Parallel I/O Support for HPF on Clusters

Peter Brezany and Viera Sipkova
Institute for Software Science
University of Vienna
Liechtensteinstr. 22, A-1090 Vienna, Austria
{brezany,sipka}@par.univie.ac.at

Abstract

Clusters of workstations are a popular alternative to integrated parallel systems designed and built by a vendor. Besides their huge cumulative processing power, they also provide a large data storage capacity which allows efficient implementations of large-scale applications which are I/O intensive. This paper proposes a language, compiler and runtime software solution to the problem of parallel I/O on clusters. The proposal is presented in the context of High Performance Fortran and its compiler that is being developed at the University of Vienna. The system provides efficient support for explicit I/O operations on parallel files, accesses to sections of multi-dimensional arrays stored in parallel files, checkpoint/restart operations, and time step output and input operations. The implementation is based on our parallel I/O library implemented on top of MPI. The paper also presents experimental performance results using the implementation of the developed software on a Beowulf-class cluster system.

1. Introduction

In the past few years, clusters of workstations have emerged as an important and rapidly expanding approach to parallel computing. These systems often employ inexpensive commodity processors, open source operating systems and communication libraries and commodity networking hardware to deliver supercomputer performance at the lowest possible price. Clusters, besides their huge cumulative processing power, also provide a large data storage capacity which allows efficient implementations of large-scale applications which are input and output (I/O) intensive. Some examples of such systems are Berkeley NOW, HPVM, Beowulf, Solaris-MC, which are discussed in [4].

In order to effectively exploit the power of clusters, appropriate programming environments and development tools are a must. Within the long-term project Aurora [2], we are developing a set of I/O intensive applications (dynamic stochastic optimization in financial planning, quantum mechanical calculations of solids, and simulation of semiconductor processes and devices) and a parallel programming environment and tools for cluster architectures. High Performance Fortran (HPF) [10] is the main programming language used in this project.

So far, the HPF development has almost fully focused on providing support for compute intensive parts of scientific and engineering applications and only minimally addressed their I/O aspects in spite of the fact that the experience from the scientific application development in the past years has shown that I/O is a major performance bottleneck for many large-scale scientific applications running on parallel platforms.

Vienna Fortran [12] was one of the first HPF-like languages attempting to provide a high-level and efficient support for parallel I/O operations [18]. Several proposals were also elaborated directly for HPF [3, 19, 25], Fortran D [13], and MPP Fortran [6]. However, no language extensions proposed to the HPF Forum were included into the final HPF specification documents. Consequently, at present, HPF and the existing HPF programming environments do not provide an appropriate framework for programming and implementing I/O intensive applications.

So far, the major I/O software research efforts have been led along two lines: parallel file system research and parallel I/O library research. The most important results include PASSION [1], IBM/PIOFS [22], Galley [16], Panda [24], Disk Resident Arrays Library [15], PAFS [5], and ViPIOS [7]. An excellent description of the recent development in this field can be found in [14].

As an extension to the MPI standard [11], the MPI-IO interface was proposed and its functions were portable implemented by a software system called ROMIO [23]. Carns et al. [21] developed a parallel file system for Linux clusters, called the Parallel Virtual File System (PVFS) and implemented MPI-IO on top of it.

The work presented in this paper describes the design

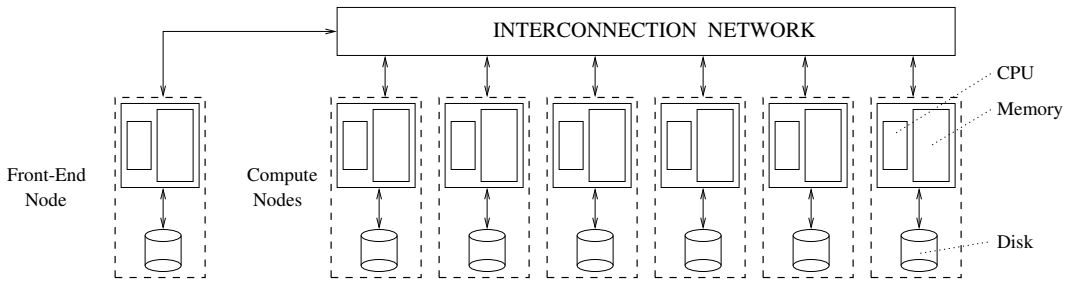


Figure 1. The Cluster Model

and implementation of a system providing the HPF users with the opportunity to access the parallel I/O functionality of clusters and other high performance systems at a high abstraction level. We addressed research issues associated with four different scenarios in which the application may need to access secondary memory frequently and potentially become I/O bound [8]:
(1) *explicit I/O operations* that are inherent in the application algorithm, (2) *timestep operations* - for time-dependent applications, snapshots of certain arrays are output at selected intervals over time; output data will then be read and analyzed by visualization tools, (3) *checkpointing* - for long-running production runs, it is desirable to save the state of certain program objects periodically (checkpoint) in order to be able to resume (restart) from a previous state in case of the application run interruption, and (4) *out-of-core computations* - their data structures are larger than the available main memory; therefore it is necessary to keep them on disk and transfer piecewise to the main memory and back.

Our prototype implementation has been done in the context of an HPF compiler called the Vienna Fortran Compiler (VFC) [9] we have developed at the University of Vienna and based on our parallel I/O library implemented on top of the MPI Linux implementation (MPICH 1.2.1).

This paper is organized as follows. Section 2 introduces the extensions of HPF to allow the HPF programs to access the cluster parallel I/O. The implementation of these extensions is briefly discussed in Section 3. Section 4 discusses the experimental performance results achieved on an eight-node Beowulf-class cluster system. The experiments are discussed in the context of a code implementing quantum mechanical calculations of solids and a synthetic code. Section 5 discusses related work. We briefly conclude in Section 6.

2 Language Support

Our design of the HPF parallel I/O concepts discussed in this section is based on the cluster model depicted in Figure 1. This model consists of a front-end node and a set of compute nodes which are interconnected by some kind of message passing network. Each node includes a processor, memory, and a disk[1]. I/O statements introduced in a program control the data flow between the program variables and the *files* which may reside physically on the front-end node disk or compute node disks.

Standard Fortran file operations introduced in an HPF program are processed by our HPF compiler, VFC, using a strategy in which one compute node, so called *master node*, reads and writes data from/to the standard Fortran files located on the front-end node disk and communicates with the computes nodes which own the appropriate parts of the data.

As a first step towards the specification of parallel I/O, the PARIO directive accompanying the OPEN statement is provided. By means of this directive, the user can indicate that a file is being opened which allows, under some conditions, concurrent accesses. Consequently, this OPEN statement and all the subsequent I/O statements operating on this file will be transformed by the VFC into the parallel form. With the PARIO directive, the user can introduce operations on *standard* parallel files (Subsection 2.1) which correspond to the sequential Fortran file model, or operations on *structured* parallel files, called *array files* (Subsection 2.2), which can be seen as a sequence of file data objects representing multidimensional arrays. The design supports parallel I/O operations including both distributed and non-distributed objects. Currently only unformatted parallel I/O is supported. The form of the directory path in the FILE specifier of the OPEN statement decides whether the file resides physically on the front-end node disk or compute node disks.

For the specification of checkpoint/restart and timestep operations, a set of special directives is provided (Subsections 2.3 and 2.4).

In this paper, we prefer an informal presentation (primarily based on examples) of the language constructs designed. A complete formal specification of their syntax can be found in [20].

[1] A node of a real cluster may include several processors and a set of disks.

```
      INTEGER, PARAMETER :: N1 = 2000, N2 = 2000
      INTEGER, DIMENSION(N1,N2) :: x, y, w
      INTEGER, DIMENSION(N1) :: z
      INTEGER :: u, s
!HPF$ PROCESSORS P(NP1,NP2)
!HPF$ DISTRIBUTE(BLOCK,BLOCK) ONTO P :: x, y
       ...
! Open new parallel file
!HPF$ PARIO
      OPEN (unit=u, file='/pariodir/stdfile.u',  &
            form='UNFORMATTED')
      WRITE (u) x, y(1:N1), s, z, w
      CLOSE (u)
       ...
!HPF$ PARIO
      OPEN (unit=u, file='/pariodir/stdfile.u',  &
            form='UNFORMATTED', status='OLD')
      READ (u) x, y, s, z(1:N1/2)
      CLOSE (u)
```

Figure 2. Examples of Specifying I/O Operations on Standard Parallel Files

2.1 I/O Operations on Standard Parallel Files

The introduction of I/O operations on standard parallel files requires only minimal modifications to HPF programs - insertion of PARIO directives immediately before the appropriate OPEN statements, as illustrated in Figure 2. We can see that the READ and WRITE operations may include scalars, distributed arrays, and non-distributed arrays. The fact that they operate on a parallel file will be determined by the runtime analysis code generated by VFC (Section 3). In this example, file *stdfile.u* will be distributed across compute node disks; multiple processes can have efficient, concurrent accesses to it. To open a file on the front-end node disk which can be shared by multiple processes, the file specifier should have, for example, the following form[2]:
file = *'nfs:./mypariodir/stdfile.u'*

The PARIO directive also allows a user to provide information such as file access patterns and file system specifics to direct optimization. Providing such hints may enable an implementation to deliver increased I/O performance or minimize the use of system resources. For example, in the directive:

!HPF$ PARIO, STRIPING_UNIT(32000)

the specifier STRIPING_UNIT specifies the suggested striping unit to be used for the new file which is to create and open [11].

[2]The *NFS* prefix expresses that the implementation is based on the NFS (Network File System) approach [26]

```
      ! Open new array file
D1: !HPF$ PARIO, ARRFILE
S1:       OPEN (unit=u, file='/pariodir/parrfile.u',  &
                form='UNFORMATTED')
D2: !HPF$ PARIO, NEWARR (N1, N2)
S2:       WRITE (u) x     ! Write array x of the shape (N1,N2)
D3: !HPF$ PARIO, NEWARR (N1), FNAME ('array_z')
S3:       WRITE (u) z     ! Write array z of the shape (N1)
D4: !HPF$ PARIO, NEWARR (N1, N2),  &
    !HPF$          FNAME ('array_w'),  &
    !HPF$          FSECTION (N1-49:N1,N2-49:N2)
          ! Allocate FRA of the shape (N1, N2), write contents
          ! of w(1:50,1:50) into the
          ! FRA section (N1-49:N1,N2-49:N2)
S4:       WRITE (u) w(1:50, 1:50)
           ...
S5:       CLOSE (u)                  ! Close array file
           ...
D5: !HPF$ PARIO, ARRFILE
S6:       OPEN (unit=u, file='/pariodir/parrfile.u',
                form='UNFORMATTED', status='OLD')
D6: !HPF$ PARIO, ARR (1)
S7:       READ (u) x                  ! Read the 1st FRA
D7: !HPF$ PARIO, FNAME('array_z'), FSECTION(1:5)
S8:       READ (u) z(11:15)      ! Read the section (1:5)
                                 ! from the second FRA
D8:   !HPF$ PARIO, FNAME('array_w'),  &
      !HPF$  FSECTION(101:150,101:150)
          ! write x(1:50,1:50) into the
          ! FRA section (101:150,101:150)
S9:       WRITE (u) x(1:50, 1:50)
S10:      CLOSE (u)                  ! Close array file
```

Figure 3. Examples of Specifying I/O Operations on a Parallel Array File

2.2 I/O Operations on Parallel Array Files

Fortran supports file organization known as *direct access* where each record is identified by an index number. If the file stores an array, it is possible to directly access an array element corresponding to a specific index value. However, programming accesses to the file elements corresponding to a multi-dimensional array section of a distributed array, as it is often required in parallel out-of-core applications, is rather awkward and may be highly inefficient. Therefore, to solve this problem, we introduce a new file type called *array file*. Array files are structured into records. Each record contains data elements associated with one array - therefore, this record is called *file resident array* (FRA). There is a descriptor associated with each FRA that contains information about its rank, its shape, and its type. The descriptor

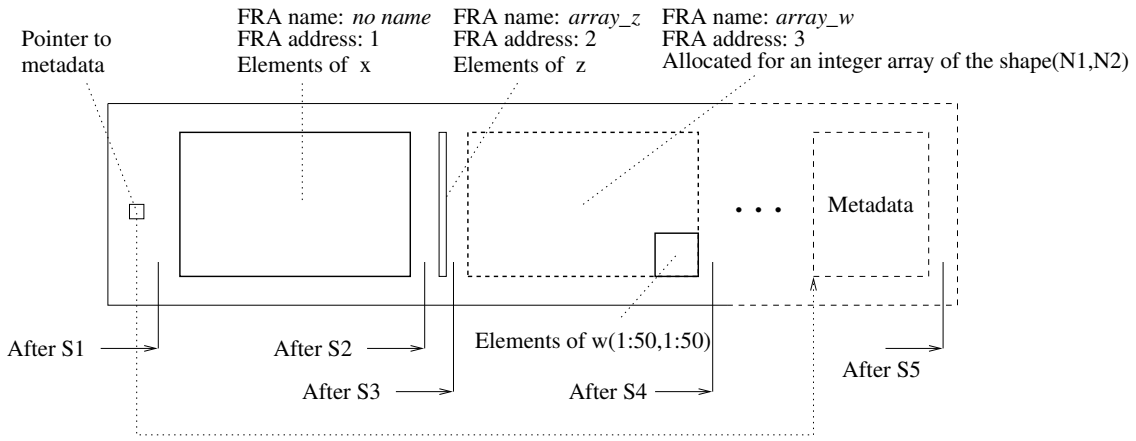


Figure 4. Storage of Arrays and Array Sections in an Array File

is stored in the metadata part of the file. The operation on an array file may specify the transfer of the whole FRA or its section; addressing the section elements is based on the information stored in the descriptor associated with this FRA. Each FRA is assigned an address which corresponds to its position in the file. When a new FRA is created, it can be optionally given a name.

Figure 3 illustrates the use of the array file operations involving the arrays declared in Figure 2, and Figure 4 sketches their effect on the array file as it can be viewed by an HPF programmer. The opening of a new array file is specified by the OPEN statement (line $S1$) which is preceded by the directive PARIO (line *D1*). The PARIO directive includes the ARRFILE specifier and may also specify a set of hints, as we briefly discussed in the last subsection. File *parrfile.u* is created and a pointer to the metadata section is allocated. The language constructs in *D2* and *S2* specify the allocation of a new integer FRA (by the clause NEWARR), of the shape ($N1$,$N2$) and the transfer of the section $\mathrm{x}(1 : N1, 1 : N2)$ into this FRA. The transfer of the section $\mathrm{z}(1 : N1)$ (the constructs in lines *D3* and *S3*) into the file can be described in a similar way; the FRA obtains the name *'array_z'* specified by the FNAME clause. The constructs in *D4* and *S4* specify the allocation of a new integer FRA of the shape ($N1$,$N2$) and writing the contents of the section $\mathrm{w}(1 : 50, 1 : 50)$ into its section $[N1 - 49 : N1, N2 - 49 : N2]$ which is specified by the FSECTION clause. Immediately before the array file is closed (line *S5*), a metadata record is appended to the file and its byte offset address is assigned to the pointer which was allocated during the OPEN operation.

The ARR specifier of the directive PARIO defines the current FRA visible and accessible from an open array file. For example, lines *D7* and *S8* specify reading the section $[1 : 5]$ from FRA having the address 2. The current and visible FRA can be also specified by a FNAME directive as shown in *D8*. An intrinsic function is provided which returns the FRA address corresponding to a given FRA name.

Other operations on array files include: SKIP $(u, *)$ - skip to the end of the array file, SKIP (u, n) - skip n FRAs, and BACK (u) - move back to the previous FRA.

2.3 Checkpoint/Restart Support

```
      CHARACTER (LEN=10) :: chname
      ...
      ! Define checkpoint object chname
!HPF$ CHECKP_DEF (chname) :: x, z, s
      ! Execute checkpoint operation
!HPF$ CHECKP_WRITE (chname)
      ...
      ! Restart from checkpoint
!HPF$ CHECKP_RESTART (chname)
      ! Deactivates all checkpoint objects
!HPF$ CHECKP_DEACTIVATE
```

Figure 5. Checkpoint/Restart Operations

A checkpoint operation saves the state of some program variables to disk to be restored during a restart operation. A checkpoint is logically an overwriting operation, as the latest checkpoint is the only one of interest. Our support for these operations does not explicitly work with files. The operations are performed on a data repository which is defined by the programming environment. We introduce the concept of *checkpoint object* that is considered as an abstract data object associated with a set of program variables that are involved in checkpoint/restart operations. A checkpoint object is defined by the CHECKP_DEF directive, where its argument represents the name of the checkpoint

object, and the right hand side consists of the list of variables related to the checkpoint object. Execution of a checkpoint operation and a restart operation is specified by the CHECKP_WRITE and the CHECKP_RESTART directive, respectively. The directive CHECKP_DEACTIVATE deactivates a specific checkpoint object or, by default, all defined checkpoint objects; however, this directive has no influence on the content of the data repository. The application of these constructs to the variables defined in Figure 2 is illustrated in Figure 5.

```
      CHARACTER (LEN=10) :: tname
      INTEGER :: tversion, obj
      ...
      ! Define timestep object tname
!HPF$ TIMESTEP_DEF (tname, OUT) :: x, z, s
      DO i= ...
        ...
!HPF$ TIMESTEP_WRITE (tname)              ! Write timestep
      END DO
      ! Deactivates timestep object tname
!HPF$ TIMESTEP_DEACTIVATE (tname)
      ...
      ! Define timestep object tname
!HPF$ TIMESTEP_DEF (tname, IN) :: x, z, s
      tversion = 1
      ! Read timestep version 1
!HPF$ TIMESTEP_READ (tname, tversion)
      tversion = 5; obj = 2
      ! Read array z from timestep version 5
!HPF$ TIMESTEP_READ_OBJECT (tname, tversion, obj)
      ! Deactivates timestep object tname
!HPF$ TIMESTEP_DEACTIVATE (tname)
```

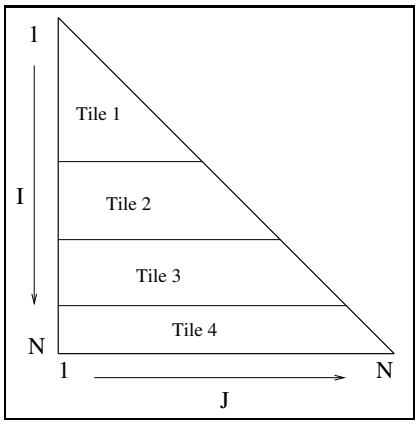
Figure 6. Timestep Operations

2.4 Time-Step Output and Input

The timestep operations are used for repeatedly writing data to disk for future analysis. Timestep is an append operation. Like the checkpoint operations, the directive TIMESTEP_DEF is used to define a timestep object which is associated with a number of program's variables. One of the optional intent arguments IN, OUT, or INOUT may be introduced to specify in which way the timestep objects will be accessed in the repository. The IN determines that the variables associated with the timestep object will be read from the existing timestep records in the data repository; OUT determines that these variables will be written into the new timestep records created in the data repository; and INOUT determines that the variables may be both read and written; however, in this case, the timestep records have already to exist. If the intent argument is omitted, the default value is INOUT.

```
      ...
      tilemax = 4
! ... Precompute parameters of iteration tiles:
!     tile_bounds, tile_sizes, low, and up ...
      DO tile = 1, tilemax     ! Out-of-Core Loop
!HPF$ PARIO, ARR(1), FSECTION(low(tile):up(tile))
        READ (pariounit) H(1:tile_sizes(tile))
!HPF$ INDEPENDENT, ON HOME(A2R(i))
        DO i = tile_bounds(tile), tile_bounds(tile+1)
          DO j = 1, tile_bounds(tile+1)-1
            H((i*(i-1)/2)+j-base) = H((i*(i-1)/2)+j-base) +
                                    A1R(j)*A2R(i) - A1I(j)*A2I(i) +
                                      B1R(j)*B2R(i) - B1I(j)*B2I(i)
          END DO
        END DO
!       ... update base ...
!HPF$ PARIO, ARR(1), FSECTION(low(tile):up(tile))
        WRITE (pariounit) H(1:tile_sizes(tile))
      END DO
      ...
```

Figure 7. Out-of-Core Code Fragment from Wien97



The directive TIMESTEP_WRITE initiates a timestep operation. Every timestep record written into the file represents one version of the timestep object. The TIMESTEP_READ directive initiates reading the specified version of the timestep object; and the TIMESTEP_READ_OBJECT directive initiates reading one variable from the specified version of the timestep object.

The directive TIMESTEP_DEACTIVATE deactivates a specific timestep object or, by default, all defined timestep

Table 1. Time (in secs) for Various I/O Implementations in the Out-Of-Core Code

Implementation Style	2 nodes	4 nodes	6 nodes	8 nodes
out-of-core loop (I/O by master node)	291.94	151.16	115.21	99.47
write sections (by master node)	40.00	35.69	34.49	33.80
read sections (by master node)	40.82	35.86	34.34	33.94
out-of-core loop (nfs I/O)	129.20	81.04	50.90	35.11
write sections (nfs mode)	0.24	0.28	0.29	0.29
read sections (nfs mode)	0.15	2.86	2.88	2.87
out-of-core loop (I/O parallel)	129.06	77.25	44.99	32.07
write sections (parallel)	0.07	0.06	0.05	0.05
read sections (parallel)	0.07	0.04	0.03	0.03

objects. The application of these constructs to the variables defined in Figure 2 is illustrated in Figure 6.

3 Implementation Notes

VFC is a source-to-source parallelization system that translates HPF programs into parallel Fortran 90/MPI message passing programs. In our approach, the overall parallel I/O software consists of two components: a set of I/O support modules included in VFC, and a parallel I/O runtime system. VFC processes the I/O specifications provided by the user and replaces them by code segments that perform the runtime analysis of file attributes and execute I/O operations. The I/O runtime system consists of a component supporting the master-node based I/O (Section 2, page 2), and the ViMPI library which is implemented in Fortran 90 on top of ROMIO [23]. The data repository supporting checkpoint/restart and timestep operations is implemented by a set of standard parallel files.

4 Performance Results

We run the experiments on the Beowulf-Cluster consisting of one front-end node and 8 compute nodes.
The parallel I/O support was applied to two model applications. The first application is a part of the Eigenvalue solver code taken from the software package WIEN97[3]. Figure 7 shows a part of the out-of-core version of the kernel loop nest of this code. In the original in-core version, the iteration space has a triangular form as depicted in the figure left. In the out-of-core version, the iteration space has been tiled in the way depicted in this figure, whereby we set N to 500500. In our experiments, we used three versions of I/O implementation: (1) using the standard Fortran I/O operations (VFC applies the master-node strategy), (2) using the array file approach - the file is stored on the front-end node disk (nfs I/O), and (3) using the array file approach - the array file is stored on the compute node disks (parallel I/O). The experimental performance results are shown in Table 1. For each version, we show the execution time of the whole out-of-core loop, and how much of this time was consumed for reading and writing array sections. These performance results show that the array file approach brings a great efficiency improvement, especially, in the case when the array file is stored on the compute node disks.

In the second I/O application study, we measured the performance of the checkpoint/restart operations introduced in Figure 5 and compared this performance with the cases when these high-level operations were implemented by explicit read and write operations. The experimental results in Table 2 show that only the parallel restart operations, in the experiments with 6 and 8 compute nodes, have visibly lower performance than the corresponding read operations. However, any implementation shows a dramatical performance improvement in comparison with the master-node I/O.

[3] WIEN97 [17] is a well established computer code used for quantum mechanical calculations of solids.

5 Related Work

So far, only a small effort has been put into the design of language extensions supporting parallel I/O in HPF. P. Brezany, et al. [18] introduced array files optimizing storage and transfer of distributed arrays; in their approach, operations on array sections were not supported. M. Snir [25] proposed an annotation for mapping of arrays to disks and I/O nodes. Bordawekar and Choudhary [3] proposed some directives for parallel I/O that can be used in conjunction with other HPF directives. In particular, they have introduced a directive for the specification of a logical disk array, an array of processors which really participate in performing I/O, a file template that is distributed across the disk array, and association of an HPF array template with a file template. M. Paleczny, et al. [13] proposed language constructs for distribution of out-of-core arrays across I/O

Table 2. Time (in secs) for Various Implementations of Checkpoint/Restart

Implementation Style	2 nodes	4 nodes	6 nodes	8 nodes
write (by master node)	694.82	611.91	591.37	578.48
write (nfs mode)	11.35	11.56	12.23	12.06
write (parallel)	1.44	0.78	0.85	0.78
read (by master node)	682.81	600.79	577.70	564.72
read (nfs mode)	3.79	5.99	6.26	4.93
read (parallel)	1.97	0.93	0.68	0.52
checkp_def (nfs mode)	0.016	0.014	0.016	0.020
checkp_def (parallel)	0.046	0.050	0.033	0.029
checkp_write (nfs mode)	11.42	12.42	12.24	12.30
checkp_write (parallel)	6.11	3.85	3.49	0.98
checkp_restart (nfs mode)	3.98	5.99	5.54	4.76
checkp_restart (parallel)	1.98	1.08	1.21	0.82

nodes. J. Nieplocha and I. Foster [15] developed a library optimizing accesses to two-dimensional out-of-core arrays; this library could be coupled to the VFC modules implementing accesses to array files.

6 Conclusions

Clusters of workstations can be equipped with 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 in HPF. The extensions to the HPF compiler and the parallel I/O runtime library have been fully implemented and the prototype implementation was evaluated on the eight-node Beowulf-class cluster system installed at our Institute. On the performance results gained from two model applications, we showed that the introduction of parallel I/O has radically improved the performance of these applications. The language concepts, presented here in the context of HPF, can be easily integrated into any other data parallel language.

References

[1] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, Syracuse University, ECE Dept., NPAC and CASE Center, September 1994.

[2] AURORA - Advanced Models and Software Systems for High Performance Computing. Special Research Program of the Austrian Science Fundation; http://www.vcpc.univie.ac.at/aurora/.

[3] R. Bordawekar and A. Choudhary. Language and compiler support for parallel I/O. IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems, April 1994.

[4] R. Buyya, editor. *High Performance Cluster Computing*, volume 1 and 2. Prentice Hall, New Jersey, 2000.

[5] T. Cortes. *Cooperative Caching and Prefetching in Parallel/Distributed File Systems*. PhD thesis, UPC: Universitat Politècnica de Catalunya, Barcelona, Spain, 1997.

[6] D.M. Pase, T. MacDonald, A. Meltzer. MPP FORTRAN Programming Model. Technical Report, Cray Research, March 1992.

[7] E. Schikuta, T. Fuerle and H. Wanek. ViPIOS: The Vienna Parallel Input/Output System. In *Proc. Euro-Par'98*, Southampton, England, 1998. Springer-Verlag, LNCS.

[8] E. Smirni, R.A. Aydt, A.A. Chien, and D.A. Reed. I/O Requirements of Scientific Applications: An Evolutionary View. In *Proc. of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 49–59, Syracuse, NY, IEEE Computer Society, 1996.

[9] S. B. et al. VFC - The Vienna HPF+ Compiler. In *Proc. of the Int. Conf. on Compilers for Parallel Computers*, Linkoping, Sweden, June-July 1998.

[10] H. P. F. Forum. High Performance Fortran, Version 2.0, February 1997.

[11] M. Forum. *MPI-2: Extensions to the Message–Passing Interface*, July 1997.

[12] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a language specification. ACPC Technical Report, University of Vienna, 1992.

[13] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler Support for Out-of-Core Arrays on Parallel Machines. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, February 1995.

[14] J. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publ., 2000.

[15] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O Library for Out-Of-Core Computations. In *Proc. of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 196–204. IEEE Computer Society Press, October 1996.

[16] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proc. of the 10th ACM International Conference on Supercomputing*, May 1996.

[17] P. Blaha, K. Schwarz, P. Dufek, and R. Augustyn. WIEN95, a full-potential, linearized augmented plane wave program for calculating crystal properties. Research Report, TU Wien, 1997.

[18] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima. Concurrent file operations in a High Performance FORTRAN. In *Proc. of Supercomputing '92*, pages 230–237, 1992.

[19] P. Brezany, P. Czerwinski, A. Swietanowski, and M. Winslett. Parallel Access to Persistent Multidimensional Arrays from HPF Applications Using Panda. In *High-Performance Computing and Networking Europe 2000*. Springer-Verlag, May 2000.

[20] Specification of the Extended I/O Support for HPF. Online document available at http://www.par.univie.ac.at/~brezany/par-io/pario-lang.ps, October 2000.

[21] P.H. Carns, W.B. Ligon III, R.B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proc. of the Externe Linux Track: 4th Annual Linux Showcase and Conference*, October 2000.

[22] IBM AIX Parallel I/O File System: Installation, Administration, and Use. IBM Document SH34-6065-00, 1995.

[23] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO portably and with high performance. In *Proc. of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[24] K. Seamons and M. Winslett. Multidimensional Array I/O in Panda 1.0. *Journal of Supercomputing*, 10(2):191–211, 1998.

[25] M. Snir. Proposal for I/O. Posted to HPFF I/O Forum, July 1992.

[26] Sun Microsystems, Inc. The NFS distributed file service. Online whitepaper available at http://www.sun.com/software/white-papers/wp-nfs/, March 1995.