AD-A256 656

NASA Contractor Report 189711

ICASE Report No. 92-46

ICASE

CONCURRENT FILE OPERATIONS IN A HIGH PERFORMANCE FORTRAN

Peter Brezany
Michael Gerndt
Piyush Mehrotra
Hans Zima

DTIC
ELECTE
OCT 29 1992
S B D

Contract Nos. NAS1-18605 and NAS1-19480
September 1992

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

92-28154

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23665-5225

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Concurrent File Operations in a High Performance FORTRAN*

Peter Brezany[a], *Michael Gerndt*[b], *Piyush Mehrotra*[c], *Hans Zima*[a]

[a]Department of Statistics and Computer Science,
University of Vienna, Bruennerstrasse 72, A-1210 Vienna, Austria
zima@par.univie.ac.at
[b]Central Institute for Applied Mathematics,
Research Centre Jülich (KFA) Postfach 1913, D-5170 Jülich, Germany
m.gerndt@kfa-juelich.de
[c]ICASE, MS 132C, NASA Langley Research Center,
Hampton VA 23681 USA
pm@icase.edu

Abstract

Distributed memory multiprocessor systems can provide the computing power necessary for large-scale scientific applications. A critical performance issue for a number of these applications is the efficient transfer of data to secondary storage. Recently several research groups have proposed FORTRAN language extensions for exploiting the data parallelism of such scientific codes on distributed memory architectures. However, few of these high performance FORTRANs provide appropriate constructs for controlling the use of the parallel I/O capabilities of modern multiprocessing machines. In this paper, we propose constructs to specify I/O operations for distributed data structures in the context of Vienna Fortran. These operations can be used by the programmer to provide information which can help the compiler and runtime environment make the most efficient use of the I/O subsystem.

*The work described in this paper is being carried out as part of the P2702 ESPRIT research project "An Automatic Parallelization System for Genesis" funded by the Austrian Ministry for Science and Research (BMWF). The research was also supported by the National Aeronautics and Space Administration under NASA contracts NAS1-18605 and NAS1-19480 while some of the authors were in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681.

1 Introduction

Distributed memory multiprocessors (DMMPs), such as Intel's Paragon and Thinking Machines' CM5, provide an attractive approach to high speed computing particularly because their performance can be easily scaled up by increasing the number of processors. The I/O bottleneck has been somewhat alleviated in these systems by powerful Concurrent Input/Output Systems (CIOSs) ([1, 2, 3, 9, 16]).

Hardware and software architectures of CIOSs provided by various DMMP vendors differ substantially. For example, the Concurrent File System^{TM} (CFS) developed by Intel for the iPSC/2 and iPSC/860 supercomputers [15] is based on an architecture which is straightforward to use while delivering high speed concurrent access to large data sets. The CFS is based on a technique called **striping**. The striping scheme allows a single file to be spread across multiple disks (striped) [3] so as to improve access speed and decrease congestion in communication links. Striping is done at the logical block level. For example the even numbered logical blocks of a file may be allocated to disk 0 while the odd numbered logical blocks are located on disk 1.

Despite significant advances in hardware, programming DMMPs has been found to be relatively difficult. Data and work have to be distributed among the processors and explicit message passing has to be used to access remote data.

In recent years, several languages extensions have been proposed to provide a high level environment for porting data parallel scientific codes to DMMPs. The fundamental goal with these approaches is to allow the user to specify the code using a global index space while providing annotations for specifying the distribution of data. The compiler then analyses such high level code and restructures the code into an SPMD (Single Program Multiple Data) program for execution on the target distributed memory multiprocessor. Work distribution is based on the owner-computes rule and non-local references are satisfied by inserting appropriate message passing statements in the generated code [7, 11, 19].

Most of these efforts are extensions of FORTRAN 77 [5, 6, 12, 14, 18, 20] or FORTRAN 90 [4, 13] and we collectively refer to them as as high performance FORTRANs. Recently, a coalition of groups from industry, government labs and academia formed the High Performance FORTRAN Forum to design a standard set of extensions to FORTRAN 90 along the lines described above [8].

Among these languages, only Vienna Fortran [20] and MPP [14] provide support for CIOSs. However, efficient use of the CIOSs is crucial for many applications, such as processing of seismic data and simulations of oil fields, and largely dictates the performance

Codes
Dist | Avail and/or Special
A-1

DTIC QUALITY INSPECTED 1

of the whole program.

Consider the situation where one program writes out an array and another program then reads the data into a target array. If we use the standard FORTRAN write statement to output the array to a sequential access file (most scientific codes use sequential access files only), the data elements are written out in the column-major order as defined by FORTRAN. When the source array is distributed across a set of processors, the processors need to synchronize and generally execute serially in order to preserve this sequence when writing out the elements of the array to secondary storage.

However, if the target array is distributed in the same manner as the source, both the output and the input may become more efficient if we do not maintain this prescribed sequential order. For example, each processor may - in parallel - write out the piece of the array that it owns, as a contiguous block. This data can then be subsequently read - also in parallel - into a similarly distributed target array.

If the distributed arrays are being written and read in the same program, for example as scratch arrays, then the compiler knows the distribution of the target array. In such a situation, it can choose the best possible order of elements on external storage so as to make both the input and output efficient. However, in general, files are used to communicate data between programs. In such situations, the compiler and runtime system *do not have any information about the distribution* of the target array and hence will have to use the standard order for the elements.

In this paper, we propose constructs which enable the user to provide some information about how the data to be stored in the files is going to be subsequently used. This information allows the compiler to parallelize read and write operations for distributed arrays by selecting a well suited data sequence in the files. Note that the language constructs described here operate on whole arrays rather than sections of arrays.

We present the concurrent I/O operations in the context of Vienna Fortran, a high performance FORTRAN language. Section 2 introduces some Vienna Fortran language constructs for specifying data distribution along with a formal model to describe these distributions. The next section presents the concurrent I/O operations being proposed here, while Section 4 provides some performance numbers to justify the need for these operations.

2 The Vienna Fortran Distribution Model

In this section we present the basic language features of Vienna Fortran. A full description of the entire language can be found in [20]. Vienna Fortran is based on a machine model

$\mathcal{M}$ consisting of a set P of processors with local memory, interconnected by some network, and a high performance file system. A Vienna Fortran program $\mathcal{Q}$ is executed on $\mathcal{M}$ by running the code produced by the Vienna Fortran Compiler on each processor of $\mathcal{M}$. This is called an **SPMD (Single-Program-Multiple-Data) model** [10].

Code generation is guided by a mapping of the **internal data space** of $\mathcal{Q}$ to the processors. The internal data space $\mathcal{A}$ is the set of declared arrays of $\mathcal{Q}$ (scalar variables can be interpreted as one-dimensional arrays with one element).

Definition 1 *Let $A \in \mathcal{A}$ denote an arbitrary array. The* **index domain** *of A is denoted by $\mathbf{I}^A$. The* **shape** *of the index domain $\mathbf{I}$, $shape(\mathbf{I})$, provides the extents in each dimension of the domain.*

2.1 Processors

In Vienna Fortran processors are explicitly introduced by the declaration of a **processor array**. The notational conventions introduced for arrays above can also be used for processor arrays, i.e., $\mathbf{I}^R$ denotes the index domain of a processor array R.

2.2 Distributions

A **distribution** of an array maps each array element to one or more processors which become the **owners** of the element, and, in this capacity, store the element in their local memory. We model distributions by functions between the associated index domains.

Definition 2 Distributions

Let $A \in \mathcal{A}$, and assume that R is a processor array. A distribution of the array A with respect to R is defined by the the mapping:

$$\delta_R^A : \mathbf{I}^A \rightarrow \mathcal{P}(\mathbf{I}^R) - \phi$$

where $\mathcal{P}(\mathbf{I}^R)$ denotes the power set of $\mathbf{I}^R$.

In Vienna Fortran the distribution of an array is specified by annotating the array declaration with a *distribution expression*. For example,

REAL $A_1(..)$, ..., $A_r(..)$ **DIST** *dex* **TO** *procs*

declares arrays $A_i, 1 \leq i \leq r$. The distribution expression *dex* defines the distribution of the arrays in the context of the given processor set *procs*. The distribution expression in its simplest form consists of a sequence of distributions, one for each dimension of the

data array. A set of intrinsic distribution functions are provided, including the commonly occurring block distribution which maps contiguous elements to a processor and the cyclic distribution which maps elements cyclically to the processor set.

Examples of arrays with some basic distributions are given below:

PROCESSORS R2D(16,16)
REAL A(256,4096) **DIST** (**BLOCK**, **BLOCK**) TO R2D
REAL B(256,4096,16) **DIST** (**CYCLIC**, **BLOCK**,:) TO R2D

The first statement declares an 16×16 two-dimensional processor array, $R2D$. The array A s declared to be distributed such that its first dimension is partitioned into blocks of size 16, while the second dimension is partitioned into blocks of size 256. These blocks are mapped to the corresponding processors of $R2D$; for example, the segment $A(49:64,3841:4096)$ is assigned to the processor $R2D(4,16)$.

The ":" in the distribution expression of B specifies that this dimension is not distributed, only the first two dimensions are distributed across the two dimensions of the processor array.

Vienna Fortran supports a wide range of facilities for distributing and aligning data arrays. Full details of these and other features of the language, including examples, can be found in [5, 20].

3 Concurrent Input/Output Operations

In this section, we describe the concurrent file operations provided in Vienna Fortran. The language distinguishes between two types of files: standard files and array files. Standard files are accessed via standard FORTRAN I/O statements whereas array files can be accessed via concurrent I/O operations only.

The first subsection describes the structure of the array files. The concurrent file operations are informally introduced in the next subsection while the last subsection specifies these operations formally.

3.1 Array Files

Input/Output statements control the data flow between program variables and the file system. The file system of machine $\mathcal{M}$ may reside physically on a host system and/or a CIOS.

Definition 3 *The file system $\mathcal{F}$ of machine $\mathcal{M}$ is defined by the union of a set of standard FORTRAN files $\mathcal{F}_{ST}$ and a set of array files $\mathcal{F}_{ARR}$.*

When transferring elements of a distributed array to an array file, each processor does input/output operations controlling the transfer of the local part of the array to or from the corresponding part of the file. A suitable file structuring is necessary to achieve high transfer efficiency.

Array files in Vienna Fortran may contain values from more than one array. Therefore, array files are structured into records. Each record contains an array distribution descriptor followed by a sequence of data elements associated with this array.

Definition 4 *An array file* $F \in \mathcal{F}_{ARR}$ *is a sequence of* **distributed array records** $< darec_1, darec_2, \ldots >$ *Each record can be associated with a distributed array,* A, *and has the form* $(\psi^A, \mathcal{O}^A)$, *where*

- ψ^A *is a distribution descriptor and has the structure* $(\mathbf{I}^A, \mathbf{I}^R, \delta_R^A)$. *Here,* δ_R^A *is the distribution used for writing out the sequence of data elements and* $\mathbf{I}^A$ *and* $\mathbf{I}^R$ *are the underlying array and processor index domains, respectively, used for defining this distribution.*

- $\mathcal{O}^A$ *is a sequence of data elements stored in this record.*

3.2 I/O Operations

Concurrent I/O operations supported by Vienna Fortran can be classified into three groups: data transfer, inquiry and file manipulation operations. These operations deal with whole arrays which are distributed across a set of processors. Thus, a global synchronization of the processors is required before they cooperate to execute the operation.

In this subsection, we informally describe the concurrent I/O operations supported by Vienna Fortran.

Writing to a File

The concurrent write statement, **CWRITE**, can be used to write multiple arrays to a file in a single statement. For each array a distributed array record is written onto the file. Vienna Fortran provides three forms of the concurrent write statement. These affect the order of data elements written out to the distributed array record.

(i) In the simplest form, the individual distributions of the arrays determine the sequence of array elements written out to the file. For example, in the following statement:

$$\textbf{CWRITE}\ (f)\ A_1,\ A_2,\ \ldots,\ A_r$$

where f denotes the I/O unit number and A_i, $1 \leq i \leq r$ are array identifiers. This form should be used when the data is going to be read into arrays with the "same" distribution as A_i. In this situation, the sequence of elements in the file are generated by concatenating the linearized local segments of A owned by the individual processors according to the increasing order of the linearized index of the processors. This is the most efficient form of writing out a distributed array since each processor can independently (and in parallel) write out the piece of the array that it owns, thus utilizing the I/O capacity of the architecture to its fullest.

(ii) Consider the situation in which the data is to be read several times into an array B, where the distribution of B is different from that of the array being written out. In this case, the user may wish to optimize the sequence of data elements in the file according to the distribution of the array B so as to make the multiple read operations more efficient. Additional parameters of the **CWRITE** statement enable the user to specify (a) the shape of the distributed array to which the read operation will be applied, and (b) its distribution. These additional specifications can then be used by the compiler to determine the sequence of elements in the output file.

If a shape is specified, the size of the arrays $A_1, \ldots, A_n$ has to be equal to the product of the extents of the specified index domain. The resulting rank and shape have to match the distribution specification. For example, the following statement can be used if A is a two dimensional array.

CWRITE (f, **PROCESSORS**='R2D(N,N)',
& **DIST**='(BLOCK,CYCLIC) TO R2D') A

Here, the elements of the array A are written so as to optimize reading them into an array which is distributed as *(BLOCK, CYCLIC)*. Depending on the sequence to be written, the processors (a) could synchronize so as to execute the correct sequence of the individual writes to secondary storage, or (b) could incur the overhead of redistributing the data internally before using a parallel write operation to output the data.

(iii) If the data in a file is to be subsequently read into arrays with different distributions or there is no information available about the distribution of the target arrays, the user may allow the compiler to choose the sequence of the elements to be written out. This is done by specifying 'SYSTEM' as the distribution in the **CWRITE** statement:

CWRITE (f, **DIST**='SYSTEM') $A_1, \ldots, A_r$

This allows the compiler and the runtime system to cooperate to determine the best possible sequence for writing out the data, given that there is no knowledge about distribution of the target arrays.

Reading from a File

A read operation to one or more distributed arrays is specified by a statement of the following form:

$$\textbf{CREAD}\ (f)\ B_1, B_2, ..., B_r$$

where again f denotes the I/O unit number and B_i, $1 \leq i \leq r$ are array identifiers. The operation reads the next r distributed array records in f. The data elements of the ith record are read into B_i. Note that the semantics of standard FORTRAN I/O operations has to be maintained. That is, if an array A is written out to a file and then read into another array B, the column-major linearization of FORTRAN arrays will determine which element of A is read into a given element of B. The actual transfer of data, thus, is done by taking into account the distribution descriptor of the ith record and the shape and the distribution of B_i.

Accessing a Distribution Descriptor

The distribution descriptor of the current distributed array record in the file can be accessed as follows:

$$\textbf{CDISTR}\ (f)$$

The value returned by this function can be used in the special Vienna Fortran case statement, **DCASE**, which allows decisions to be made based on the value of the distribution descriptors.

Other Operations

- **COPEN** (*colist*) - Open an array file.
- **CCLOSE** (*cclist*) - Close an array file*.
- **CSKIP** ($f, *$) - Skip to the end of file.
- **CSKIP** (f, n) - Skip n distributed array records.
- **CBACKARRAY** (f) - Move back to the previous array record.
- **CREWIND** (f) - Rewind the file.

*The operations **COPEN** and **CCLOSE** have the same meaning and the lists *colist* and *cclist* have the same form as their counterparts in FORTRAN 77.

- **CEOF** (f) - Check for end of file.

Note that the concurrent I/O operations supported by Vienna Fortran can be applied only to the special array files defined here, and conversely array files can only be accessed through these operations.

3.3 Operations on Array Files

In this subsection, we formally describe the semantics of the concurrent operations on array files.

Definition 5 *1. A file F can be viewed as a concatenation of two sequences*[†]

$$F = \overleftarrow{F} \ \mathbf{cat} \ \overrightarrow{F}$$

where $\overleftarrow{F}$ *is part of the file which has already been processed and* $\overrightarrow{F}$ *is the rest of the file. These components are not directly accessible to the programmer.* **cat** *is the operation of sequence concatenation.*

2. *The null sequence is denoted by the symbol* $<>$.

3. *If* $n - tuple = (item_1, item_2, ..., item_n)$, *then* $n - tuple \downarrow i$ *is the ith item of* $n - tuple$.

4. *If* $F = < rec_1, rec_2, ..., rec_n, ..., rec_m >$ *for* $0 \leq n \leq m$, *then*

$$\begin{array}{lcl} first(F) & = & rec_1 \\ rest(F) & = & < rec_2, ..., rec_m > \\ last(F) & = & rec_m \\ withoutlast(F) & = & < rec_1, ..., rec_{m-1} > \\ firstn(F, n) & = & < rec_1, ..., rec_n > \\ withoutfirstn(F, n) & = & < rec_{n+1}, ..., rec_m > \end{array}$$

5. $B :\Leftarrow \mathcal{O}^A$ *means the transfer of the file data elements denoted by* $\mathcal{O}^A$ *to the distributed array B.*

6. B **reorder** $:\Leftarrow (\mathcal{O}^A, \psi^A, \psi^B)$ *has the following interpretation:* $\mathcal{O}^A$ *is read from the file, then it is reordered into an intermediate sequence which matches* ψ^B, *and finally, this sequence is transferred to the distributed array B of the program.*

[†]The formal specification of the array file operations presented in this subsection, is partly based on the file model proposed by Tennent [17].

7. *Let* $\psi^A = (\mathbf{I}^A, \mathbf{I}^{R1}, \delta^A_{R1})$ *and* $\psi^B = (\mathbf{I}^B, \mathbf{I}^{R2}, \delta^B_{R2})$. *An equivalence relation,* $\equiv$, *is defined among distribution descriptors. We say,* $\psi^A \equiv \psi^B$, iff *shape* $(\mathbf{I}^A)$ = *shape* $(\mathbf{I}^B)$, *shape* $(\mathbf{I}^{R1})$ = *shape* $(\mathbf{I}^{R2})$ *and the two distributions* δ^A_{R1} *and* δ^B_{R2} *are equivalent.*

8. i/o-operation*(F)* $A_1, A_2, ..., A_m$ *is equivalent to:*

$$\begin{array}{ll} i/o - operation(F) & A_1 \\ i/o - operation(F) & A_2 \\ \ldots & \\ i/o - operation(F) & A_m \end{array}$$

Definition 6 *Array file operations are defined as follows*[‡]*:*

1. **Data transfer operations**

- **CWRITE** *(F) A is equivalent to:*

$$\begin{array}{lll} \overleftarrow{F} & := & \overleftarrow{F} \ \mathbf{cat} \ <(\psi^A, \ \mathcal{O}^A \ )> \\ \overrightarrow{F} & := & <> \end{array}$$

- **CWRITE***(F,* **SHAPE** = *'$(E_1, ..., E_n)$',* **PROCESSORS**=*'$R(N_1, ..., N_m)$',* **DIST**=*'dex') A*

  is equivalent to:

$$\begin{array}{lll} \overleftarrow{F} & := & \overleftarrow{F} \ \mathbf{cat} \ <(\psi^{new}, \ \mathcal{O}^A \ )> \\ \overrightarrow{F} & := & <> \end{array}$$

  where

$$\begin{array}{lll} \psi^{new} & = & (\mathbf{I}^{NEW}, \mathbf{I}^R, dex) \\ \mathbf{I}^{NEW} & = & [1:E_1] \times ... \times [1:E_n] \end{array}$$

- **CWRITE** *(F,* **DIST**=*'SYSTEM') A*

  is equivalent to:

$$\begin{array}{lll} \overleftarrow{F} & := & \overleftarrow{F} \ \mathbf{cat} \ <(\psi^{system}, \ \mathcal{O}^A \ )> \\ \overrightarrow{F} & := & <> \end{array}$$

  where ψ^{system} *is implementation defined.*

- **CREAD** *(F) B is equivalent to:*

[‡]Auxiliary operations, such as opening and closing files, are not included in the formal definition here.

if $(\psi^A \equiv \psi^B)$ then

$B :\Leftarrow first(\vec{F}) \downarrow 2$

else

B **reorder** $:\Leftarrow (first(\vec{F}) \downarrow 2, \psi^A, \psi^B)$

endif

$\overleftarrow{F} := \overleftarrow{F}$ **cat** $first(\vec{F})$

$\vec{F} := rest(\vec{F})$

where $\psi^A = (first(\vec{F}) \downarrow 1)$

2. **Inquiry operations**

- **CDISTR (F)** *is equivalent to:*

$$first(\vec{F}) \downarrow 1$$

- **CEOF** *(F) is equivalent to:*

$$\vec{F} = <>$$

3. **File manipulation operations**

- **CREWIND** *(F) is equivalent to:*

$$\vec{F} := \overleftarrow{F} \textbf{ cat } \vec{F}$$
$$\overleftarrow{F} := <>$$

- **CBACKARRAY** *(F) is equivalent to:*

$$\vec{F} := last(\overleftarrow{F}) \textbf{ cat } \vec{F}$$
$$\overleftarrow{F} := withoutlast(\overleftarrow{F})$$

- **CSKIP (F, ∗)** *is equivalent to:*

$$\overleftarrow{F} := \overleftarrow{F} \textbf{ cat } \vec{F}$$
$$\vec{F} := <>$$

- **CSKIP (F, n)** *is equivalent to:*

$$\overleftarrow{F} := \overleftarrow{F} \textbf{ cat } firstn(\vec{F}, n)$$
$$\vec{F} := withoutfirstn(\vec{F}, n)$$

4 Performance

In this section, we present some performance measurements to justify the need for user control over the manner in which data from distributed arrays is transferred to and from secondary storage.

Consider the following declarations:

PARAMETER (NP =...)
PARAMETER (N =...)

PROCESSORS P(NP, NP)
REAL A(N, N) **DIST** (*BLOCK*, *BLOCK*)

Here, A, is a $N \times N$ array, block distributed in both dimensions across an $NP \times NP$ processor array. Figure 1 shows the distribution of elements of the array A for the case of $N = 4$ and $NP = 2$.

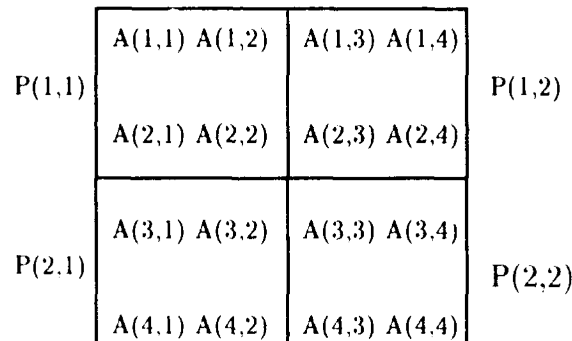

Figure 1: A two-dimensional block distributed array

If such an array is written out using a standard FORTRAN write statement, the semantics enforce the column-major linearization of the data elements. This would require close synchronization of the processors owning A to execute the write statement. Besides this serialization, another drawback is that each processors writes only small blocks of the individual columns. On most systems, such as the iPSC/860, the best performance for I/O operations is reached for large blocks. The same inefficiencies recur, if the data has to be subsequently read into a similarly block distributed two-dimensional array.

On the other hand, if we use the simplest form of the **CWRITE** statement as proposed in the last section, the sequence of the data elements in the file would be as follows:

A(1,1),A(2,1),A(1,2),A(2,2),A(3,1),A(4,1),A(3,2),A(4,2),A(1,3), A(2,3),A(1,4),
A(2,4),A(3,3),A(4,3),A(3,4),A(4,4)

Each process can thus write its local elements as one block, in parallel with the other processes. Similarly, reading the data into a similarly distributed array can also be executed in parallel.

In order to determine the overheads involved in writing out an array distributed as described above, we implemented five versions of the write statement on the Intel iPSC/860. The system consists of 32 processing nodes and 4 I/O nodes using CFS to manage the file system.

The first four versions of our experiment, preserve the standard FORTRAN linearization order, while the last uses the sequence suggested above.

In the first implementation, *CENT*, each process sends its local block of elements to a designated process which collects the entire array. This central process then writes the array out to the CFS using a standard FORTRAN write statement.

The next three implementations, SEQ0, SEQ1 and SEQ2 again preserve the column-major linearization of the array and use CFS's file modes 0, 1 and 2 respectively [9], to write out the array. In *SEQ0*, each process manages its own file pointer. All processes write unsynchronized to the same file. They position their file pointer to the appropriate position in the file for each subcolumn that they have to output.

The processes work with a common file pointer in version *SEQ1* and thus have to be closely synchronized. For each part of a column, the appropriate process performs the write while the other processes are waiting.

In *SEQ2*, the write operations are executed as collective operations. The columns are written sequentially. Thus, each process which owns a part of the column writes its part. Other processes perform the write with zero length information. The information written in such a collective operation is ordered in the output file according to the process numbers.

The last version, *NEW*, uses the implementation suggested in this paper. That is, instead of writing out the data in the column-major order, each process writes out its local piece as contiguous block. The processes perform a single collective write using the CFS's file mode 3.

Table 1 shows the times measured for a 1000×1000 array distributed blockwise across a 4×4 processor array. Since the performance depends heavily on whether the file to be written exists prior to the operation or not, we present timings for both cases. The problem is that if the file does not already exist, new disk blocks have to be allocated

Version	Including file creation	Pre-existing file
CENT	4.3	4.0
SEQ0	52.2	6.5
SEQ1	43.9	7.9
SEQ2	42.3	4.4
NEW	1.9	1.6

Table 1: Time (in secs) for writing out a distributed array

every time the file is extended. This is particularly an issue with the versions, *SEQ0*, *SEQ1* and *SEQ2* since each individual write for a part of the column extends the file.

It is clear from our experiments, that at least on the iPSC/860, that the version *NEW* performs better than the rest of the implementations. This indicates that I/O bound applications running on distributed memory machine may achieve much better performance if the user can provide information which would help the compiler and runtime system to choose the best possible sequence of the data elements written out to secondary storage.

The concurrent I/O operations described in the last section are currently being integrated into the Vienna Fortran Compilation System. We will report on the performance of these operations, in the context of actual applications, at a later date.

5 Conclusions

Vendors of massively parallel systems usually provide high-capacity parallel I/O subsystems. Efficient usage of such subsystems is critical to the performance of I/O bound application codes. In this paper, we have presented language constructs to express parallel I/O operations on distributed data structures. These operations can be used by the programmer to provide information which will allow the compiler and runtime environment to optimize the transfer of data to and from secondary storage. The language constructs presented here have been proposed in the context of Vienna Fortran, however, they can be easily integrated into any other high performance FORTRAN extension.

Acknowledgment

We would like to thank Barbara Chapman for her helpful comments and discussions. We would also like to thank one of the referees for the insightful and detailed comments.

References

[1] J.C. Admiraal and C. Pronk : *Distributed Store Allocation and File Management*, Microprocessors and Microsystems, Vol. 14, No 1, 10-16, January/February 1990

[2] R.K. Asbury, D.S. Scott: *FORTRAN I/O on the iPSC/2: Is There Read After Write?* , Proceedings of the DMCC 4, 129-132, 1989

[3] P. Beadle: *A Distributed File System for K2*, Technical Report No. 88/17, ETH Zürich

[4] S. Benkner, B. Chapman, H. Zima: *Vienna Fortran 90*, Proceedings of "Scalable High Performance Computing Conference", April 26-29, Williamsburg, 1992

[5] B. M. Chapman, P. Mehrotra, H. Zima: *Programming in Vienna Fortran*, Scientific Programming, Vol.1, No.1, 1992.

[6] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu: *Fortran D language specification*, Department of Computer Science Rice COMP TR90079, Rice University, March 1991

[7] H.M. Gerndt: *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*, Ph.D. Dissertation, University of Bonn, Austrian Center for Parallel Computation Technical Report Series ACPC/TR90-1, 1989

[8] High Performance FORTRAN Forum, *High Performance FORTRAN Language Specification*, Technical Report, Rice University, Houston, TX.

[9] *iPSC/2 and iPSC/860 Manuals*, Intel, 1990

[10] A.H. Karp: *Programming for Parallelism*, Computer 20(5), 43-57, May 1987

[11] C. Koelbel and P. Mehrotra: *Compiling global name-space parallel loops for distributed execution*, IEEE Transactions on Parallel and Distributed Systems, October 1991

[12] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 364-384. Pitman/MIT-Press, 1991.

[13] J. H. Merlin: *ADAPTing FORTRAN 90 Array Programs for Distributed Memory Architectures, Proceedings of the First International Conference of the ACPC*, September 1991, Salzburg, Austria

[14] D.M. Pase: *MPP FORTRAN Programming Model, Draft 1.0*, Technical Report, Cray Research, October 1991

[15] P. Pierce: *A Concurrent File System for a Highly Parallel Mass Storage Subsystem*, Proceedings of the DMCC 4, 155-160, 1989

[16] T.W. Pratt, J.C. French, P.M. Dickens, S.A. Janet, Jr. : *A Comparison of the Architecture and Performance of Two Parallel File Systems*, Proceedings of the DMCC 4, 161-166, 1989

[17] R.D. Tennent: *Principles of Programming Languages*, Prentice Hall, New Jersey, 1981

[18] P. S. Tseng: *A systolic array programming language*, Proceedings of the DMCC 5, 1125-1130, 1990

[19] H. Zima, H.-J. Bast, and H.M. Gerndt: *SUPERB - a tool for semi-automatic MIMD/SIMD parallelization*, Parallel Computing, 6, 1-18, 1988

[20] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald: *Vienna Fortran - a language specification*, ACPC Technical Report Series, University of Vienna, Vienna, Austria, 1992. Also available as ICASE INTERIM REPORT 21, MS 132c, NASA, Hampton VA 23681.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

[illegible]

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
September 1992

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
CONCURRENT FILE OPERATIONS IN A HIGH PERFORMANCE FORTRAN

5. FUNDING NUMBERS
C NAS1-18605
C NAS1-19480

WU 505-90-52-01

6. AUTHOR(S)
Peter Brezany, Michael Gerndt, Piyush Mehrotra, and Hans Zima

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
ICASE Report No. 92-46

9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING MONITORING AGENCY REPORT NUMBER
NASA CR-189711
ICASE Report No. 92-46

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Michael F. Card
Final Report

To appear in Supercomputing '92 (November 1992)

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unclassified - Unlimited

Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Distributed memory multiprocessor systems can provide the computing power necessary for large-scale scientific applications. A critical performance issue for a number of these applications is the efficient transfer of data to secondary storage. Recently several research groups have proposed FORTRAN language extensions for exploiting the data parallelism of such scientific codes on distributed memory architectures. However, few of these high performance FORTRANs provide appropriate constructs for controlling the use of the parallel I/O capabilities of modern multiprocessing machines. In this paper, we propose constructs to specify I/O operations for distributed data structures in the context of Vienna Fortran. These operations can be used by the programmer to provide information which can help the compiler and runtime environment make the most efficient use of the I/O subsystem.

14. SUBJECT TERMS
distributed-memory multiprocessors; concurrent input/output; data distribution; Fortran language extensions

15. NUMBER OF PAGES
17

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		