The Data Diffusion Space for Parallel Computing in Clusters

Jorge Buenabad-Chávez and Santiago Domínguez-Domínguez

Sección de Computación
Centro de Investigación y de Estudios Avanzados del IPN
Ap. Postal 14-740, D.F. 07360, México
{jbuenabad,sdguez}@cs.cinvestav.mx

Abstract. The data diffusion space (DDS) is an all-software shared address space for parallel computing on distributed memory platforms. It is an extra address space to that of each process running a parallel application under the SPMD (Single Program Multiple Data) model. The size of DDS can be up to 2^{64} bytes, either on 32- or on 64-bit architectures. Data laid on DDS diffuses, or migrates and replicates, in the memory of each processor using the data. This data is used through an interface similar to that used to access data in files.
We have implemented DDS for PC clusters with Linux. However, being all-software, DDS should require little change to make it immediately usable in other distributed memory platforms and operating systems. We present experimental results on the performance of two applications both under DDS and under MPI (Message Passing Interface). DDS tends to perform better in larger processor counts, and is simpler to use than MPI for both in-core and out-of-core computation.

1 Introduction

Today PC clusters are widely used as platforms for parallel computing. Both message-passing and distributed shared memory environments are available for developing parallel applications on these platforms. Except for relatively simple communication patterns, message-passing programming is complicated; the programmer must specify when and which data to pass between which processing nodes. It is still more complicated for out-of-core computation, since the programmer must specify, or know, the data partitioning in disk space. However, message-passing libraries, such as MPI (Message Passing Interface) [12] and PVM (Parallel Virtual Machine) [15], are widely used because they do not require special hardware or operating system support.

A distributed shared memory (DSM) simplifies parallel programming because the location of data is not an issue. *Shared data* moves between processing nodes automatically and according to the access pattern of each application. Most DSM designs require either hardware or operating system support, which is, nonetheless, readily available in most hardware platforms and operating systems. If a DSM supports mapping files onto the *shared memory*, out-of-core

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 61–71, 2005.

computation is as simple to program as in-core computation. This will be most useful in 64-bit architectures, as in 32-bit architectures only 4 GB are available, while out-of-core applications today range in the hundreds of GB.

In this paper we present the data diffusion space (DDS), an all-software shared address space for parallel computing on clusters. It is an extra address space to the virtual address space of each process running a parallel application. DDS is for shared data only, which the programmer must explicitly specify as such through simply declaring it within a `C struct` declaration. Shared data automatically diffuses, or migrates and replicates, in the memory of each processor using the data, under a multiple-readers-single-writer protocol.

The size of DDS can be up to 2^{64} bytes, either on 32- and on 64-bit architectures. Hence shared data may not all be resident in memory. Some data will be in the disk space of processing nodes. However, the programmer uses the same interface to gain access to shared data (without specifying any location for data). This interface is similar to that used to access data in files. For a read, the programmer first calls $DDS\_Read()$; for a write the programmer first calls $DDS\_Write()$. The programmer then uses the data as it uses data in its local address space. After using the data the programmer must call $DDS\_UnRead()$ or $DDS\_UnWrite()$, respectively.

Data diffusion takes place by dynamically mapping data onto the memory of each processor using the data. Under out-of-core computation, DDS also maps shared data onto disk space in each processing node. These applications are likely to improve their performance under DDS, because DDS first tries to satisfy data requests from the memory of other nodes, instead of remote disk space.

In Section 2 we present related work. In Section 3 we present the architecture of DDS and its programming model. In Section 4 we show some empirical data on the performance of DDS compared to that of MPI for in-core and out-of-core applications. We offer some conclusions and describe future work in Section 5.

2 Related Work

A useful classification of DSM systems is that based on whether the implementation is all-hardware, mostly hardware, mostly software, or all-software [6]. All-hardware DSM moves data between processing nodes by hardware only, and at a fairly small granularity of typically 16 to 128 bytes. It includes cache-coherent non-uniform memory access (CC-NUMA) architectures, such as DASH [7] and Origin [9], and data diffusion architectures (also known as cache only memory architectures, or COMAs), such as DDM [19] and COMA-F [5]. In CC-NUMAs, data moves to the cache of each using processor, whether the data is *local* (resident in the nearest main memory node to a processor) or *remote*. In COMAs, the organisation of main memory is associative, and thus data moves to main memory nodes, and from these into processor caches, if available.

Mostly hardware DSM also moves data by hardware at a fairly small granularity, but little of its operation (e.g., gaining access to a memory region) is carried out by system software. Examples include Alewife [1] and KSR-1 [4]. Mostly

software DSM is the well known virtual shared memory based on paging. Based on commodity virtual memory hardware, it has been widely investigated and improved. The first representative, IVY [8], adopted sequential consistency as its memory consistency model, incurring in general a significant communication overhead to keep data coherent. This overhead has since been reduced through the adoption of more efficient consistency models [11], such as release consistency and lazy release consistency, and optimisations relating to the implementation of the DSM [17].

All-software DSM does not rely on any hardware support other than network communication hardware. Access to shared data is controlled by software primitives (linked to the application) whose invocation is instrumented/coded either by a compiler or the application programmer. All-software, compiler assisted DSM includes Orca [2] Shasta [16], Midway [3], and CAS-DSM [10]. The C Region Library (CRL) [6] is also all-software DSM but with no compiler support. The programmer must call CRL procedures to map and gain access to shared data, and also to relinquish access to, and unmap, shared data.

DDS is similar to CRL regarding the use of shared data. However, the mapping of shared data in DDS is made only once. Another difference is that DDS manages a 2^{64} byte shared address space, both in 32- or in 64-bit architectures.

3 The Data Diffusion Space

The data diffusion space (DDS) was designed to simplify the programming of parallel applications under the SPMD (Single Program Multiple Data) model. Under this model, a process is created on each processing node to run a parallel application. With DDS, the DDSP process is also created, and runs, on each processing node. DDS is organised into a library to which a parallel application is linked.

3.1 Architecture

Figure 1 shows the DDS architecture. The data diffusion space is extra to that of each process running a parallel application. Data in the diffusion space is dynamically mapped onto the address space of whichever application process is using the data. We will use the term *shared data* to refer to data in the diffusion space from now on.

When an application process requests shared data, and this data is not resident in its local memory, the DDSP process requests the data from a remote memory node (as described in Section 3.2). When the data arrives, it is placed somewhere in the address space of the application by DDSP. The address where the data was placed is given back to the application through the DDS interface (as described in Section 3.3). DDSP processes communicate through TCP sockets, using blocks of up to 64 KB.

3.2 Protocol

Shared data diffuses under a multiple-readers-single-writer data coherency protocol. For a read request, a copy of the data is obtained; for a write request, an

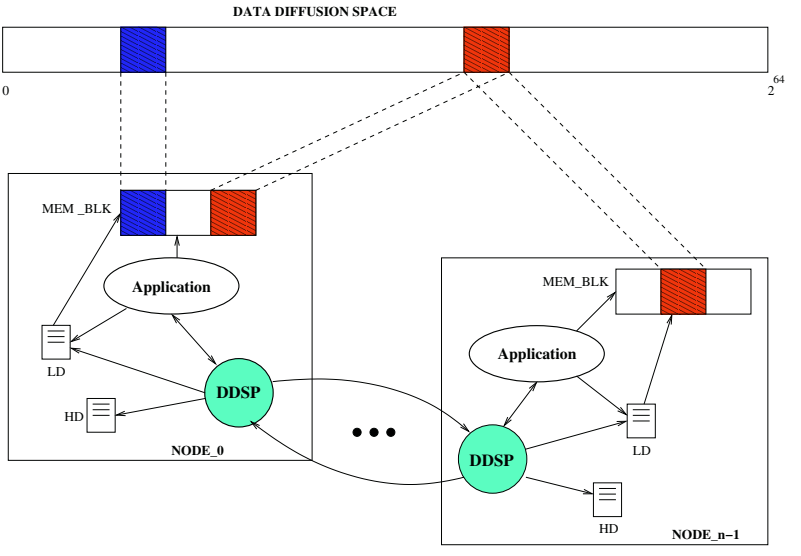

Fig. 1. DDS Architecture.

exclusive copy is obtained invalidating all other copies, thus ensuring all processors have the same view of the shared data.

The DDS protocol is similar to that of COMA-F (Cache-only Memory Architecture-Flat), an all-hardware distributed shared memory architecture [5]. It is homeless and directory-based. Data has no home location. It moves to the memory of the accessing processors and resides there, either until it is invalidated by a write by a processor or until it is evicted to give room to other data most recently used.

COMA-F uses associative main memory. Hence data has no home location therein. When a read or write misses in a memory node, a request is sent to the *home directory* of the relevant data item. This directory holds the location (node) and state information (exclusive, shared) of the item. If the home directory node is that location, it services the request; otherwise it sends the request to a node that currently has the item. A home directory is managed in each node, and some bits of each item address are used to identify a home directory.

DDS uses two directories in each node (see Figure 1). The *local directory* (LD) plays the role of an associative memory directory. A data item address is looked up there to see if the corresponding data item is in the memory. However, our local directory organisation keeps track of data not only in the memory of a node, but in both the memory and the disk space of the node. When a memory is needed to store recently used items, *exclusive* items less recently used are swapped out onto disk space. *Shared* items are just discarded.

When a read or write misses in a node, the DDS protocol sends a request to the relevant home directory, which is used and identified as described above for the COMA-F protocol.

3.3 Programming Model

Figure 2 shows the use DDS in the addition of two matrices: $C = A + B$. The programmer must define shared data within the *DDS* C structure. Before using shared data, the programmer must call *DDS_Init* as shown in that figure. In each processing node, *DDS_Init* maps the shared data to the diffusion space, initialises the local directory and the home directory, and starts the DDSP process.

In the matrix addition code, $ROWS/nprocs$ rows are calculated by each processor. Before accessing data, each processor must gain access to it, through calling *DDS_Write* or *DDS_Read*. When these procedures return, the relevant data is already in the processor memory, and will remain there until the corresponding *DDS_UnWrite* or *DDS_UnRead* is issued.

```
struct DDS {                                  /* declaring shared data */
  unsigned int A[ROWS][COLUMNS];
  unsigned int B[ROWS][COLUMNS];
  unsigned int C[ROWS][COLUMNS];
};
:
main() {
:
  DDS_Init(sizeof(struct DDS), &myid);        /* initialising DDS */
:
  rows = ROWS/nprocs;
  offset = myid * (ROWS/nprocs);
  for (r=0; r < rows; r++){
    i = r + offset;
    DDS_Write(DDS_C, i*COLUMNS, COLUMNS); /* gaining access */
    DDS_Read(DDS_A, i*COLUMNS, COLUMNS);  /* to shared data */
    DDS_Read(DDS_B, i*COLUMNS, COLUMNS);
    for (j=0; j<NCA; j++){                    /* using shared data */
      (dds_shmem[off_C+i])[j] = (dds_shmem[off_A+i])[j] +
                                (dds_shmem[off_B+i])[j];
    }
    DDS_UnWrite(DDS_C, i*COLUMNS, COLUMNS);
    DDS_UnRead(DDS_A, i*COLUMNS, COLUMNS);
    DDS_UnRead(DDS_B, i*COLUMNS, COLUMNS);
  }
:
```

Fig. 2. DDS programming model example: matrix addition.

DDS_A, *DDS_B* and *DDS_C* are *enumeration* constants 0, 1 and 2, respectively. They refer to the order in which arrays A, B and C were declared within the *DDS* structure. They are used at run time to index the array *dds_vars*, where, for each DDS variable/array, the size of each element, the total number

of elements and the initial (DDS) shared address are found. This information is used, along with the other two parameters sent to DDS_Write/DDS_Read, to calculate the DDS address of the data being accessed. The data is actually accessed through pointers held in the array dds_shmem, and the variables off_A, off_B and off_C, which are *locally* shared between the DDSP process and the application process. The variables off_A, ..., off_C (or that related with other defined shared data) are updated by DDSP according both to the address of the data requested with DDS_Read or DDS_Write, and to the actual location where that data is placed in the local memory, possibly after being requested from a remote node.

4 Performance Evaluation

To evaluate the performance of DDS we ran two applications on a 16-node PC cluster using different numbers of processors, both applications under DDS and under MPI/MPI-IO [13]. The version of MPI-IO we used is also known as ROMIO [18], and was used for our MPI version to be either in-core or out-of-core. In the out-of-core version, PVFS [14] is used and data is partitioned round robin into disk space along nodes by block (stripe in PVFS terminology). Each block is the size of n/p rows, where n is the number of rows in each array, and p is the number of processors used in each application run. Under DDS, out-of-core applications are programmed as in-core applications are. There is no need to specify a data partitioning into disk space.

It must be noted that, in programming out-of-core applications under MPI, programmers partition, or know the partition of, data into disk space. Also, programmers read data from, and write data to, disk space. That is, the programmer knows that data does not fit in memory in its entirety, and thus uses some memory only as a temporary buffer. The number of I/O requests is thus implicitly defined by the programmer. Under DDS, some reads and writes can be satisfied from copies in other memory nodes; hence the number of I/O requests can potentially be reduced.

The 16-node cluster configuration is as follows. Each node has 1 Intel Celeron 1.7 GHz processor, 512 MB RAM memory, and a hard disk drive. Hard disk drives are, however, of different make, size (1 GB, 3 GB, 4 GB and 8 GB) and speed. All nodes are interconnected by a 3COM Fast Ethernet switch with 48 ports. The operating system is Linux RedHat 9.0.

4.1 Matrix Multiplication

Our first application is a matrix multiplication (MM) algorithm: $C = A * B$. A and C are managed by rows (the C language default) and the matrix B by columns (the elements of a column are stored in consecutive localities in memory and/or in disk space). The matrices used were of size $16K \times 16K \times 8bytes$ (long type), or 2 GB each, a total of 6 GB for the three matrices. The matrices are partitioned into disk space such that each processor has $\lfloor \frac{n}{p} \rfloor$ consecutive columns

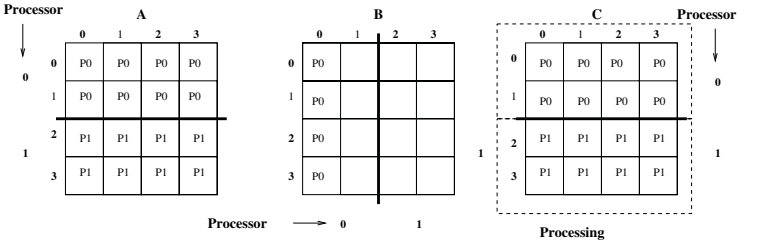

Fig. 3. Matrix multiplication: thick lines indicate data partitioning among nodes; dashed rectangles the elements in array C processed by each processor.

of B and $\lfloor \frac{n}{p} \rfloor$ rows of A and C (but C is only written). The multiplication is as follows. Each processor calculates the total value of each element in $\lfloor \frac{n}{p} \rfloor$ rows in matrix C. Each processor reads $\lfloor \frac{n}{p} \rfloor$ rows of A into memory, but only $\lfloor \frac{n}{p} \rfloor / f$ at a time, where $f = 1, 2$ and 4 for $p = 16, 8$ and 4, respectively. Then reads all columns of B, one at a time, f times, to calculate the value of all elements in n/p rows of C. Fig. 3 shows the data partitioning and processing for $n = 4$ and $p = 2$.

Table 1. MM: I/O requests under DDS and MPI-PVFS $16K * 16K$ matrices of 8-byte integers.

DDS			MPI-PVFS		
Processors	Reads	Writes	Processors	Reads	Writes
4	69632	4096	4	69632	4096
8	34816	2048	8	34816	2048
16	17408	1024	16	17408	1024

Table 1 shows the *average* ($total/p$) number of I/O *requests* under DDS and under MPI-PVFS. Under both, the number of I/O requests is the same on 4, 8 and 16 processors. This is somewhat surprising because it means that, under DDS, the columns of array B, which are the ones shared by all processors, were not diffused at all. The reason is as follows. In 4, 8 and 16 processors, each processor uses just above 256 MB of memory to store shared data. On the other hand, the amount of memory required by the rows of A and C that each processor holds in memory at any time is $\lfloor \frac{n}{p} \rfloor / f = (16384/4)/4 = 1024$ in all processor-count configurations (recall that for $p = 4$, $f = 4$, ... and for $p = 16$, $f = 1$). This is a total of $1024 \times 16384 \times 8$ (bytes) $= 128$ MB for each A and C. Since that many rows A and C are wired (with DDS_Read), there is very little memory for the columns of B to remain resident in main memory, and thus are evicted from main memory just after being used.

However, each processor uses all the columns of B in the same order, and thus once a column of B is resident in main memory, should it not be diffused

to other processors (thus reducing the amount of read requests)? This did not happen because disk drives in our platform are of different speed, and because we did not synchronise MM (both under DDS and MPI-PVFS) periodically. Since processors started reading their rows of A at different speed from different disks, they did not access columns in B concurrently at all.

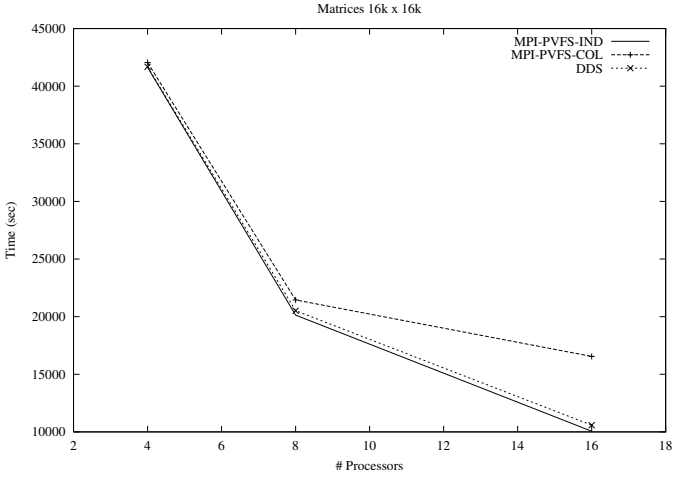

Fig. 4. MM: response time under DDS and MPI-PVFS.

Figure 4 shows the execution time of MM under DDS and MPI-PVFS, the latter both with independent I/O (MPI-PVFS-IND) and with collective I/O (MPI-PVFS-COL). DDS and MPI-PVFS-IND show almost the same performance in 4, 8 and 16 processors because they incur the same number of I/O operations and because these operations are independent in both versions. MPI-PVFS-COL also incurred the same number of I/O operations. However, synchronisation of collective operations, coupled with different speed of disk drives, increased response time.

4.2 Fast Fourier Transform

Our second application applies the Fast Fourier Transform (FFT) to restore degraded or defocused images. For an image of $N \times N$ pixels, a matrix of size $N \times N \times 8$ (float type) bytes is used. From this matrix, another matrix is created, which corresponds to an autocorrelation process of the original image that contains $M \times M$ images, where $M = 2N$ (see Figure 5). The size in bytes of this matrix is $2N \times 2N \times (N \times N) \times 8 = (N^4) \times 32$ bytes.

The image matrix is physically partitioned among processors by rows. In Figure 5, for $p = 4$, processor 0 (out of four) stores in disk space the images in the first row, processor 1 stores the images in the second row and so on.

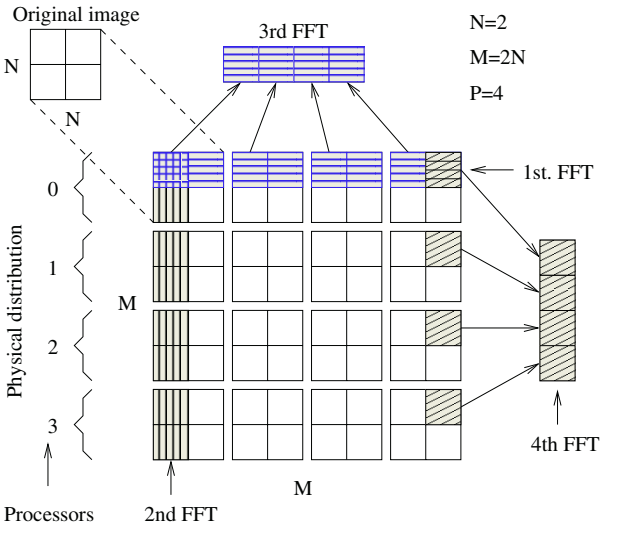

Fig. 5. FFT: data partitioning and processing of the images matrix.

Each processor applies the FFT to M/p rows and to M/p columns four times, as follows (see Figure 5): along entire rows (1st FFT), along entire columns (2nd FFT), jumping through rows (3rd FFT), and jumping through columns (4th FFT).

Table 2 shows the number of I/O requests per processor, both under MPI-PVFS and DDS on 4, 8 and 16 processors, for an original matrix of size 64×64 pixels. The *total* number of I/O requests is the same in all processor-count configurations. The images matrix is of size $((64)^4) \times 32 = 512$ MB, and could held in memory in all processor-count configurations.

Table 2. FFT: I/O requests under DDS and MPI-PVFS.

DDS			MPI-PVFS		
Processors	Reads	Writes	Processors	Reads	Writes
4	4096	4096	4	12288	12288
8	2048	2048	8	6144	6144
16	1024	1024	16	3072	3072

For each processor-count configuration, the number of I/O requests is fewer under DDS than under MPI-PVFS. Under DDS only the initial reads to load data into memory and the final writes to store results in disk space are incurred. Along the computation, other reads and writes are satisfied from copies in other memory nodes. There are more I/O requests under MPI-PVFS because each node manages only one memory buffer to hold an entire row of images at a time. As mentioned earlier, the application was programmed to manage both in-core

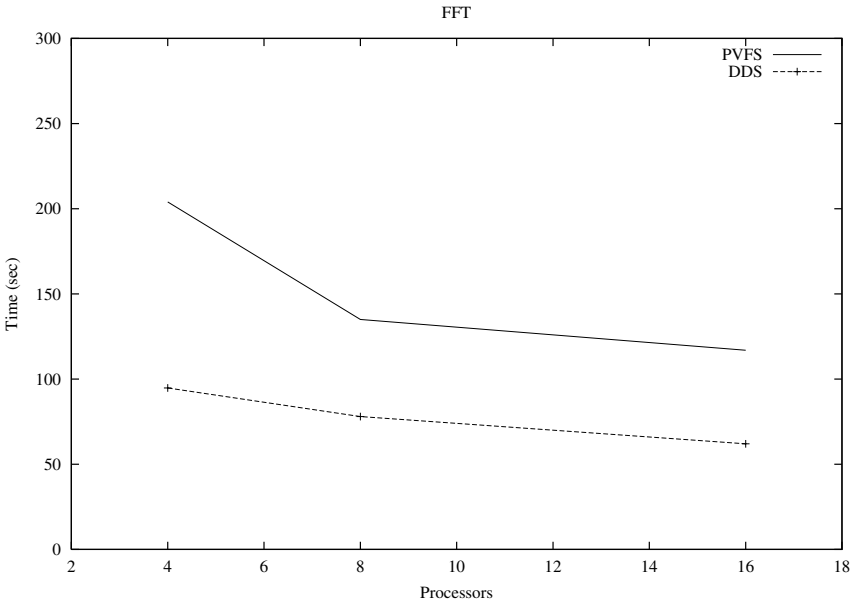

Fig. 6. FFT: response time under DDS and MPI-PVFS.

and out-of-core conditions (managing more buffers complicates programming even more).

Figure 6 shows execution of FFT under DDS and MPI-PVFS. In all processor-count configurations, DDS performs better than MPI-PVFS because it incurs fewer I/O overhead, reducing response time by half on average.

5 Conclusions and Future Work

We presented the data diffusion space (DDS), an extra shared address space for parallel computing under the SPMD model on distributed memory platforms. Compared with message passing, DDS is simpler to use and potentially offers improved performance both for in-core and out-of-core applications. On applications tested, DDS shows good performance up to 16 processors.

Programming a parallel application under DDS requires that *DDS_Read* and *DDS_UnRead*, or *DDS_Write* and *DDS_UnWrite*, functions be called to access data. DDS brings the data to the memory of the accessing processor whichever the current location of the data is, either other memory nodes or local or remote disk space.

We are currently designing a parallel file system with support to mapping files onto DDS. To support the shared memory programming model completely, we are also designing an extension to the C language and its compiler to avoid the use of the DDS interface entirely.

References

1. A. Agarwal *et* al. *The MIT Alewife Machine: Architecture and Performance.* In Proceedings of the 22nd ISCA (1995) 943–952.
2. H.E. Bal, M.F. Kaashoek and A.S. Tanenbaum. *ORCA: A Languaje for Parallel Programming of Distributed Systems.* IEEE Transactions on Software Engineering (March 1992) 190 – 205.
3. B.N. Bershad, M.J. Zekauskas and W. A. Sawdon. *The midway distributed shared memory system.* In Proceedings of COMPCON'93 (1993) 528–537.
4. J. Buenabad-Chávez, H.L. Muller, P.W.A. Stallard and D.H.D Warren. *Virtual memory on data diffusion architectures.* Parallel Computing 29 (2003) 1021–1052.
5. T. Joe. *COMA-F: A Non-hierarchical Cache Only Memory Architecture.* Stanford University Department of Electrical Engineering. PhD Thesis, 1995.
6. K.L. Johnson, M. F. Kaashoek, and D. A. Wallach. *CRL: High-Performance All-Software Distributed Shared Memory.* In Proceedings of the 5th SOSP (1995).
7. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. *The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor.* In Proceedings of the 17th ISCA (1990) 148–159.
8. K. Li. *Shared Virtual Memory Systems on Loosely Coupled Multiprocessors* (IVY). Yale University. PhD thesis,1986
9. J. Laudon and D. Lenoski. *The SGI Origin: A ccNUMA Highly Scalable Server.* In Proceedings of the 24th ISCA (1997) 241–251.
10. N. P. Manoj, K. V. Manjunath and R. Govindarajano. *CAS-DSM: A compiler assisted sofware distributed shared memory.* International Journal of Parallel Programming 32 (2004) 77–122.
11. M.D. Marino and G. Lino de Campos. *A speedup comparative study: three third generation DSM systems.* In Proceedings of the 7th International Conference on Parallel and Distributed Systems (2000) 153–158 (Workshops).
12. MPI: The Message Passing Interface. http://www-unix.mcs.anl.gov/mpi/
13. MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/docs.html
14. PVFS: The Parallel Virtual File System. http://parlweb.parl.clemson.edu/pvfs/
15. PVM: Parallel Virtual Machine. http://www.epm.ornl.gov/pvm/pvm_home.html
16. D.J. Scales, K. Gharachorloo and C.A. Thekkath. *Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory.* In Proceedings of the 7th ASPLOS (1996) 174–185.
17. M. Swanson, L. Stoller and J. Carter. *Making distributed shared memory simple, yet efficient.* In proceedings of the Third International Workshop on High-Level Parallel Programming Models and Supportive Environments (1998) 2–13.
18. R. Thakur, E. Lusk, and W. Gropp. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation.* Technical Report 234, Mathematics and Computer Science Division, Argonne National Laboratory, 1997.
19. D.H.D. Warren and S. Haridi. *DATA DIFFUSION MACHINE: A Scalable Shared Virtual Memory Multiprocessor.* In Proceedings of the International Conference on Fifth Generation Computer Systems (1988) 943–952.