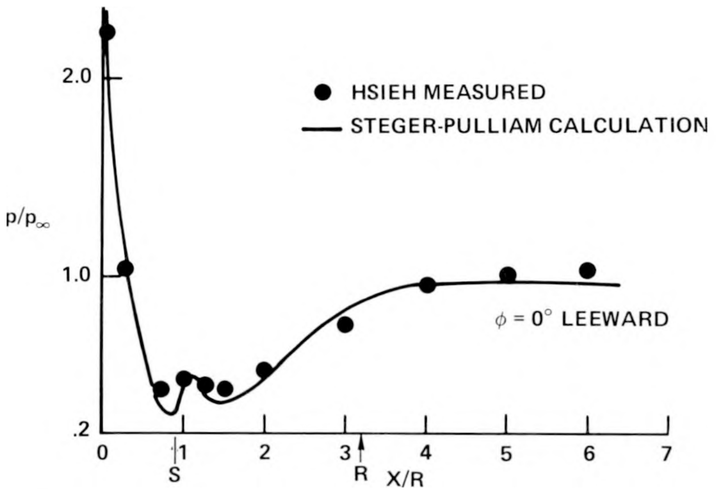


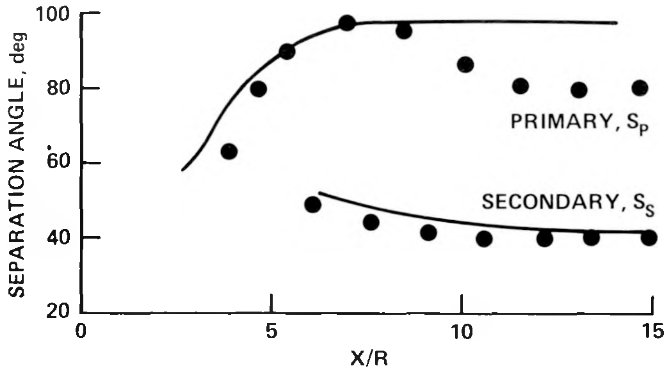
FIGURE 2.

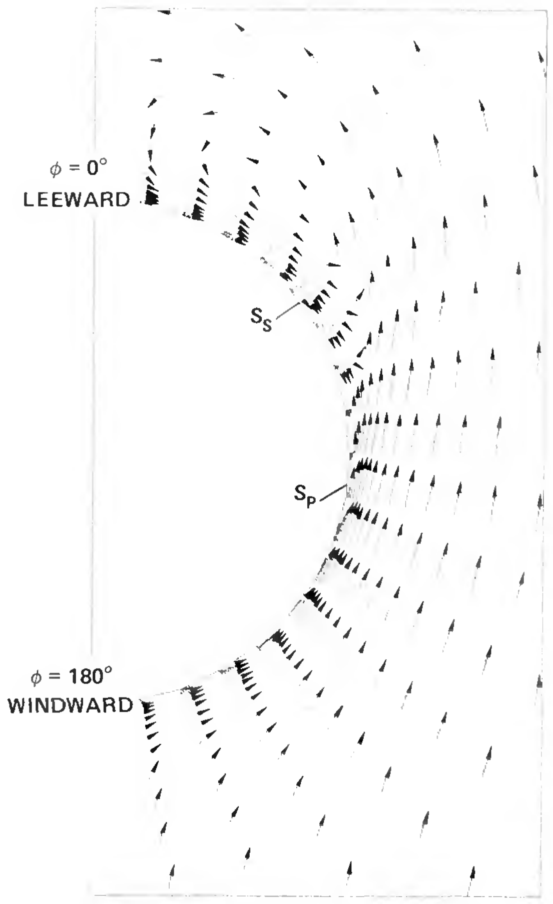
VISCOUS FLOW ABOUT A HEMISPHERE - CYLINDER

$M_\infty = 1.2 \quad \alpha = 19° \quad Re_D = 445{,}000$





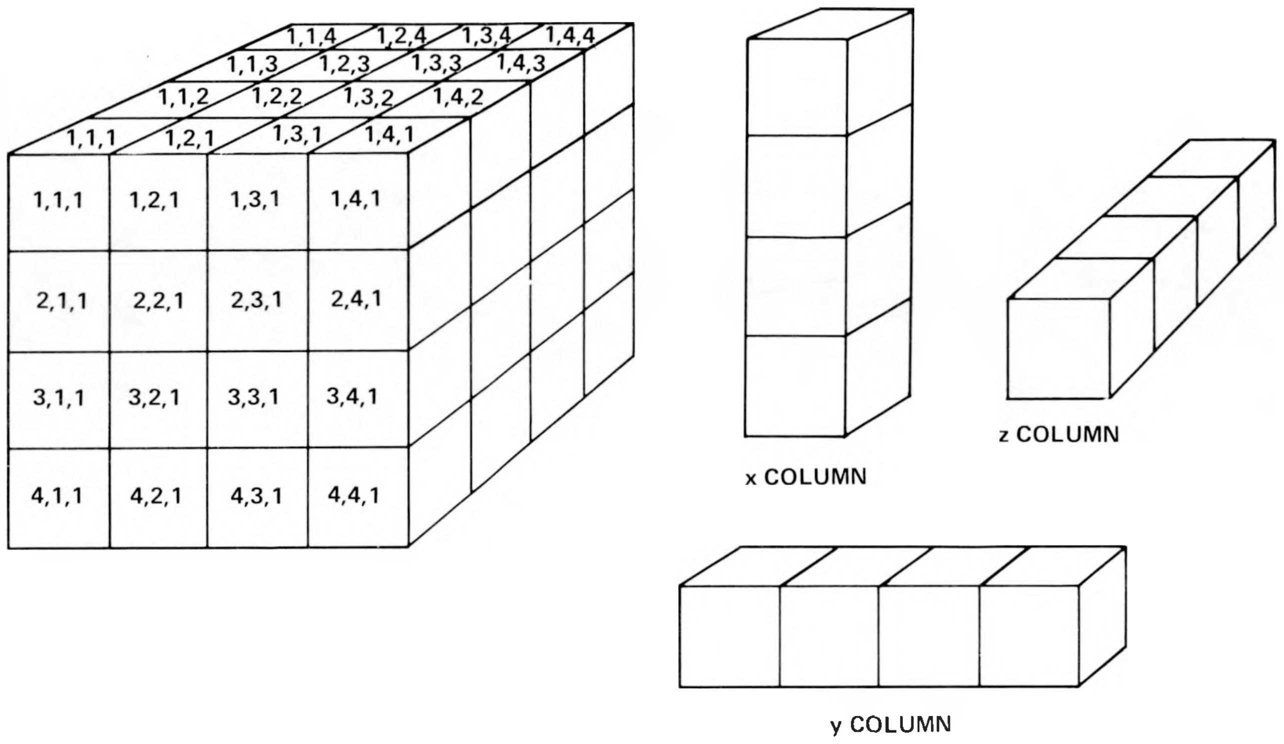




CROSSFLOW VELOCITY VECTORS

FIGURE 3.

DATA BUFFERING ON ILLIAC IV



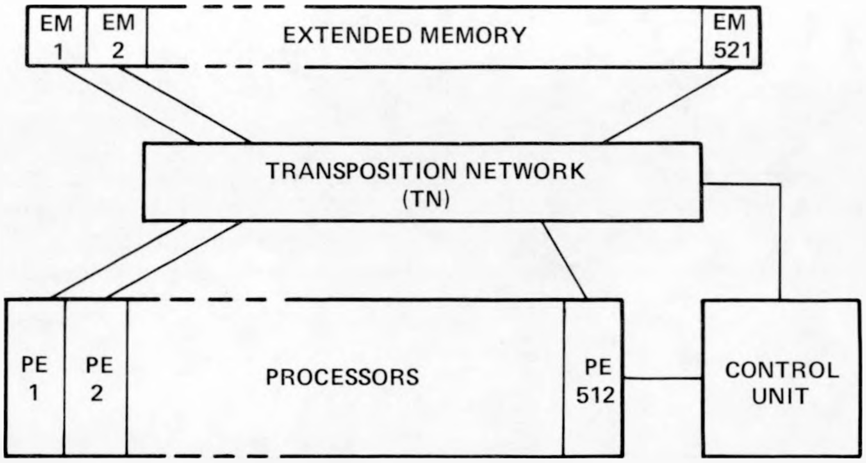
SKEWED DATA STORAGE OF SUBBLOCKS

1,1,1															
2,1,1	1,2,1			1,1,2											
3,1,1	2,2,1	1,3,1		2,1,2	1,2,2			1,1,3							
4,1,1	3,2,1	2,3,1	1,4,1	3,1,2	2,2,2	1,3,2		2,1,3	1,2,3			1,1,4			
	4,2,1	3,3,1	2,4,1	4,1,2	3,2,2	2,3,2	1,4,2	3,1,3	2,2,3	1,3,3		2,1,4	1,2,4		
		4,3,1	3,4,1		4,2,2	3,3,2	2,4,2	4,1,3	3,2,3	2,3,3	1,4,3	3,1,4	2,2,4	1,3,4	
			4,4,1			4,3,2	3,4,2		4,2,3	3,3,3	2,4,3	4,1,4	3,2,4	2,3,4	1,4,4
							4,4,2			4,3,3	3,4,3		4,2,4	3,3,4	2,4,4
											4,4,3			4,3,4	3,4,4
															4,4,4

FIGURE 4.

BURROUGHS NASF CONCEPT

BURROUGHS FMP CONCEPT



BURROUGHS SPS CONCEPT

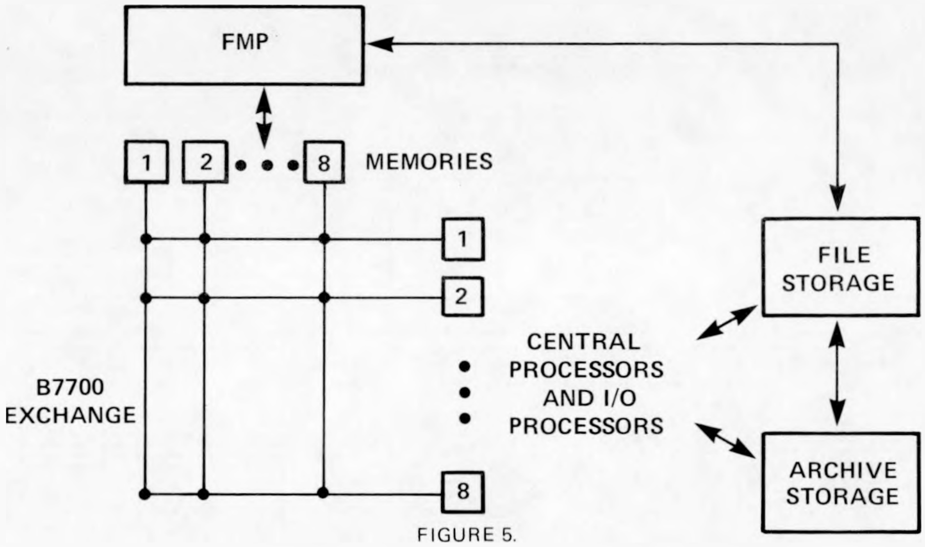
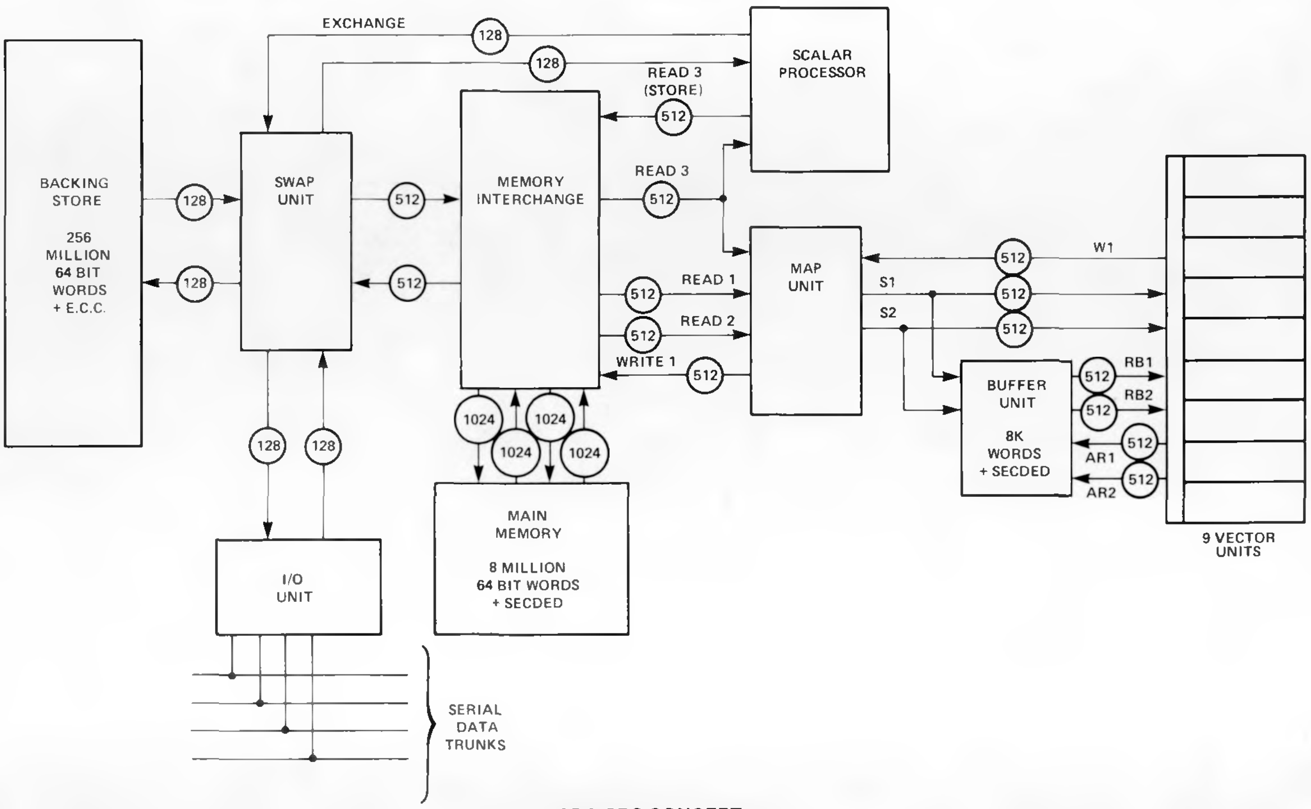


FIGURE 5.

CDC NASF CONCEPT

CDC FMP CONCEPT



CDC SPS CONCEPT

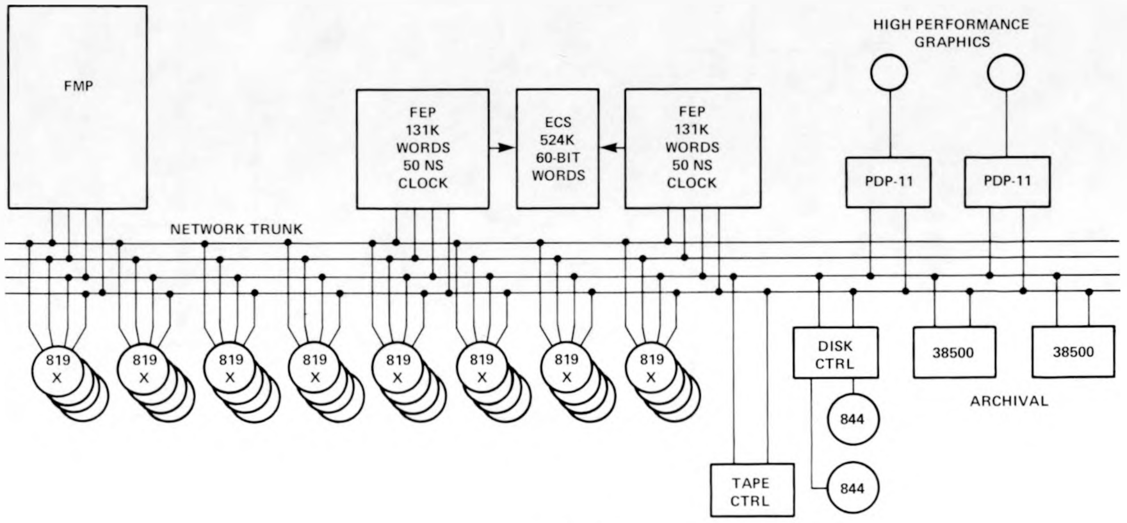


FIGURE 6.

DAP - A FLEXIBLE NUMBER CRUNCHER

by

S. F. Reddaway
International Computers Ltd.
London, England

ABSTRACT

A wide range of routines and large applications can be performed on the Distributed Array Processor[1] (DAP). The DAP goes back to fundamentals, offering new and very different basic capabilities. These capabilities have been applied to a number of applications and techniques, many of which have been implemented on the pilot DAP which has 1024 processing elements arranged 32 x 32 and has been running for two years

A three dimensional stellar evolution simulation in which each of 1024 stars interacts every time step with every other star under gravitational force has been implemented in DAP-Fortran. Such total interaction and the lack of a grid structure mean that it is not an obvious array processor problem. However, $1\frac{1}{2}$ million force components are calculated each time step in 3.3 seconds, giving a performance on this problem of about 2 times an IBM 360/195.

A Hadamard transform, implemented in DAP-Fortran with 2048 16-bit integers, takes 1.3 msec; due to the use of low precision fixed point and the absence of multiplications, this is about 10 times faster than the 1024 point complex FFT dealt with in reference 1.

Tridiagonal systems of N equations have been solved using a "recursive doubling" technique needing log N steps. The technique has been used to solve Poisson's equation on a 32 x 32 grid with an ADI method; a straightforward implementation in DAP-Fortran took 150 msecs for 12 iterations, at which point it had nearly reached the final precision. An improved method implemented in DAP-Fortran does most of the computing on increments using shorter words and took 85 msecs for 12 iterations; at this point the error was rather less than the straightforward method and the final precision was substantially better.

Reference 1 reported 29 msec for an assembly language 32 x 32 matrix inversion with full pivoting. This has since been reduced to 26 msec. A DAP-Fortran implementation, using essentially the published code, executes in 30 msec, representing a 15% overhead compared with assembly language. Many

DAP-Fortran routines, for example matrix multiply, have a substantially lower overhead.

Several arithmetic functions are remarkably fast. Reference 1 has already reported square root as significantly faster than multiplication. Logarithm and exponential, now being implemented, are expected to be almost as fast as multiplication.

On the DAP, many problems can use parallel table look-up on a common table. Recent ideas can improve earlier DAP performance estimates by a factor of up to 10.

The Processing Element of the DAP is simple, general and flexible, with few decisions built in to the hardware. Operations are built up in low level software so that simple things such as sign changing or comparing numbers are much faster than complex ones such as floating point arithmetic. Hence the organisational overheads normally associated with numerical work tend to be small. Boundaries and conditional operations can be handled effectively by activity control, often with less degradation of performance than on a conventional machine.

Simple categorisations of parts of jobs into "scalar" and "vector" have proved unsound if applied to the DAP. Effective DAP solutions have been found to most large compute-bound jobs studied, with nearly all processing being done in the DAP. Indeed, factors which make jobs large and CPU-bound, tend to make them well suited to the DAP. The range of application of the DAP already extends from scene analysis to number crunching, and the limits to its potential have yet to be found.

DAPs with 64 x 64 processing elements, 4 times more powerful than the pilot with 2M Bytes of integral store, are being manufactured. The first delivery will be to Queen Mary College, London University.

REFERENCE

1. P.M. Flanders, D.J. Hunt, S.F. Reddaway and D. Parkinson, "Efficient High Speed Computing with the Distributed Array Processor", High Speed Computer and Algorithm Organisation (Academic Press, Inc., New York, 1977), pp 113-128.

Dup

ARRAY PROCESSING ON THE PDP-10

by

Neil Maron
and
George G. Sutherland
University of California
Lawrence Livermore Laboratory
Livermore, California 94550

ABSTRACT

We are attaching a Floating Point Systems, Inc.(+) microprogrammable array processor (AP190-L) to the M-division PDP-10. The AP has a cycle time of 167 nanoseconds (ns) and is interfaced though another microprocessor to the PDP-10 memory bus and I/O bus. The AP is capable of producing the results of a multiply and an addition every cycle. This is permitted because of parallel functional units. The total time for a multiply is 3 cycles and for an addition is 2 cycles.

We will report on the PDP-10 monitor overhead required to obtain useful results from the array processor as well as some benchmark results of PDP-10 performance enhancements.

(+)Reference to a company or product does not imply the exclusion of any other that may be suitable.

DISTRIBUTED NUMERIC COMPUTING ENGINES: MINICOMPUTER-BASED VECTOR AND PARALLEL COMPUTING

by

W. Morven Gentleman
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

In many scientific and engineering institutions, the requirement for massive computing power comes not from a few problems with immense requirements and tight realtime constraints, but rather from a large number of problems, each with substantial requirements but requiring only reasonable turnaround. In this situation, the enhanced reliability, the opportunities for modular growth, and the cost (in today's world where economies of scale are associated with volume production rather than unit capacity) make it attractive to consider a distributed solution to meeting the computing needs instead of the conventional large machine solution. From currently commercially available equipment, it is possible to put together configurations such that each user, or at least each small group of users, has an individual computer of considerable power, with network access to shared services such as a central file store or microfilm graphics devices. The individual numeric computing engines can reasonably be given ample local backing store, through virtual memory techniques can appear to be given large main memories, and through add-on vector processors or collections of subservient microcomputers can be given adequate compute power. There are
ter of considerable power, with network access to
shared services such as a central file store or microfilm graphics devices. The individual numeric computing engines can reasonably be given ample local backing store, through virtual memory techniques can appear to be given large main memories, and through add-on vector processors or collections of subservient microcomputers can be given adequate compute power. There are
software problems, but not worse than for any new architecture.

Some specific configurations will be presented. Their pros and cons will be discussed and performance estimated. The use of these configurations to solve several sample problems will be discussed, in order to address that vital question: could such computing be exploited by average users without requiring an overwhelming effort to learn a new way of programming?

DESIGN OF ARITHMETIC ELEMENTS FOR BURROUGHS SCIENTIFIC PROCESSOR

by

Daniel D. Gajski
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

and

L. P. Rubinfield
Burroughs Corporation
Great Valley Laboratories
P. O. Box 517
Paoli, Pennsylvania 19301

ABSTRACT

The design criteria and implementation of the Arithmetic Element (AE) of the Burroughs Scientific Processor, a vector machine intended for scientific computation requiring speed of up to 50 million floating-point operations per second, is discussed. An array of 16 AEs operate in lockstep mode, executing the same instruction on 16 sets of data. The 16 AEs are one stage in a pipeline which consists of 17 memory modules, an input alignment network, and an output alignmment network. The AE itself is not pipelined. It can perform over one hundred different operations including a floating-point addition, subtraction and multiplication (320 ns), division (1280 ns), square root (1920 ns), among the others. Eight registers are provided for the storage of intermediate values and results. Modulo 3 residue arithmetic is used for checking hardware failures.

Optimization of Vector Operations in an Extended Fortran Compiler

Ken Kennedy

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Visiting Staff Member
Los Alamos Scientific Laboratory
Los Alamos, New Mexico 87545

Abstract

The advent of new versions of Fortran which recognize vectors as computational entities gives rise to some interesting problems for the compiler designer. If the intermediate representation is carefully chosen, the full power of scalar optimization techniques can be brought to bear on vectors as well, resulting in significant code improvement. Redundant subexpression elimination, constant folding, code motion, and vector register allocation are examples of applicable techniques. New optimizations, such as those concerned with the management of vector temporary storage, can also be found. This paper discusses possible designs for an optimizing vector Fortran compiler and gives examples of some of the transformations it could perform. The main conclusion is that vector Fortran can be translated into excellent code, even if the target machine has no vector instructions.

I. Introduction

There has been substantial recent interest in languages which provide vector extensions to Fortran; examples are Vectran [PaW75,PaW78] and BSP Fortran [Bur77]. This paper discusses some preliminary ideas on the design of an optimizing compiler for such a language. Since the additional complexity of scanning and parsing vector extensions to standard Fortran is fairly small, concentration is on the semantic phases of the compiler – optimization, storage allocation, and code generation.

Although much of the interest in vector languages is due to the advent of vector machines like the Cray-1, vector notation is a convenient language feature even on a scalar machine. In scientific applications, vectors occur naturally, and a good scientific language should permit their use as atomic entities. Careful examination of many scientific programs will show that the innermost loops are often just scalar encodings of vector operations. In other words, vector extensions make sense because they are useful for specifying algorithms, whether or not the algorithms will be run on a vector machine. I shall contrast the problems of compiling for scalar versus vector machines; as it happens, the optimization phases should be quite different.

A major consideration in designing a vector language compiler is the extent to which code from an existing Fortran compiler can be used. Clearly, the scanner and parser will not need much modification. If scalar optimizations can be easily upgraded to handle vectors as well, the effort required to produce efficient code for vector extensions will be significantly reduced. For this reason the discussion considers how best to include vectors in a classical compiler design.

The paper is organized into six sections, including this introduction. Section II describes classical compiler organization and presents a plausible internal format for vector operations. Storage mappings are the subject of Section III, which describes one representation of array access functions. Section IV introduces a variety of array optimizations, discusing their applicability. Section V concludes the paper by sketching the overall organization of vector compilers for both vector and scalar machines.

I make no attempt to include a systematic discussion of *automatic vectorization,* although this should be an important part of any Fortran compiler for a vector machine. However, the usefulness of many of the optimization techniques described here is predicated on the existence of a preliminary vectorization pass. In other words, many of the inefficiencies that vector optimization attempts to eliminate are ones that might be introduced by vectorization. This is another instance of the rule that optimization usually creates further opportunities for optimization.

II. Conceptual Preliminaries

If we are to implement vectors in Fortran with a minimal effort, we must first understand the structure of traditional Fortran compilers. A fairly typical organization, used in the Fortran H compiler [LoM69], is depicted in Figure 1.

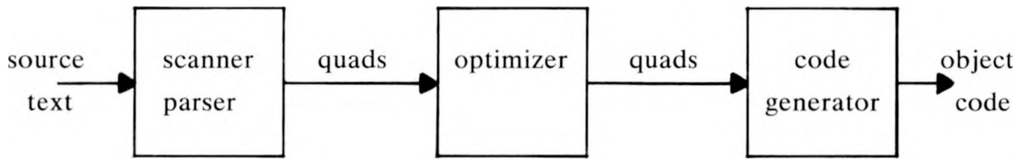


Figure 1. Classical compiler organization.

First a *lexical scan and parse* phase converts the source program to some intermediate code, quadruples for example. Quadruples usually have fields specifying the operation, up to two source operands, a result operand, and auxiliary information; the latter field is used by the optimizer and code generator to specify which operands are found in registers, which registers they occupy, whether the instruction should be deleted, and so on. The *optimization* phase accepts quadruples as input and produces quadruples as output, but with the auxiliary information fields filled in. The *code generator* translates this final intermediate code into a machine language program.

The functioning of the semantic phases in such a compiler can be demonstrated by an example. Consider the statement

$$I = I + J*K$$

The parser might convert this to the following quadruples

$$\begin{aligned} T_1 &\leftarrow J*K \\ I &\leftarrow I+T_1 \end{aligned}$$

where T_1 is a compiler-generated temporary. Suppose that the optimizer then discovers that J*K is a *redundant subexpression,* one whose value has been previously computed and stored in temporary T_{J*K}. Then the multiplication can be eliminated.

$$\begin{aligned} &T_1 \leftarrow T_{J*K} \\ &I \leftarrow I+T_1 \end{aligned}$$

On further study, the optimizer might discover that T_1 is never used again, so all of its uses can be replaced by T_{J*K} and its definition can be eliminated.

$$I \leftarrow I+T_{J*K}$$

Next, a *register allocator* attempts to assign heavily-used quantities to CPU registers. If it discovers that *I* is a loop induction variable, it will probably assign *I* to a register, say R3. This would be recorded as auxiliary information.

$$I(R3) \leftarrow I(R3) + T_{J*K}$$

With optimization completed, the code generation phase assigns addresses to both *I* and T_{J*K} and, based on the auxiliary information, selects a code fragment to represent the quadruple.

$$\text{ADD} \quad R3,A(T_{J*K})$$

The code fragment, along with all other fragments and some initializing code, would be written out in linkage editor format.

How is this process complicated by vectors? Keep in mind that we want to introduce vectors at just the right time to take advantage of the scalar code already in place. One problem with the Fortran-to-Fortran vectorizer at Los Alamos is that the output vector operations bypass Fortran optimization, so many opportunities for code improvement are missed [Bas78]. If vector languages are to be suitably efficient, the full power of Fortran optimization techniques must be brought to bear on them. It seems clear then that vectors should be introduced in a form that can be handled by the scalar optimization phase with only slight modifications. This implies that vectors should be atoms in the language of quadruples itself; that is, we should be able to write

$$A \leftarrow B*C$$

where A, B, and C represent vectors.

There are two aspects of the use of vectors which should be treated separately in intermediate code. When used in a computation, a vector is simply a sequence of values. However, this value sequence may be arranged in storage in a variety of ways. Thus, I shall draw a careful distinction between these two actions – *operation* on a vector and *extraction* of vector values from a pattern of cells in storage. To keep this distinction in mind, I shall consistently use the term *vector* to mean the value sequence and the term *array* to refer to the storage pattern.

Formally, a *vector* is a sequence of values of a given type; the *length* of a vector is the number of elements in the sequence. An *array* is a multiple value stored in program memory; an array has four main attributes (a fifth will be discussed in the next section):

dimensionality – the number of subscripts it uses in access operations,
shape – a vector of dimension sizes, one for each dimension,
origin – the vector of starting indices, and
address – the location of the origin element in storage.

For each array A, all this information is encoded in a function f_A called the *storage mapping function* for A. In a simple array access to A(I,J), f_A(I,J) is evaluated for the address of the desired cell. Within this model, range checking can be included, if we wish, by specifying that the result of evaluating f_A on illegal indices is the *undefined atom* Ω.

Using this model we can identify three operation classes:

1) array definition
2) array-vector conversion
3) vector computation

An *array definition* is the specification of a new storage mapping function g from an old one. For example, in Vectran the IDENTIFY statement gives rise to a new array definition. Consider the statement

```
IDENTIFY /L1,L2/ A(I,J) = B(J,I*2)
```

which defines A as a subarray whose elements occupy the same storage as the elements of B. No copying is to be done. The proper response is to define a new storage mapping function f_A such that

$$f_A(I,J) = f_B(J,I*2) \quad 1 \leq I \leq L1,\ 1 \leq J \leq L2$$

$$f_A(I,J) = \Omega \qquad \text{otherwise}$$

An *array-vector conversion* produces a vector of values from a storage mapping function; the vector consists of all values in the array, in canonical order, beginning with the the one at its origin. Thus if f_A is defined as above,

$$V \leftarrow \text{getvector}(f_A)$$

produces the vector

A(1,1),A(2,1),...,A(L1,1),A(1,2),A(2,2),...,A(L1,L2)

in the Fortran canonical order. Often, in response to a language construct like the subscript iterator in BSP Fortran,

X(1:99:2),

the transformation to a vector consists of two steps – first the subscript mapping function for the subarray is constructed, then the vector is extracted according to the new function. Array-vector transformations are not limited to extraction; conversion may go in the opposite direction, depositing a vector of values into some pattern of storage cells via the instruction

$$\text{putvector}(f_A,V)$$

A *vector computation* applies some operator to one or more vectors to produce a vector (or possibly scalar) result. For example,

$$V_3 \leftarrow V_1 * V_2$$

To be legal, the operands in such a computation must either have the same length or one must be a scalar. Two operands which meet this criterion are said to be *conformable*. Conformability errors can arise in conversion from vector to array form as well; an attempt to put a vector into an array that is either too large or too small is illegal.

To see how these three operations interact, consider the following Vectran code

```
      RANGE /L1/ A(100),B(100) /L2,L1/ C(100,100)
         .
         .
         .
   10 A(*) = B(*) * C(I,*)
```

The code generated for statement 10 must consist of these parts:

1) create a storage mapping function f_{CS} for the subarray C(I,*),
2) check the conformability of f_B and f_{CS},
3) extract vectors based on these two storage mappings,
4) compute the product vector,
5) check for conformability with f_A, and
6) save the vector according to f_A.

A high-level intermediate code version might look like this:

```
def[$f_{CS}(k) = f_C(I,k)$]
conform($f_{CS}, f_B$)
$V_B \leftarrow$ getvector($f_B$)
$V_C \leftarrow$ getvector($f_{CS}$)
$V_A \leftarrow V_B * V_C$
conform($V_A, f_A$)
putvector($f_A, V_A$)
```

Note that the *def* operation is clearly a macro; its expansion will be discussed in the next section. Conformity checks will usually be optimized away at compile-time; this is always possible in Vectran, for example.

The correspondence between these operations and low-level machine code should be obvious. Vector computations are register-to-register vector instructions, array-vector conversions are vector register loads and stores, and array definition is dope vector computation, although the latter will usually be done at compile time without using explicit dope vectors.

III. Representation of Storage Mappings

Conceptually, each storage mapping function will be embodied in a dope vector. Although this dope vector may consist entirely of compile-time constants, it is useful to envision it when trying to understand the algebra of access functions. The format described here is adapted from one originally proposed by Kantorovich in 1956 [ErS76}. Each dope vector will contain the following infomation about the array it describes:

1) the address P
2) dimensionality D
3) shape $(n_1,\dots,n_D)$
4) origin $(i_1,\dots,i_D)$
5) stride $(d_1,\dots,d_D)$

where the stride vector contains the distances between adjacent elements along the same dimension. Thus for the array A(5,10) of four-byte words, the dope vector would contain

1) address *addr*(A(1,1))
2) dimensionality 2
3) shape (5,10)
4) origin (1,1)
5) stride (4,20)

where the stride is given in bytes. In this scheme the address of element $(j_1,\dots,j_D)$ is given by the formula

$$f(j_1,\dots,j_D) = P + \sum_{k=1}^{D}(j_k - i_k)*d_k$$

so in our example, A(2,3) would be located at

$$\mathit{addr}(\mathrm{A}(1,1)) + (2-1)*4 + (3-1)*20$$

$$= \mathit{addr}(\mathrm{A}(1,1)) + 44$$

It may be useful, for the purpose of optimization, to keep several extra fields in the dope vector. For example, a conventional trick is to use the location of the hypothetical zeroth element of an array in address calculations. This location is computed as follows:

$$A_0 = f_A(0,\dots,0) = P + \sum_{k=1}^{D}(0-i_k)*d_k$$

$$= P - \sum_{k=1}^{D} i_k * d_k$$

Then the address computation becomes

$$f(j_1,\dots,j_D) = A_0 + \sum_{k=1}^{D} j_k * d_k$$

in which a substantial number of subtractions have been eliminated. Thus, A_0 will usually be saved in the dope vector. Another useful quantity is the *size* of the array, the product reduction of the shape vector, because this is exactly the length of a vector extracted from the array.

Suppose we have an array A with storage mapping function f_A and an array B defined in terms of A:

$$B(j_1,\ldots,j_{DB}) = A(k_1,\ldots,k_{DA})$$

where

$$k_i = a_{i0} + a_{i1}*j_1 + \ldots + a_{iDB}*j_{DB}$$

Then to compute the dope vector for B, we apply the linear transformation given by the equations above to the entries in the dope vector for f_A to yield:

$$\boxed{\begin{array}{l} \text{1) address } P_A + \sum\limits_{k=1}^{DA}((\sum\limits_{m=1}^{DB} a_{km}*iB_m) - iA_k)*dA_k \\ \text{2) dimensionality } DB \\ \text{3) shape } (nB_1,\ldots,nB_{DB}) \\ \text{4) origin } (iB_1,\ldots,iB_{DB}) \\ \text{5) stride } (dB_1,\ldots,dB_{DB}) \end{array}}$$

where

$$dB_m = \sum_{k=1}^{DA} a_{km}*dA_k$$

As an example, consider the dynamic definition

IDENTIFY /L1/ X(I) = Y(J,2*I)

The dope vector for X should be –

$$\boxed{\begin{array}{l} \text{1) address}_X\ P_X = P_Y + (J-1)*dY_1 + 2*dY_2 \\ \text{2) dimensionality}_X = 1 \\ \text{3) origin}_X = (1) \\ \text{4) shape}_X = (L1) \\ \text{5) stride}_X = 2*dY_2 \end{array}}$$

as we would expect.

IV. Optimizations

Let us now consider the optimization techniques which are applicable to the extended intermediate code. First, a large class of scalar optimizations carry over directly. Subsections A through D describe these. Second some new and fairly radical optimizations are possible; these are discussed in E and F.

A. Redundant Expression Elimination

The intermediate code for the expression

A*B(1:50) + A*C

is the following

$$\begin{aligned}
&V_A \leftarrow \text{getvector}(f_A)\\
&\text{def}[f_{BS}(i) = f_B(i),\ i=1,50]\\
&V_{BS} \leftarrow \text{getvector}(f_{BS})\\
&V_{A*B} \leftarrow V_A * V_{BS}\\
&V_C \leftarrow \text{getvector}(f_C)\\
&V_{A2} \leftarrow \text{getvector}(f_A)\\
&V_{A*C} \leftarrow V_{A2} * V_C\\
&V_{A*B+A*C} \leftarrow V_{A*B} + V_{A*C}
\end{aligned}$$

Clearly the second *getvector* on f_A is redundant and can be eliminated; on a vector register machine, this is akin to register allocation.

Complex reasoning on the part of the compiler may uncover other redundancies. For example, suppose it determines that

$$f_{BS} = f_C$$

In other words, these two functions map to the same set of locations. Then the compiler can reduce the intermediate code to

$$\begin{aligned}
&V_A \leftarrow \text{getvector}(f_A)\\
&V_C \leftarrow \text{getvector}(f_C)\\
&V_{A*C} \leftarrow V_A * V_C\\
&V_{A*B+A*C} \leftarrow 2 * V_{A*C}
\end{aligned}$$

a substantially more efficient code sequence.

B. Constant Folding

In its full generality, *constant folding* means shifting computations from run-time to compile-time. Although the opportunities to perform operations on constant arrays will be limited, they will crop up from time to time, particularly as initialized tables. Constant arrays can also be computed:

```
      DO 10 I=1,100
   10 A(I) = I
```

This loop can be replaced by a modification of f_A to reference the vector (1,2,3,...,100) initialized in the load module. If the vector is multiplied by a constant C before being used,

A = A*C

then the initialized vector can be changed to (C,2C,...,100C).

Constant folding can eliminate many of the set-up operations in vector code, since these operations depend on the primarily constant dope vectors. In the first intermediate code example in Section II, several conformity check operations appeared. These amount to comparing the sizes of the two arrays (recall that the *size* of an array is equal to the product reduction of its shape vector). If both sizes are compile-time constants, as they often will be, the conformity check can be eliminated.

C. Code Motion

Obviously the movement of loop-invariant array operations out of loops is just as desirable as its scalar analog. Not so obvious, however, is whether of not it will be applicable often enough to make it worth the effort. I claim that code motion will be extremely useful in compilers that include an automatic vectorization phase, as the following example demonstrates.

```
          DO 20 I = 1,N
             .
             .
             .
          DO 20 J = 1,M
             .
             .
             .
          B(I,J) = B(I,J) + A(J)*C(J)
       20 CONTINUE
```

Assuming there are no other computations affecting A, B, C, I, or J, we can vectorize the computation to produce

```
          DO 20 I = 1,N
             .
             .
             .
          B(I,*) = B(I,*) + A(*)*C(*)
          DO 20 J = 1,M
             .
             .
             .
       20 CONTINUE
```

The array multiplication can now be removed from the outer loop.

```
          T(*) = A(*)*C(*)
          DO 20 I = 1,N
             .
             .
             .
          B(I,*) = B(I,*) + T(*)
          DO 20 J = 1,M
             .
             .
             .
       20 CONTINUE
```

Many other scalar optimizations have valid analogs for vectors.

D. Vector Register Allocation

On a machine like the Cray-1 with chaining of vector operations, the importance of good local register allocation cannot be overstated. To get the maximum computational power, the execution units must have enough operands to keep busy. Retaining vectors in registers between operations is one way to achieve this. Many of the scalar register allocation algorithms can be used for vector allocation if they are suitably adapted. However, because chaining puts heavy requirements on vector registers and because the number of such registers is small, the emphasis will be on local rather than global allocation. Also the similarity between *getvector* operations and vector loads may leave register allocation less freedom of choice than its scalar counterpart.

E. Temporary Management Optimizations

Some of the really interesting optimizations are essentially new, arising from the scale of vector operations. An example is temporary management. In vector compilers, storage comes at a premium, and the generation of too many vector intermediate quantities can quickly lead to problems. For this reason, an extended Fortran compiler must carefully manage its temporary storage. Temporary arrays will probably need to be allocated and deallocated dynamically. Since vector registers may be thought of as temporaries and spillage out of them creates a need for memory, temporary management is closely related to register allocation.

A useful machine-dependent technique for reducing temporary storage requirements is *strip mining* [Lov77]. Suppose we have an array statement

```
A = A*B + C*D
```

that we wish to execute on a Cray-1. Suppose also that every array has length greater than 64 so, if each vector operation is executed through to the end, we are guaranteed to require more temporary storage than the registers provide. However, we will not need the extra storage if we process in groups of 64.

```
   DO 10 K = 1,LENGTH,64
   L = MIN(K+63,LENGTH)
   A(K:L) = A(K:L)*B(K:L) + C(K:L)*D(K:L)
10 CONTINUE
```

In this example, written in BSP Fortran, the temporaries generated during the computation can now reside entirely in vector registers – at most 64 locations are needed on any iteration of the loop. Scalar operations can be produced by strip mining with a strip width of one.

The mapping function approach to array storage, described earlier, can assist the management of temporaries, since this scheme permits temporary arrays with different names to occupy the same storage. This makes it possible for the optimization phase to ignore temporary management and use as many temporaries as it needs, treating the address parts of the storage mapping functions as variables. The storage management package will take care of overlaying temporary space at run time.

In any case, run-time storage management is a costly feature to include in a Fortran-based language. Two techniques may make the cost more palatable.

1) If the compiler allocates fixed-length and variable-length temporaries to different areas, the allocation of the fixed-length area can be done at compile time, eliminating some run-time overhead and permitting efficient code to be generated for access to those temporaries.

2) The number of fixed-length temporaries can be increased by strip mining with a width equal to the length of vector registers in the target machine.

F. Copy Avoidance

A somewhat radical approach to optimization of vector operations is based on the technique of *copy optimization* in the set theoretic language SETL [Scz75]. The idea is straightforward – aviod copying arrays whenever possible. In other words, copy only when forced to do so by the semantics of the language. Consider the following code sequence.

```
X(*) = A(*,K)
  .
  .
  .
B(*) = 5.0*X(*)
```

A copy is avoided (or at least delayed) if the compiler simply adjusts the storage mapping function for X to reference the storage for A. Thus, instead of

```
def[$f_{AS}(i) = f_A(i,K),\ i=1,L1$]
$V_A \leftarrow$ getvector($f_{AS}$)
putvector($f_X$,$V_A$)
  .
  .
  .
$V_X \leftarrow$ getvector($f_X$)
$V_B \leftarrow 5.0*V_X$
putvector($f_B$,$V_B$)
```

it would generate

```
def[$f_X(i) = f_A(i,K),\ i=1,L1$]
  .
  .
  .
$V_X \leftarrow$ getvector($f_X$)
$V_B \leftarrow 5.0*V_X$
putvector($f_B$,$V_B$)
```

There are two dangers in this scheme. First, the compiler must be wary of stores into the array A which may, as a side effect, change the value of X; X must be copied before any such store is executed. Second, the compiler must be careful not to lose the storage originally assigned to X, so it can perform copies quickly when they are forced.

It is possible to carry the idea of delayed operations quite a bit further. For example, if the only use of the array B computed above is in the statement

```
Y = SUM(B)
```

where SUM is the sum reduction function, it would be possible to delay the multiplication by 5.0 until after the reduction.

```
Y = 5.0*SUM(X)
```

A scheme very much like this one has been proposed for APL compilers, to help eliminate many of the storage remapping operations like rotate, transpose, and so on [GuW78].

V. Compiler Organization

Any extended Fortran compiler will probably be adapted to several machines, including some with no vector instructions. Thus we must consider the problems of compiling for both machine types. Scalar machine code should be structured differently from vector code, so the optimizers will not look exactly alike. Take the array expression

```
A(*) = A(*) + B(*)*C(*)
```

When expanded into vector intermediate code, this would be effectively reduced to:

```
T(*) = B(*)*C(*)
A(*) = A(*)+T(*)
```

Suppose that, after vector optimization, we wish to expand this to scalar code. The result might be the following.

```
   DO 10 I=1,length(A)
10 T(I) = B(I)*C(I)
   DO 20 I=1,length(A)
20 A(I) = A(I)+T(I)
```

The optimizing transformation known as *loop fusion* [AlC71] could now join these two loops.

```
   DO 20 I=1,length(A)
   T(I) = B(I)*C(I)
20 A(I) = A(I)+T(I)
```

Thereupon temporary array T could be reduced to a scalar.

If the original statement had been directly expanded, this result would have been obvious. However, it is not wise to perform these expansions right away, even when compiling for a scalar machine, because opportunities for redundant subexpression elimination might be missed. When programming in a high-level language, the programmer should feel free to concentrate on his algorithm, leaving small efficiency-related details to the compiler. In so doing, he may introduce some redundant array subexpressions, and it is easiest to eliminate these as early as possible.

Optimization for a vector machine will be performed on two levels. First, the intermediate code with vector set-up macros unexpanded will be optimized in an attempt to eliminate as many aggregate operations as possible. This done, the macros will be expanded into equivalent scalar form and re-optimized as scalar code. The aim of the second phase is primarily to improve calculations involving the subscript mapping functions. The stages of the vector compiler, then, are as follows.

0) lexical analysis and parsing
1) automatic vectorization
2) high-level optimization
3) expansion of access functions, strip mining
4) vector register allocation
5) scalar optimization
6) scalar register allocation
7) storage assignment and code generation

For a scalar machine, automatic vectorization will be skipped, and the expansion of access functions will be beefed up to expand all vector operations. Of course, the scalar optimization phase will follow up the expansion with extensive loop fusion. Phases of the compiler for a scalar machine:

0) lexical analysis and parsing
1) high-level optimization
2) scalarization – expansion of vector operations
3) global scalar optimization, including loop fusion
4) register allocation
5) storage assignment and code generation

It is interesting to observe that the expansion of vector operations from the vector intermediate code is fairly straightforward. For example

$$\begin{aligned}
&V_A \leftarrow \text{getvector}(f_A)\\
&V_B \leftarrow \text{getvector}(f_B)\\
&V \leftarrow V_A{*}V_B\\
&\text{putvector}(f_A,V)
\end{aligned}$$

could be expanded, using the template elements for f_A and f_B

address:	P_A,P_B
origin:	i_A, i_B
shape:	n_A,n_B
stride:	d_A,d_B

where A and B are one-dimensional, to the following pseudo-Fortran code.

```
   J = $P_B$
   DO 10 I = $P_A$, $P_A+n_A*d_A$, $d_A$
   @I = @I*@J
   J = J + $d_B$
10 CONTINUE
```

where @I indicates an indirect reference through the operand, a common feature of intermediate codes.

An increase in the level of language supported usually increases the burden on optimization. More work is required to make the generated code competetive with the best equivalent program written in a lower level language. But this should be viewed as a challenge. The state of the art in code improvement methods has been rapidly advancing over the past few years, so I can say with some confidence that Fortran with vector extensions can be compiled, by judicious application of these techniques, into excellent code for both vector and scalar machines.

Acknowledgments

Much of my thinking on the representation of vectors has been influenced by a model being developed at IBM Research by Barry Willner, John Huang, and Sandy Liles. My concept of storage mappings was partially worked out in conversations with Bob Johnson of LASL and Wayne Wilson of IBM. George Paul of IBM Research and Forest Baskett of LASL have provided me with a great deal of information about vectors, vector machines, and vector languages. To all of these people go my heartfelt thanks.

References

AlC72 F.E. Allen and J. Cocke, "A catalogue of optimizing transformations," *Design and Optimization of Compilers*, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, N.J., 1972, pp. 1-30.

Bas78 F. Baskett, private communication, 1978.

Bur77 Burroughs Corporation, "Implementation of FORTRAN," Burroughs Scientific Processor, 1977.

ErS76 A.P. Ershov and M.R. Shura-Bura, "Formation of programming in the USSR," English text edited by K. Kennedy, to appear in *Proc. International Research Conference on the History of Computing*, Los Alamos, N.M., 1976.

GuW78 L.J. Guibas and D.K. Wyatt, "Compilation and delayed evaluation in APL," *Fifth Annual ACM Symposium on Principles of Programming Languages*, Tuscon, Arizona, January 1978, pp. 1-8.

Lov77 D.B. Loveman, "Program improvement by source-to-source transformation," *J.ACM* 24, 1, January 1977, pp. 121-145.

LoM69 E.S. Lowry and C.W. Medlock, "Object code optimization," *Comm. ACM* 12, 1, January 1969, pp. 13-22.

PaW75 G. Paul and M.W. Wilson, "The VECTRAN language: an experimental language for vector/matrix array processing," Report G320-3334, IBM Palo Alto Scientific Center, August 1975.

PaW78 G. Paul and M.W. Wilson, "An introduction to VECTRAN and its use in scientific applictions programming," *Proc. LASL Workshop on Vector and Parallel Processors*, Los Alamos, N.M., September 1978.

Scz75 J.T. Schwartz, "Optimization of very high level languages I; value transmission and its corollaries," *J. Computer Languages, 1, Pergamon Press, 1975, pp. 161-194.*