비정형 응용을 위한 워크스테이션 클러스터링 환경에서의 병렬 입출력 시스템

(A Parallel I/O System on Workstation Clustering Environment for Irregular Applications)

노 재 춘 † 박 성 순 †† 알록 샤우드리 †††
(Jaechun No) (Sung-Soon Park) (Alok Choudhary)

요 약 워크스테이션 클러스터 환경은 그 가격 대 성능비가 일반적으로 MPPS보다 좋고, 그 소프트웨어나 하드웨어가 쉽게 이후에 개선될 수 있기 때문에 병렬처리 분야에서 새로운 대안으로 연구되고 있다. 본 논문에서는 "집단적 입출력 클러스터링(Collective I/O Clustering)"이라 불리는 워크스테이션 클러스터를 위한 실행시간 라이브러리의 설계 및 구현 방안을 제시한다. 이 라이브러리에서는 통신 및 입출력 시스템 하에서 완벽하게 통합되는 워크스테이션 클러스터 상에서 비정형 응용 프로그램의 입출력을 위해, 사용자에 친숙한 프로그래밍 모형을 제공한다. 이 집단적 입출력 클러스터링에서는 두 가지 형태의 입출력 방식이 가능하다. 첫 번째 입출력 방식에서 할당되는 모든 프로세서들은 연산 노드뿐만 아니라, 입출력 서버의 역할도 수행하는 형태이다. 두 번째 입출력 방식에서는 오직 일부분의 프로세서들만이 입출력 서버의 역할을 수행하는 형태이다. 그리고 본 논문에서는 통신과 입출력 비용을 최적화 하기 위해 압축과 소프트웨어 캐싱 기능을 집단적 입출력 클러스터링에 적용한 결과를 보인다. 모든 성능실험 결과는 아르곤 연구소에서 보유하고 있는 IBM SP2를 사용하여 얻었다.

Abstract Clusters of workstations (COW) are becoming an attractive option for parallel scientific computing, a field formerly reserved to the MPPs, because their cost-performance ratio is usually better than that of comparable MPPS, and their hardware and software can be easily enhanced to the latest generations. In this paper we present the design and implementation of our runtime library for clusters of workstations, called "Collective I/O Clustering". The library provides a friendly programming model for the I/O of irregular applications on clusters of workstations, being completely integrated with the underlying communication and I/O system. In the collective I/O clustering, two I/O configurations are possible. In the first I/O configuration, all processors allocated can act as I/O servers as well as compute nodes. In the second I/O configuration, only a subset of processors can act as I/O servers. The compression and software caching facilities have been incorporated into the collective I/O clustering to optimize the communication and I/O costs. All the performance results were obtained on the IBM-SP machine, located at Argonne National Labs.

1. Introduction

Clusters of workstations (COW) are becoming an attractive option for parallel scientific computing [1,2], a field formerly reserved to the MPPs, because their cost-performance ratio is usually better than that of comparable MPPS, and their hardware and software can be easily enhanced to the latest generations. The new high speed local area networks, such as ATM, Myrinet, or the Gigabit Ethernet, allow to build high performance-low cost clustered systems using

· 이 논문은 1998년 안양대학교 교내 연구의 부분지원에 의한 결과임
† 비 회 원 : 미국 아르곤연구소 연구원
jano@mcs.anl.gov
†† 종신회원 : 안양대학교 컴퓨터학과 교수
sspark@aycc.anyang.ac.kr
††† 비 회 원 : 미국 노스웨스턴대학교 전자전산공학과 교수
choudhar@ece.nwu.edu
논문접수 : 1998년 11월 5일
심사완료 : 2000년 3월 8일

workstations as a basic building block[3]. The large-scale parallel scientific applications can be solved in cost-effective way on the clusters of workstations. Most of those applications have tremendous I/O requirements [4], including check pointing of large-scale data sets, and writing of periodical snapshots for further visualization. Furthermore, a large subset of those applications are irregular applications, where accesses to data are performed through one or more levels of indirection [5]. Sparse matrix computations, particle codes, and many CFD described via indirections, exhibit this feature. A typical computational science analysis cycle for these applications involves several steps: mesh generation, domain decomposition, simulation, visualization and interpretation of results, and archival of data and results for visualization check pointing and postprocessing of results.

To circumvent the I/O problems, two solutions have been traditionally used: sequential I/O on one processor and data distribution to the other processors, and storing each processor local data to a local data file [6,7]. Both solutions have inherent problems associated: sequential I/O is a major bottleneck for the application performance, and local files must rely on some kind of preprocessing to create the local files from a single data file of the mesh and on some postprocessing to recombine the data into a global file canonically ordered. As the number of grid points being considered moves to the million-to-billion range, sequential recombination of the MBytes/GBytes of data into the anticipated terabytes of data simply becomes unfeasible.

In previous works[8,9] we presented the design and implementation of a high-performance runtime system that can support all types of irregular application's I/O on MPPs. One of the main goals in developing the scheme was to always maintain data in some global canonical order to avoid the explosive number of file formats and to enhance parallelism using collective I/O [10]. By using the runtime system, the application processes can cooperate in reading or writing data using a single parallel file that is always ordered to avoid pre/post processing steps and bottlenecks from the previous solutions described above. Two other major goals were to use a software caching scheme to enhance data reuse, and to use compression to reduce the communication time and storage space required for the application data sets. It was implemented on an Intel Paragon at Caltech and on the ASCI/red Teraflops machine[11] at Sandia National Lab., and the evaluation results were very promising[8].

In this paper we present the design and implementation of a runtime system for clusters of workstations. The I/O architecture of a COW usually relays on a set of I/O servers, having local disks, and a set of diskless nodes. The design of our runtime library perfectly fits this feature, as we distinguish two kind of processors: I/O servers and compute nodes. The leading idea is to distribute data over the I/O servers, while executing computation on compute nodes (of course, a server can also be a compute node.) All I/O details, such as data exchange, data distribution, and collective I/O, are transparent to the application programmer. Moreover, the runtime library provides a friendly programming model for the I/O of irregular applications on clusters of workstations, being completely integrated with the underlying communication and I/O system.

The rest of the paper is organized as follows: Section 2 presents some design motivations. Section 3 describes the design and implementation of collective I/O clustering operations of the runtime library, presenting two I/O configurations: in the first I/O configuration, all processors are clients and I/O servers, and in the second I/O configuration, a subset of processors will only be I/O servers. Section 3 also shows the design for the software caching scheme developed in our runtime library to enhance data reuse. Section 4 shows the performance results on the IBM/SP machine located at Argonne National Labs. Finally, some conclusions are shown in section 5.

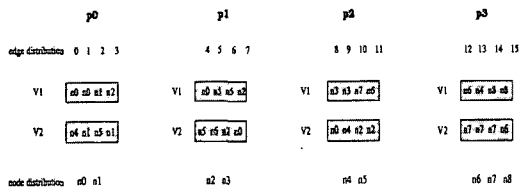
2. Motivations

We describe a typical irregular application, in which it sweeps all the edges of an unstructured mesh, to use a general application in our I/O system

as shown in Figure 1(a). In the application, an input mesh file is read, and then the edges and nodes are distributed over processors. We used block distribution mechanism to spread them to the processors. The loop termination value, *no_of_edges_partitioned_per_proc* represents the number of edges partitioned to a single processor. In the nested loops, *edge[j].V1* and *edge[j].V2* mean two nodes connected by an edge, *edge[j]*. The reference pattern is specified by *edge[j].V1* and *edge[j].V2*, called *indirection array*, and also these values are used to access to a global array. *X* is a data array which contains the physical values associated with each node. In this application, a node has an array consisting of 4 doubles, and other 2 floats.

```
read input_mesh_file containing all edges;
partition input_mesh_file among processors;
.....
for (iter=0; iter<NIter; iter++) {
  for (j=0; j<no_of_edges_partitioned_per_proc; j++) {
    Q=F(x[edge[j].V1], x[edge[j].V2]);
    /* F represents a math. function. */
    /* V1, V2 are two nodes connected by an edge */
    x[edge[j].V1] += Q;
    x[edge[j].V2] += Q;
  }
}
```

(a) A Typical Irregular Application



(b) An Example of The Edge and Node Distribution

Fig. 1 A Typical Irregular Application and Mesh Distribution

Figure 1(b) shows an example of the edge and node partition by using block distribution. In the processor 0, *n2, n4,* and *n5* are the remote indirection elements whose physical values mush be fetched from processor 1 and 2. All the remote values are fetched before the computation.

After the computations are finished, the data (physical values associated with a node) are written to a global array whose access pattern is determined by indirection elements, *edge[j].V1* and *edge[j].V2*, using the collective I/O clustering method.

The main objectives for the collective I/O clustering are as follows:

- *Provide flexibility needed to the various I/O configurations for a cluster of workstations.* In a workstation cluster, several I/O configurations are possible based on the available I/O servers and network bandwidth of the system. First, one or more local disks can be assigned to a single processor each, thus making it possible to act as an I/O server. Next, only a subset of processors are dedicated as I/O servers by receiving more memory space and aggregate bandwidth of the system. The collective I/O clustering is designed to support the both I/O configurations.
- *Provide scalability.* Different I/O configurations for a cluster of workstations can reveal different requirements with respect to the interconnect speed and number of I/O servers available. Some I/O configuration will perform best with well-balanced data distribution among I/O servers, while others will perform best with the minimum data migration by exploiting as much data locality as possible. Therefore, it is very important to support a wide range of I/O policies to provide highly scalable I/O internal structure with respect to the more processors and I/O servers.
- *Provide user-controllable stripe unit.* Appropriate declustering of I/O requests over I/O servers should be addressed to produce high performance I/O bandwidth [12] and has been successfully implemented in the several file systems [13,14,15]. In the collective I/O clustering, we use a user-controllable stripe unit which is specified by GF (Group Factor) in the file-creation time. Each processor's data domain for I/O is partitioned according to the GF size, and distributed over all I/O servers.
- *Provide compression facility.* Compression has

been traditionally used to reduce disk space requirement [16], but recently it has been applied to parallel applications managing large arrays with the aim of reducing the total execution time [17,18]. The collective I/O clustering combines compression facility to achieve two major goals: reducing disk space requirement, and reducing the total execution time.

We describe an overview of the collective I/O clustering on an irregular application in Figure 2. In the collective I/O clustering, a global file is organized as a sequence of sub files, each of which is stored in an I/O server's local disk. When a file is reading or writing, the appropriate schedule information is constructed. In the schedule information, each processor's data domain for I/O is determined, based on the two-phase method described in [10,19]. These data domains are logically partitioned into non-overlapping blocks, which may or may not span several local disks. The logical data domain partition is determined by GF ($1 \leq GF \leq$ *number of I/O servers*), which is given by users at the file-creation time. If GF is 1, the whole data domain of a processor is transferred to the appropriate I/O server. If GF is greater than 1, the data domain of a processor is divided into the contiguous GF number of blocks, and then these blocks are distributed across all I/O servers in round-robin fashion. Each I/O server stores a sub file, which consists of the blocks received from the processors.

There are three main advantages in the file structure supported by the collective I/O clustering. Unstructured data access pattern can be converted into simple contiguous I/O access pattern to the I/O servers. Also, for a file, by supporting a sequence of sub files, we can expect less I/O interferences, when reading or writing data. For example, with P processors trying to access a file and S number of I/O servers, each storing one sub file, only P/S processors contend to access the same sub file, instead of all P processors contending to access to a single file. Finally, since all the sub files still maintain global view of the file, we don't need any special postprocessing step (sort or merge) for the further processes.

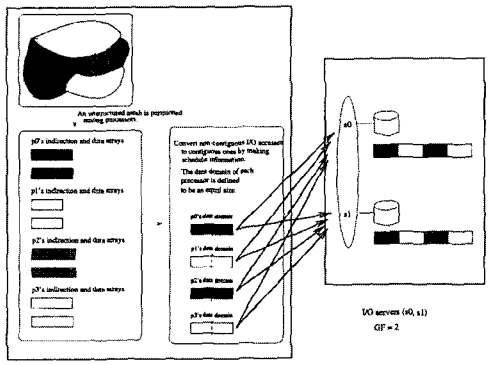


Fig. 2 An Overview of the Collective I/O Clustering on an Irregular Application

3. Implementation Details

In this section, we describe an overview of the collective I/O clustering. The method is organized into three main stages: Schedule construction a preprocessing stage, communications for exchanging the remote indirection and data elements between processors, and collective I/O clustering to perform I/O operation on I/O servers. As mentioned above, we configure a processor pool in two models. In the first model, the whole processors allocated act as I/O servers. In the second model, only a subset of processors act as I/O servers. In either cases, since applications can be run on all processors allocated, the collective I/O clustering facilitates the available resources as much as possible.

3.1 Schedule Construction Stage

The schedule stage describes the communication and I/O patterns required for each processor. The two main steps are involved in computing the schedule information. First, each processor is assigned a data domain into which it is responsible for reading and writing. The data domain of each processor is defined to be an almost same size to distribute I/O workload evenly. Next, the indirection elements in the local memory are individually scanned to decide which

processor is responsible for performing I/O for the corresponding data. Based on the scanning, the indirection elements are coalesced into a single message per destination processor. Once the schedule information is constructed, it can be repeatedly used in the irregular application whose access pattern does not change during computations. If the access pattern is changed, the appropriate schedule must be reconstructed, and then be used for the collective I/O clustering.

3.2 Communication Stage

In this stage, the remote indirection and data elements in each processor's local memory must be exchanged, according to the processor's data domain determined in the schedule. Let $\{I_0(0), I_0(1), I_1(2), I_1(0)\}$ and $\{d(I_0(0)), d(I_0(1)), d(I_1(2)), d(I_1(0))\}$ be the indirection and data arrays, respectively, stored in processor 0's local memory, and $\{I_0(2), I_0(3), I_1(3), I_1(1)\}$ and $\{d(I_0(2)), d(I_0(3)), d(I_1(3)), d(I_1(1))\}$ be the indirection and data arrays, respectively, stored in processor 1's local memory, where $I_i(j)$ and $d(I_i(j))$ are the indirection and data elements that must be assigned to processor i in the local position of j. In case of writing, $I_1(2)$, $I_1(0)$ and $d(I_1(2))$, $d(I_1(0))$ in the processor *0* are transferred to the processor 1. Similarly, in the processor 1, $I_0(2)$, $I_0(3)$ and $d(I_0(2))$, $d(I_0(3))$ are transferred to the processor 0. As a result, processor 0 collects $\{d(I_0(0)), d(I_0(1)), d(I_0(2)), d(I_0(3))\}$ and processor 1 collects $\{d(I_1(0)), d(I_1(1)), d(I_1(2)), d(I_1(3))\}$ into its local memory.

3.3 Collective I/O Clustering for Read and Write Operations

Based on the data domain arranged in the previous stage, the collective I/O clustering is executed. This stage takes buffer offset at which to start I/O, buffer length which is the number of data elements for I/O and pointer to the buffer. The additional information is also passed to the collective I/O clustering, such as the file name which is used for actual data and meta-data files, compression flag to check if the compression is applied for, number of I/O servers *(NumOfIO)*, and GF, which defines the number of partitions on each processor's data domain. For example, with GF to be 4, each processor's data domain is divided into 4 partitions. Hereafter, we call the partition as *block*. Among the processors allocated, as many as *NumOfIO* processors are configured as I/O servers, starting from the lowest-rank processor.

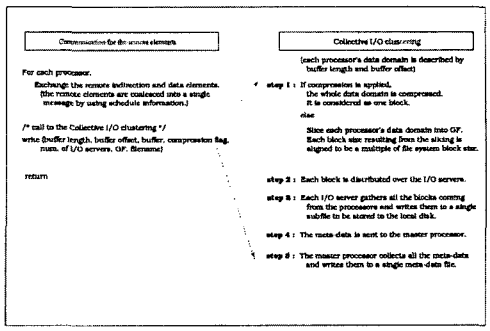


Fig. 3 Collective I/O Clustering for Write Operation

Figure 3 shows the steps involved in the collective I/O clustering for a write operation. In the step 1, the write operation begins with slicing each processor's data domain into GF size. Each block resulting from the slicing is then converted to be the same size over all processors, while making the block size to fit to multiple of file system block size. By maintaining the same size of blocks, we can easily determine the block number and local position in the block from which the data should be retrieved when we read data from the sub files. The performance can be also improved by aligning to the file system block size since the reading and writing of the fragments is replaced by writing into the file system block. In the step 2, all the blocks are distributed to the appropriate I/O servers in round-robin fashion. If GF is 1, each processor's entire data domain goes to an appropriate I/O server, with exploiting maximum data locality. If GF is the same size as *NumOfIO*, each processor's data domain is distributed over all I/O servers, with exploiting maximum data parallelism. When the compression is applied, the size of GF is restricted to 1 to maintain data integrity, since one bit of data loss can cause the entire I/O failure. The compressed data domain of each processor goes to the appropriate I/O server. In the step 3, each I/O server gathers all the

blocks coming from the processors and writes them to a single sub file to be stored into its local disk. If the compression is applied before writing, the whole length of a sub file is aligned to be a multiple of file system block size. In the step 4, the meta-data describing the characterization of the sub file is sent to the master processor. In the step 5, the master processor collects all the meta-data and writes them into single meta-data file. Each meta-data contains information about a sub file, such as the path to the local disk where the sub file is stored, uncompressed and compressed sub file lengths, block list containing the block size and relative offset in the sub file, and so on.

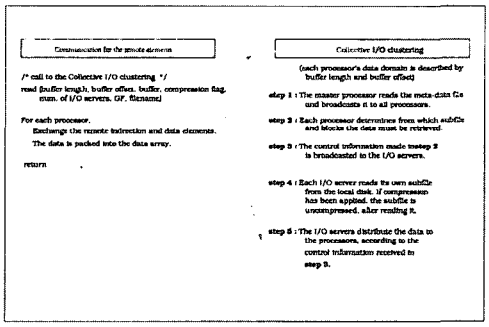


Fig. 4 Collective I/O Clustering for Read Operation

Figure 4 shows the steps involved in the collective I/O clustering for a read operation. In the step 1, the master processor reads the meta-data file and broadcasts it to all the processors. In the step 2, each processor's data domain is mapped to the blocks stored in the sub files, and also the appropriate I/O servers to retrieve the data are determined. In the step 3, the control information containing the local block number, position in the block and data count to retrieve is broadcasted to the I/O servers. In the step 4, each I/O server reads a sub file from its own local disk. If the compression has been applied, after reading it from the disk, the sub file is uncompressed. In the step 5, the I/O servers distribute data to all the processors based on the control information received in the step 3.

3.4 Software Caching Scheme combined with the Collective I/O Clustering

To reduce the I/O cost, we develop the software caching scheme for the collective I/O clustering. The motivation of the software caching scheme in irregular applications is that the same data may be accessed repeatedly during the execution of subsequent irregular loops. The data need not be written or read to or from files, by keeping it to processor's local memory and by using it in the subsequent loops. The basic goals and design of the software caching scheme are as follows: First and foremost, it is to reduce I/O to the maximum extent possible. To achieve this, a read and write I/O phases is divided into two read and write I/O phases, where the second read and write phase only accesses data that is "new-data". Second goal is to utilize the schedule information which is constructed in the beginning and only build incremental schedule for "new-data" when necessary. To satisfy these aims, we added the following two steps to the basic collective I/O clustering. 1) Reading data partially from files and redistributing data into appropriate locations of each processor; and 2) s/w caching phase to modify schedule information. In the write operation, redistribution step precedes the file write step. The execution pattern of the software caching scheme to perform a write operation in irregular applications is as follows:

```
mesh data 1 - create mesh data file which will be
  referred in loop1
mesh data 2 - create mesh data file which will be
  referred in loop2
schedule - compute schedule information for loop1 and
  loop2
loop1
write1 - write the data which has been referred in loop1
  but will not be referred in loop2
s/w caching - modify schedule information to access
  new data
loop2
write2 - write all data which has been referred in loop2
```

In write1 phase, some data which will not be used in a subsequent loop is written to files. Since only some portion of data is written to file, the I/O cost in the write1 is smaller than those in the write2. In s/w

caching phase, schedule information for the new loop is reconstructed, which includes the size of re-referred data and the modified data domain of each processor in the re-referred data.

4. Experimentation

The experimentation results for the collective I/O clustering were obtained on the IBM-SP at Argonne National Labs. The IBM-SP, called Quad, has 80 nodes, each running AIX 4.2.1 with patches. Each node has a 256MB of main memory and two of 4.5 GB disks. The nodes are interconnected by a IBM's high-performance switch.

4.1 Performance Evaluation without and with Compression

The collective I/O clustering was combined with the irregular application shown in Figure 1. Initially, the application reads 12M of edges and 8M of nodes. The nodes and edges are distributed over 64 processors, based on block distribution. As a result, each processor receives 192K of edges and 128K of nodes. A single node has a 40B of physical values consisting of 2 floats and an array of 4 doubles.

In the application, after the computations are finished, the physical values of the two nodes connected by an edge are written to the files. For example, in the Figure 1, processor 0 writes the physical values of the nodes *n0, n1, n2, n4* and *n5*, and processor 1 writes the physical values of the nodes *n0, n5, n2, n6*, and so on. The files are ordered by the node numbers, *n0, n1, n2, etc.* We obtained all the performance results by using 64 processors, and, in a single write iteration, 320MB of data was written to the files. We iterated the whole procedures 7 times. Each time we read the different edges and nodes, thus each iteration requiring to provide a new schedule before writing data to the files. In an iteration, we ran the computations 20 times and wrote the resulting data every 5 runs, by setting *NIter* in the Figure 1 to 20. Consequently, we ran the application by constructing 7 schedules and by writing 35 times of 320MB of data. The times were averaged over all runs.

Figures 5 and 6 show the I/O bandwidth and time components for the collective I/O clustering, in which the number of I/O servers is 4, 8, 16, 64, respectively. The I/O bandwidth is obtained by dividing 320 MB into the averaged schedule and write times over all runs.

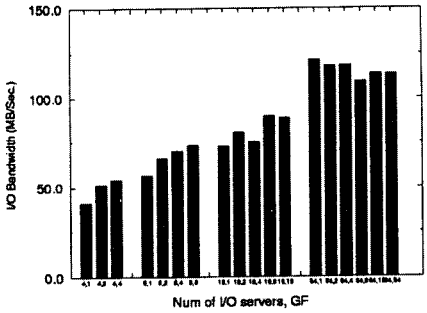


Fig. 5 I/O Bandwidth for the Collective I/O Clustering as a function of (number of I/O servers, GF)

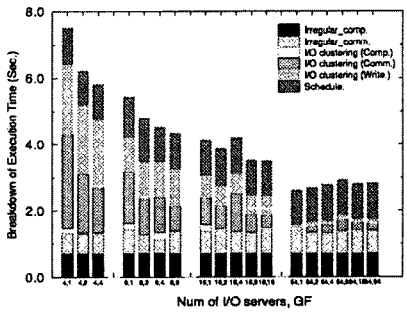


Fig. 6 Breakdown of Execution Time for the Collective I/O Clustering as a function of (number of I/O servers, GF)

The write time includes the times obtained by performing two stages, one is for communicating the remote indirection and data elements between processors, and the other for writing data to the files by using the collective I/O clustering. In the application shown in Figure 1, after communicating the remote elements for a write operation, each processor is responsible for writing 5MB of data. Since we spread the nodes and edges over processors

in block distribution, the times for the schedule and communication to access remote element do not change significantly, while varying the I/O nodes. Usually, irregular applications use a variety of heuristic data mapping to optimize the communication requirements[20]. With such schemes, the times for communicating remote elements, irregular_comp and irregular_comm in Figure 6, can be greatly reduced by allowing to access the local elements. In each I/O server, we varied GF size from 1 to the number of I/O servers. In the 4 I/O server configuration, as the GF size increases, the time for performing the collective I/O clustering becomes small. With the 1 of GF size, each processor sends 5MB of data to an I/O server, taking large communication overhead. As the GF size increases, the message size becomes small, *5MB/GF*, taking less communication overhead. In the same number of I/O servers, the write portion of the collective I/O clustering does not significantly change with the variation of GF, since each I/O server collects the same amount of data from all processors to make a sub file. Without block alignment, each sub file's size is *(TotalProcessors/NumberOfI-Oservers)*MsgSize*, where *MsgSize* is *Sizeof-DataDomainPerProc/GF*. As the number of I/O servers increases, each sub file's size becomes small, thus taking less time for the write portion of the collective I/O clustering. We can see the same behaviors with 8 and 16 I/O servers.

Figures 5 and 6 also show the I/O bandwidth and time components for the collective I/O clustering in which all processors are configured as I/O servers. In the Figures, the 1 of GF size shows the best performance. Since each processor writes the data collected in its data domain to the local disk, no communication requires in the collective I/O clustering. This case exploits as much data locality as possible. When the GF size increases, the I/O clustering requires the communication overhead, showing less performance results.

Figures 7 and 8 show the performance results with compression, while varying the number of I/O servers from 4 to 64. To maintain the data integrity, only GF of 1 is supported with the compression. We used *lzrw3* compression algorithm. Since the compression is incorporated in the collective I/O clustering, no effect occurs in the schedule and communication for accessing the remote elements. In the collective I/O clustering, after each processor's data domain is compressed, the communication and write operations are performed with the compressed data. Therefore, those times are significantly reduced, when compared to the Figures 5 and 6. However, the compression time takes more than 1 sec in every cases. This offsets the performance gains obtained by communicating the compressed data and writing it to the I/O servers. In the 4 I/O server configuration, the reduction in the communication and write operations exceeds the compression overhead, resulting in the better performance with the compression. However,

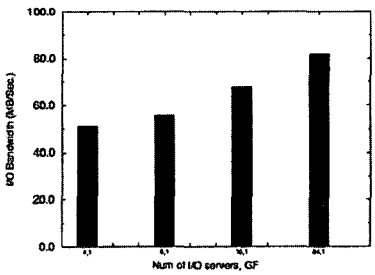


Fig. 7 I/O Bandwidth for the Collective I/O Clustering combined I/O Clustering combined with Compression as a function of (number of I/O servers, GF)

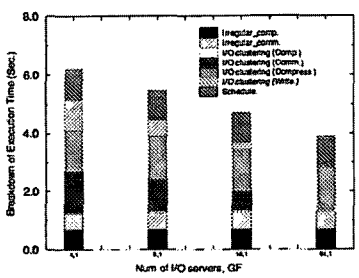


Fig. 8 Breakdown of Execution Time for the Collective I/O Clustering combined with Compression as a function of (number of I/O servers, GF)

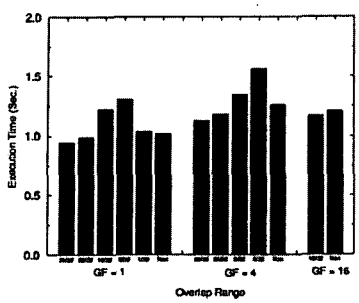


Fig. 9 Execution Time for the Software Caching scheme as function of overlap range and GF

as the number of I/O servers increases, the cost for the communication and write operations becomes small, and using the compression rather degrades the performance.

4.2 Performance Evaluation with Software Caching Scheme

We obtained the performance results for the software caching using 32 processors. Figure 9 shows the results, while varying the overlap range and GF size. The overlap range is defined as the data area overlapped between two irregular loops. In the Figure 9, *Non* shows the performance results obtained by without using the software caching scheme. In the experiments, we varied the overlap range from 1/32 to 31/32, and the GF size from 1 to 16. Figure 9 shows that the execution time increases until the overlap range approaches to 8/32, and then begins to drop. In the software caching scheme, when we change the overlap range from 31/32 to 1/32, the I/O cost increases due to the less overlap data area. On the other hand, the communication cost decreases as the overlap area is varied from 8/32 to 1/32. This reduction in the communication overhead exceeds the I/O cost, causing the reduced execution time.

5. Conclusions

We developed the runtime system, called Collective I/O Clustering, that can support the irregular applications on a cluster of workstations. In the system, two I/O configurations are possible; in the first configuration, all the processors act as I/O servers as well as compute nodes, and in the second configuration, only a subset of processors are dedicated as I/O servers. Further, the system supports a user-controllable stripe unit which is specified by GF in the file-creation time. We evaluated the collective I/O clustering on the IBM-SP at Argonne National Labs. We found that, in the first configuration, exploiting data locality produces the best performance by requiring no communication cost. On the other hand, in the second configuration, as the data in each processor's domain is more distributed over I/O servers, the performance results become better due to the less communication overhead. As an optimization, we combined the collective I/O clustering with the compression. As far as the reduction in the communication and I/O costs outperforms the compression overhead, the results with compression are improved, when compared with those without compression. However, with the small communication and I/O costs, compression overhead becomes a burden to produce better performance. It should be noted that, several aspects such as the network and I/O bandwidth of the system and size of the data set to be written affect the compression performance. Also, choosing the compression algorithm with less overhead is very critical to improve performance. Finally, in the software caching scheme, by accessing only the overlap data area between two irregular loops, the I/O cost can be optimized.

References

[1] A.Arpaci-Dusseau and et.al. "High-Performance Sorting on a Network of Workstations," In *SIGMOD'97 Proceedings*, Tucson, Arizona, USA, May 1997.

[2] P.Skordos. "Parallel Simulation of Subsonic Fluid Dynamics on a Cluster of Workstations," In *Proceedings of the 4th IEEE Int. Symp. on High Performance Distributed Computing*, Virginia, USA, Aug. 1995.

[3] T.Anderson, D.Culler, and D.Patterson. "A Case for Network of Workstations: NOW," IEEE Micro, Feb. 1995.

[4] J.T.Poole. "Preliminary Survey of I/O Intensive

Applications," Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.
[5] S.D.Sharma, R.Ponnusamy, B.Moon, Y.S.Hwang, R.Das, and J.Saltz. "Run-time and Compile-time Support for Adaptive Irregular Problems," In *Supercomputing*, IEEE Press, Nov. 1994.
[6] L.A.Schoof and V.R.Yarberry. "ExodusII: A Finite Element Data Model," Technical Report SAND 94-2137, Sandia National Lab., Dec. 1994.
[7] R.Thakur, E.Lusk, and W.Gropp. "I/O Characterization of a Portable Astrophysics Application on the IBM SP and Intel Paragon," Technical Report MCS-P534-0895, Argonne National Lab., Aug. 1995.
[8] J.Carretero, J.No, S.Park, A.Choudhary, and P.Chen, "Compassion: A Parallel I/O Runtime System Including Chucking and Compression for Irregular Applications," In *Proceedings of the International Conference on High-Performance Computing and Networking*, Amsterdam, Holland, Apr. 1998.
[9] J.No, S.S.Park, J.Carretero, A.Choudhary, and P.Chen, "Design and Implementation of a Parallel I/O Runtime System for Irregular Applications," In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, USA, Mar. 1998.
[10] R.Bordawekar, J.M.Rosario, and A.Choudhary. "Design and Evaluation of Primitives for Parallel I/O," In *Proceedings of Supercomputing*, pp.452-461, 1993.
[11] T.Mattson and G.Henry, "The ASCI Option Red Supercomputer," In *Intel Supercomputer Users Group. Thirteenth Annual Conference,* Albuquerque, USA, Jun. 1997.
[12] T.H.Cormen and D.Kotz, "Integrating Theory and Practice in Parallel File Systems," Technical Report PCS-TR93-188, Dept. of MCS, Dartmouth College, Mar. 1993.
[13] F.E.Bassow. Installing, Managing, and using the IBM AIX Parallel I/O File System, IBM Document No. SH34-6065-00, IBM Kingston, NY. Feb. 1995.
[14] P.F.Corbett, S.J.Baylor, and D.G.Feitelson. "Overview of the Vesta Parallel File System," In *Proceedings of '93 Workshop on Input/output in Parallel Computer Systems*, pp.1-16, Dec. 1993.
[15] Intel Corporation. *Paragon User's* Guide, Jun. 1994.
[16] T.A.Welch. "A Technique for High Performance Data Compression," *IEEE Computer*, 17(6):8-19, Jun. 1984.
[17] National Center for Supercomputing Applications. HDF reference manual version 4.1. Technical Report, University of Illinois, 1997.
[18] K.E.Seamons and M.Winslett. "A Data Management Approach for Handling Large Compressed Arrays in High Performance Computing," In Proceedings of the *Fifth Symposium on the Frontiers of Massively Parallel Computations*, pp.119-128, Feb. 1995.
[19] R.Thakur, A.Choudhary, R.Bordawekar, S.More, and S.Kudatipidi, "Passion: Optimized I/O for Parallel Systems," *IEEE Computer*, Jun. 1996.
[20] R.Das, M.Uysal, J.Saltz, and Y.S.Hwang. "Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures," *Journal of Parallel and Distributed Computing*, 22(3):462-479, Sep. 1994.

노 재 춘

1985년 2월 이화여자대학교 전자계산학과 졸업(학사). 1993년 12월 Western Illinois 대학교 전산학과 석사. 1999년 5월 Syracuse 대학교 전자전산공학과 졸업 박사. 1999년 8월 ~ 현재 Argonne 연구소에 근무하면서 Science Data Management Project 참여. 관심분야는 병렬입출력

박 성 순

1984년 2월 홍익대학교 전자계산학과 졸업. 1987년 2월 서울대학교 대학원 계산통계학과 졸업(석사). 1994년 2월 고려대학교 대학원 전산과학과 졸업(박사). 1988년 2월 ~ 1990년 7월 공군사관학교 전산학과 전임강사. 1996년 12월 ~ 1998년 2월 Northwestern 대학교 Post Doc. 1999년 5월 ~ 1999년 8월 IBM T.J. Watson 연구소 방문연구원. 1994년 3월 ~ 현재 안양대학교 부교수. 관심분야는 병렬입출력시스템, Clustering File System, 병렬 컴파일러

알록 샤우드리

1982년 6월 인도 Birla Institute of Technology and Science 전자전산공학과 졸업(학사). 1986년 2월 Massachusetts 대학교 전자전산공학과 졸업(석사). 1989년 8월 illinois 대학교(Urbana-Champaign) 전자전산공학과 졸업(박사). 1989년 8월 ~ 1996년 8월 Syracuse 대학교 전자전산공학과 부교수. 1996년 8월 ~ 현재 northwestern 대학교 전자전산공학과 부교수. 관심분야는 고성능 컴파일러 및 실행시간시스템, 고성능 DB, 병렬 입출력, 고성능 컴퓨터구조