Runtime Support for Parallel I/O in PASSION *

Rajesh Bordawekar *Rajeev Thakur* *Ravi Ponnusamy* *Alok Choudhary*

Northeast Parallel Architectures Center
111 College Place, Rm 3-228
Syracuse University, Syracuse NY 13244
rajesh, thakur, ravi, choudhar @npac.syr.edu

Corresponding Author: Rajesh Bordawekar
Tel: (315) 443-4061
Fax: (315) 443-1973

Abstract

We are developing a compiler and runtime support system called **PASSION**: **P**arallel **A**nd **S**calable **S**oftware for **I**nput-**O**utput, to translate out-of-core data-parallel programs to message passing node programs with explicit parallel I/O. This paper describes the design and implementation of the runtime system used in PASSION. We explain the basic model used by the compiler and runtime system. We describe the runtime routines for regular problems and provide an initial framework for runtime support for out-of-core irregular problems. Preliminary performance results are presented for 2D FFT and Laplace equation solver, and a kernel from a molecular dynamics code.

*This work was supported in part by NSF Young Investigator Award CCR-9357840, a grant from Intel SSD, and IBM, and in part by USRA CESDIS Contract # 5555-26. This work was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by CRPC.

1 Introduction

I/O for parallel systems has drawn increasing attention in the last few years as it has become apparent that I/O performance rather than CPU or communication performance may be the limiting factor in future computing systems. Large scale scientific computations, in addition to requiring a great deal of computational power, also deal with large quantities of data. At present, a typical Grand Challenge Application could require 1Gbyte to 4Tbytes of data per run [dRC94]. These figures are expected to increase by orders of magnitude as teraflop machines make their appearance. Although supercomputers have very large main memories, the memory is not large enough to hold this much amount of data. Hence, data needs to be stored on disk and the performance of the program depends on how fast the processors can access data from disks. Unfortunately, the performance of the I/O subsystems of MPPs has not kept pace with their processing and communications capabilities. A poor I/O capability can severely degrade the performance of the entire program. The need for high performance I/O is so significant that almost all the present generation parallel computers such as the Paragon, iPSC/860, Touchstone Delta, CM-5, SP-1, nCUBE2 etc. provide some kind of hardware and software support for parallel I/O [CFPB93, Pie89, DdR92].

At Syracuse University, we consider the I/O problem from a language, compiler and runtime support point of view. We are developing a compiler and runtime support system called **PASSION**: **P**arallel **A**nd **S**calable **S**oftware for **I**nput-**O**utput. PASSION provides support for compiling out-of-core programs [TBC94], parallel input-output of data, communication of out-of-core data, redistribution of data stored on disks, many optimizations including data prefetching from disks, data reuse etc. This paper describes the design and implementation of the runtime system used in PASSION. We first explain the basic model used by the compiler and runtime support system. The runtime routines have to take care of both parallel I/O and interprocessor communication. We describe the runtime library for regular problems along with some of the optimizations incorporated in the runtime library to reduce the amount of I/O. We have also developed an initial framework for runtime support for out-of-core irregular problems. Preliminary performance results on the Intel Touchstone Delta and Intel iPSC/860 are presented.

The rest of this paper is organized as follows. The basic model used by the compiler and runtime system is described in Section 2. Section 3 describes runtime system of PASSION. Performance results are discussed in Section 4 followed by a discussion of related work in Section 5 and Conclusions in Section 6.

2 Model for Out-of-Core Compilation

In the SPMD (Single Program Multiple Data) programming model, each processor has a local array associated with it. In an in-core program, the local array resides in the local memory of the processor. For large data sets, however, local arrays cannot entirely fit in main memory. In such cases, parts of the local array have to be stored on disk. We refer to such a local array as an *Out-of-core Local Array (OCLA)*. Parts of the OCLA need to be swapped between main memory and disk during the course of the computation.

The basic model of out-of-core compilation is shown in Figure 1. The simplest way to view this model is to think of each processor as having another level of memory which is much slower than main memory. Since the local arrays are out-of-core, they have to be stored in files on disk. The local array of each processor is stored in a separate file called the **Local Array File (LAF)** of that processor. The node program explicitly reads from and writes into the file when required. If the I/O architecture of the system is such that each processor has its own disk, the LAF of each processor will be stored on the disk attached to that processor. If there is a common set of disks for all processors, the LAF will be distributed across one or more of these disks. In other words, we assume that each processor has its own *logical disk* with the LAF stored on that disk. The mapping of the logical disk to the physical disks is system dependent. At any time, only a portion of the local array is fetched and stored in main memory. The size of this portion is specified at compile time and it usually depends on the amount of memory available. The portion of the local array which is in main memory is called the **In-Core Local Array (ICLA)**. All computations are performed on the data in the ICLA. Thus, during the course of the program, parts of the LAF are fetched into the ICLA, the new values are computed and the ICLA is stored back into appropriate locations in the LAF.

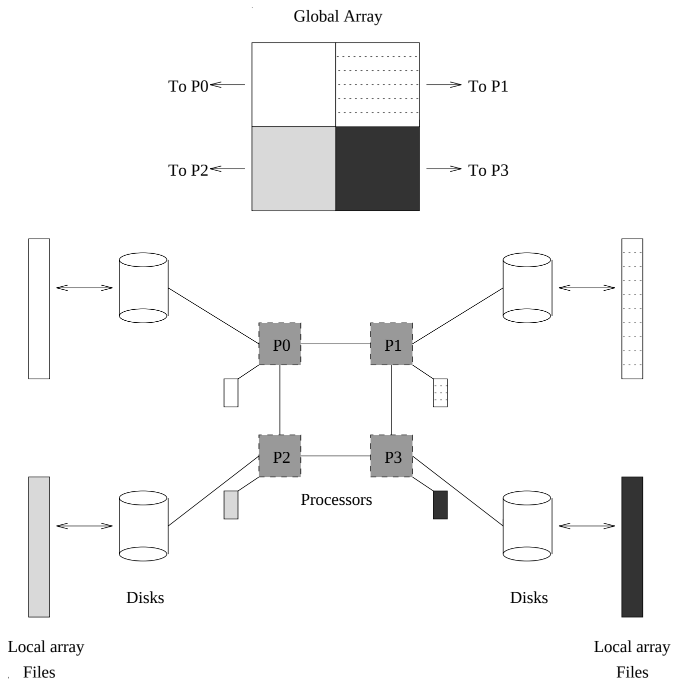

Figure 1: Model for Out-of-Core Compilation

3 Runtime Support in PASSION

During program execution, it is necessary to fetch data from the LAF into the ICLA and store the newly computed values from the ICLA back into appropriate locations in the LAF. Since the global array is distributed, a processor may need data from the local array of another processor. This requires data to be communicated between processors. Thus the node program needs to perform I/O as well as communication, both of which are not explicit in the source HPF program. The compiler does basic code transformations such as partitioning of data and computation, and inserts calls to runtime library routines for disk accesses and communication. The runtime support system for the compiler consists of a set of high level specialized routines for parallel I/O and collective communication. These routines are built using the native communication and I/O primitives of the system and provide a high level abstraction which avoids the inconvenience of working directly with the lower layers. For example, the routines hide details such as buffering, mapping of files on disks, location of data in files, synchronization, optimum message size for communication, best communication algorithms, communication scheduling, I/O scheduling etc.

3.1 Issues in Runtime Support

Consider the HPF program fragment given in Figure 2, which solves Laplace's equation by Jacobi iteration method. The arrays A and B are distributed as (block,block) on a 4×4 grid of processors as shown in Figure 3. Consider the out-of-core local array on processor P5, shown in Figure 3(B). The value of each element (i, j) of A is calculated using the values of its corresponding four neighbors in B, namely $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$. Thus to calculate the values at the four boundaries of the local array, P5 needs the last row of the local array of P1, the last column of the local array of P4, the first row of the local array of P9 and the first column of of the local array of P6. Before each iteration of the program, P5 gets these rows and columns from its neighboring processors. If the local array was in-core, these rows and columns would have been placed in the overlap areas shown in the Figure 3(B). This is done so as to obtain better performance by retaining the DO loop even at the boundary. Since the local array is out-of-core, these overlap areas are provided in the local array file. The local array file basically consists of the local array stored in either row-major or column major order. In either case, the local array file will consist of the local array elements interspersed with overlap area as shown in Figure 3(D). Data from the file is read into

```
parameter (n=1024)
real A(n,n), B(n,n)
..........
!HPF$ PROCESSORS P(4,4)
!HPF$ TEMPLATE T(n,n)
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P
!HPF$ ALIGN with T :: A, B
...........
    FORALL (i=2:n-1, j=2:n-1)
        A(i,j) = (B(i,j-1) + B(i,j+1) + B(i+1,j)
            + B(i-1,j))/4
...........
    B = A
```

Figure 2: HPF Program Fragment

the in-core local array and new values are computed. The in-core local array also needs overlap area for the same reason as for the out-of-core local array. The example shown in the figure assumes that the local array is stored in the file in column major order. Hence, for local computation, columns have to be fetched from disks and then written back to disks.

At the end of each iteration, processors need to exchange boundary data with neighboring processors. In the in-core case, this would be done using a shift type collective communication routine to directly communicate data from the local memory of the processors. In the out-of-core case, there are two options:-

- **Direct File Access**: Since disks are a shared resource, any processor can access any disk. In the direct file access method, a processor directly reads data from the local array file of some other processor as required by the communication pattern. This requires explicit synchronization at the end of each iteration.
- **Explicit Communication**: Each processor accesses only its own local array file. Data is read into memory and sent to other processors. Similarly, data is received from other processors into main memory and then saved on disk. This is similar to what would be done in in-core compilation methods.

Consider a situation in which each processor needs to communicate with every other processor (all-to-all communication). In the Direct File Access Method, this will result in several processors trying to simultaneously access the same disk, resulting in contention for the disk. A minimum of one block of data, the size of which is system dependent, is transferred during each disk access. Even if a processor actually needs a small amount of data, one whole block will be transferred for each access from every processor. So the Direct File Access Method has the drawback of greater disk contention and higher granularity of data transfer. Also, in some communication patterns (eg. broadcast), the same piece of data may be fetched repeatedly by several processors. In the Explicit Communication Method, each processor accesses only its own local file and reads the data to be sent to other processors into its local memory. This data is communicated to other processors. Thus, there is no contention for a disk and since the data to be sent to all other processors has to be read from disk, the high granularity of data access from disk is less of a problem. In addition, the time to communicate data between processors is at least an order of magnitude less than the time to fetch data from disk. However, this requires a communication phase in addition to I/O.

3.2 I/O Optimizations

3.2.1 Data Reuse

One way to reduce the amount of I/O is to reuse the data already fetched into main memory instead of reading it again from disk. This can be explained with the help of the Laplace equation solver program

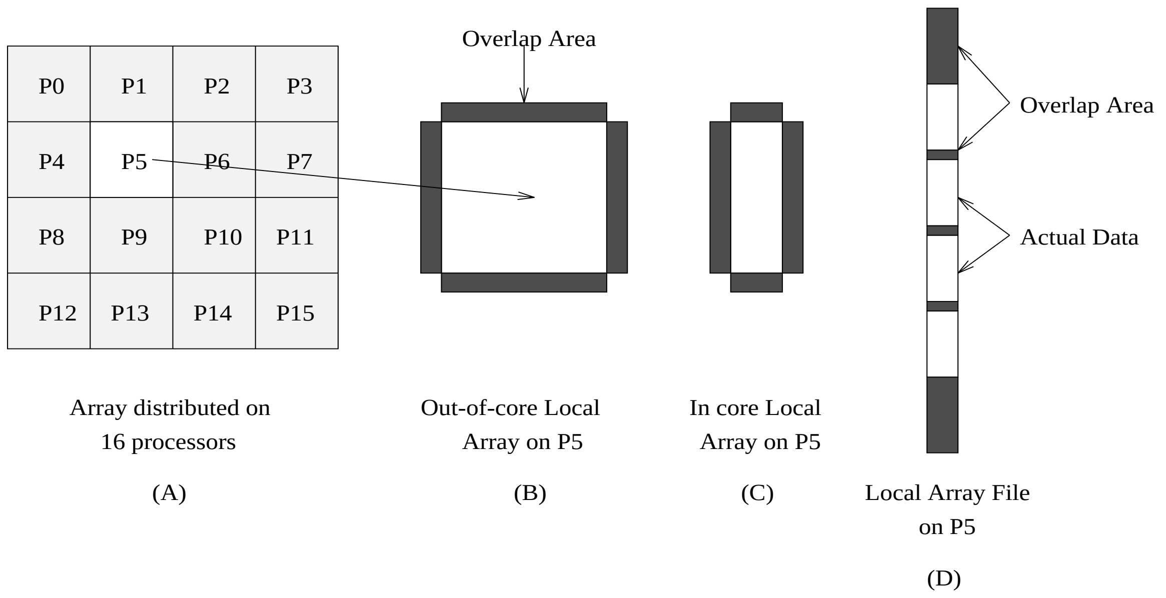

Figure 3: Example of OCLA, ICLA and LAF

discussed earlier. Suppose the array is distributed along columns. Then the computation of each column requires one column from the left and one column from the right. The computation of the last column requires one column from the overlap area and the computation of the column in the overlap area cannot be performed without reading the next column from the disk. Hence, instead of reading in the column in the overlap area again with the next set of columns, it can be reused by moving it to the first column of the array and the last column can be moved to the overlap area before the first column. If this move is not done, it would be required to read the two columns again from the disk along with data for the next slab. The reuse thus eliminates the reading of two columns in this example. In general, the amount of data reuse would depend on the intersection of the sets of data needed for computations involving two consecutive slabs.

3.2.2 Data Prefetching

Since portions of the OCLA are fetched one slab at a time, the time spent on I/O can be minimized by issuing a prefetch [Dav90, CKP91] for the next slab as soon as the current slab has been fetched. Thus the reading in of the next slab can be overlapped with the computation being performed on the current slab. Prefetching is done with an asynchronous I/O read operation. An example of how prefetching can be done in the Laplace Equation solver is shown in Figure 4.

3.3 PASSION Runtime Library

We are developing a library of runtime routines using which we can compile any general out-of-core HPF program. The routines are divided into four categories — Array Management/Access Routines, Communication Routines, Mapping Routines and Generic Routines. Some of the basic routines and their functions are listed in Table 1.

3.3.1 Array Management/Access Routines

These routines handle the movement of data between the in-core local array and the local array file. Any arbitrary regular section of the out-of-core local array can be read for an array stored in either row-major or column-major order. The information about the array such as its shape, size, distribution, storage format etc. is passed to the routines using a data structure called the Out-of-Core Array Descriptor (OCAD) [TBC94].

```
Call communication routine to perform overlap shift.
Call I/O routine to read the ICLA.
do l=1, k               ! k = no. of slabs
    if (l < k), Prefetch the next slab
    do j = l1, u1
        do i = l2, u2
            A(i,j)=((B(i,j-1)+B(i,j+1)+B(i-1,j)+B(i+1,j))/4)
        end do
    end do
    Call I/O routine to store the result.
    if (l < k), wait for the prefetched slab to reach main memory
end do
```

Figure 4: Prefetching in the Laplace equation solver

3.3.2 Communication Routines

The Communication Routines perform collective communication of data in the out-of-core local array. We use the Explicit Communication Method described earlier. The communication is done for the entire OCLA, i.e. all the off-processor data needed by the OCLA is fetched during the communication. This requires inter-processor communication as well as disk accesses.

3.3.3 Mapping Routines

The Mapping Routines perform data and processor/disk mappings. Data mapping routines include routines to generate local array files from a global file. These routines use the two-phase data access strategy [Jua92, BdRC93]. Disk mapping routines map physical disks onto logical disks.

3.3.4 Generic Routines

The Generic Routines perform computations on out-of-core arrays. Examples of these routines are out-of-core transpose and out-of-core matrix multiplication.

3.4 Runtime Support for Out-of-Core Unstructured Problems

Unstructured or irregular problems [Pon94] are an important subclass of scientific applications. In irregular problems, patterns of data access cannot be predicted until runtime. In such problems, optimizations that can be carried out at compile-time are limited. At runtime, however, the data access patterns of a loop-nest are usually known before entering the loop-nest. This makes it possible to utilize various preprocessing strategies to optimize the computation. These preprocessing strategies primarily deal with reducing data movement between processor memories. Preprocessing methods for in-core computations have been developed [MSS$^+$88, Pon94] for a variety of unstructured problems including explicit multi-grid unstructured computational fluid dynamic solvers [Mav91, HB91], molecular dynamics codes (CHARMM, AMBER, GROMOS, etc.) [BBO$^+$83], and diagonal or polynomial preconditioned iterative linear solvers [VSM90].

Figure 5 illustrates a typical irregular loop. The data access pattern is determined by the array **inter_list**, which is known only at runtime. This array is called an *indirection array.* Prior knowledge of loop data access patterns (values of **inter_list**) makes it possible to predict which data elements need to be communicated between processors. A runtime pre-processing phase [MSS$^+$88] can be used to carry out optimizations. By pre-processing data access patterns, optimizations such as *software caching* and *communication vectorization* [DMS$^+$94] can be performed.

We have developed runtime routines in PASSION to efficiently implement out-of-core irregular computations on distributed memory machines. Due to space constraints, we only describe the case where the indirection arrays are out-of-core but the data arrays are in-core. Such a situation actually appears in irregular problems like molecular dynamics codes, where the size of data arrays associated with particles (e.g.

	Array Management Routines	
	PASSION Routine	Function
1	**read_section**	Reads a regular section from LAF to ICLA
2	**write_section**	Writes a regular section from ICLA to LAF
3	**read_with_reuse**	read_section with data reuse
4	**prefetch_read**	Asynchronous (non-blocking) read of a regular section
5	**prefetch_wait**	Wait for a prefetch to complete
	Array Communication Routines	
	PASSION Routine	Function
6	**oc_shift**	Shift type collective communication on out-of-core data
7	**oc_multicast**	Multicast communication on out-of-core data
	Array Mapping Routines	
	PASSION Routine	Function
8	**oc_disk_map**	Maps disks to processors
9	**oc_file_map**	Generates local files from global files
	Generic Routines	
	PASSION Routine	Function
10	**oc_transpose**	Transposes an out-of-core array
11	**oc_matmul**	Performs out-of-core matrix multiplication

Table 1: Some of the PASSION Runtime Routines

```
        real    x(n_atom), dx(n_atom)                ! data arrays
        integer inter_ptr(niter)                     !
        integer inter_list(niter)                    ! indirection array

L1:     do i = 1, n_atom                             ! outer loop
L2:       do j = inter_ptr(i), inter_ptr(i+1)-1      ! inner loop
            dx(inter_list(i)) = dx(inter_list(i)) + x(i)
          end do
        end do
```

Figure 5: An Example with an Irregular Loop

atoms) is considerably small compared to the indirection list (indirection array). Since interactions among particles in a problem size of O(n) can be O(n^2), it is sufficient to consider only the indirection arrays to be out-of-core. For instance, in one of the test cases presented in this paper, the size of data arrays is around 14K, however, the indirection array of size six million [Pon94]. The out-of-core pre-processing phase analyzes data access pattern and computes a communication schedule [MSS+88]. The computed communication schedule can be used to exchange in-core data items among processors.

3.4.1 Implementation Details

We describe implementation details of a non-bonded force calculation kernel of a molecular dynamics code. The kernel is similar to the loop shown in Figure 5. To successfully complete the computation associated with an atom, the interaction among neighboring atoms must be known. In this case, interactions among atoms are stored in an indirection array. Since indirection arrays are very large, the computation has to be done in phases. This allows the indirection array elements fetched from disk to be stored in-core.

The execution of the kernel is done in three phases: 1) Data Distribution, 2) Pre-processing, and 3) Computation. In order to balance load on processors, the atoms are distributed in round-robin fashion over the processors. After data distribution, the indirection arrays are analyzed. The pre-processing phase

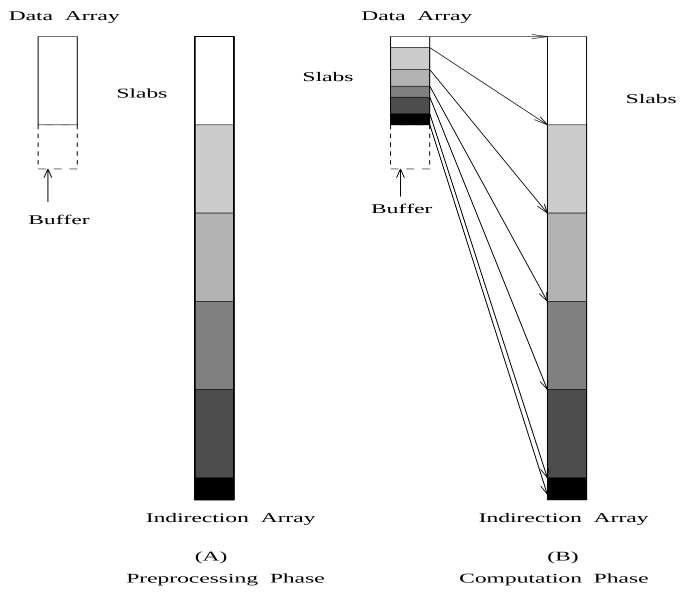

Figure 6: Data and Indirection Arrays

involves the fetching of indirection arrays in slabs into processor memory and operating on them. Pre-processing involves 1) address translation which uses a lookup table [DMS$^+$94] and 2) determination of communication patterns for in-core data arrays. The pre-processing phase optimizes communication by using techniques similar to those used in PARTI/CHAOS runtime library [DMS$^+$94, Pon94]. The third phase carries out computation and communication.

Figure 6 illustrates the stripmining of out-of-core arrays in pre-processing and computation phases. Figure 6(a) shows the stripmining of the indirection array in the preprocessing phase. Each indirection array slab is represented using a different shade. As described before, each slab is brought into memory and analyzed for the data access pattern. On each processor, additional space is allocated for each data array at the end of the local portion to store off-processor elements. Figure 6(b) shows the stripmining of data and indirection arrays in the computation phase. Each slab of the data array and the corresponding slabs of indirection arrays are marked using the same shade. Since the interaction list size of atoms is not the same, the size of the data array slab varies according to the indirection array slab. Here, the indirection array is preprocessed to ensure more efficient utilization of memory by choosing the size of the data slab so that the corresponding indirection array can be stored in memory. It should be noted that even if the data array is in-core, the computation on the data array has to be stripmined so that the corresponding indirection array slab can be in-core.

4 Performance

We present preliminary performance results for both structured an unstructured problems on the Intel Touchstone Delta and iPSC/860.

4.1 Structured Problems

As examples of structured problems, we consider the Laplace equation solver described earlier and also a 2D FFT code. The performance of the Laplace equation solver on the Intel Touchstone Delta is shown in Table 2. We use Intel's Concurrent File System (CFS) [Pie89] on the Delta which has 64 disks. The table compares the performance of the three methods — direct file access, explicit communication and explicit communication with data reuse. The array is distributed in one dimension along columns. We observe that the direct file access method performs the worst because of contention for disks. The best performance is obtained for the explicit communication method with data reuse as it reduces the amount of I/O by reusing

Table 2: Performance of Laplace Equation Solver (time in sec. for 10 iterations)

	Array Size: $2K \times 2K$		Array Size: $4K \times 4K$	
	32 Procs	64 Procs	32 Procs	64 Procs
Direct File Access	73.45	79.12	265.2	280.8
Explicit Communication	68.84	75.12	259.2	274.7
Explicit Communication with data reuse	62.11	71.71	253.1	269.1

Table 3: Out-of-core 2D-FFT for 4K $\times$ 4K array (time in sec., Slab_size as a fraction of local array size)

Processors	Slab_size	Slab_size	Slab_size	Slab_size
	1/16	1/8	1/4	1/2
16	616.84	562.01	502.71	484.73
32	596.54	537.89	499.78	456.35
64	610.91	589.88	517.34	478.72

data already fetched into memory. If the array is distributed in both dimensions, the performance of the direct file access method is expected to be worse because in this case each processor, except at the boundary, has four neighbors. So, there will be four processors contending for a disk when they try to read the boundary values.

Table 3 presents the performance of the two-dimensional Fast Fourier Transform on the Intel Touchstone Delta. The Fast Fourier Transform routine uses a PASSION routine to perform out-of-core transpose. The array is initially distributed in one dimension along columns. In the first phase, the program performs in-core FFT on the array slabs. Then the out-of-core array is transposed and the in-core FFT is again performed on the slabs of the transposed array. This example was run on 16, 32 and 64 processors. The slab_size was varied from 1/16 to 1/2 of the local array size. As the slab_size is increased, the performance tends to improve because the number of I/O requests reduces. In all cases, however, the time taken is dominated by I/O. As the number of processors is increased to 64, the parallel file system is saturated with requests from all the processors, and therefore, the performance tends to degrade.

4.2 Unstructured Problems

We studied the performance of the out-of-core unstructured molecular dynamics code on an Intel iPSC/860 with 16 processing nodes, 2 I/O nodes and 4 disks. The iPSC Concurrent File System (CFS) stripes data uniformly over disks. User can specify the number of disks used for data striping. By default, data is striped over all the available disks (in our case, 4). We present the performance of a molecular dynamics (MD) kernel for 14K and 6K atoms cases. For this kernel to be executed in-core, atleast 32 processors are required, since each processor on this system has 8MBytes of memory. As our system has only 16 processors, we ran the program by keeping the indirection arrays out-of-core and the data array in-core.

Table 4 shows pre-processing times for several slab sizes for 14K atoms. This experiment was performed by varying the number of processors. It can be observed that as the available memory increases, the performance is improved. In addition, as the number of processors is increased, better performance is observed. The best performance for is observed for 8 processors with slab size of 64K (92.16 sec). Table 5 shows pre-processing times for several slab sizes for 6K atoms. The best performance is observed for 8 processors with slab_size of 64K (72.79 sec). Each iteration requires reading a indirection array slab from disk and writing the modified indirection array slab back to the disk. As a result, each iteration requires two I/O accesses. Since the file is distributed over all 4 disks, each I/O access results in 4 disks accesses. As the number of processors increases, the number of disk accesses also increases saturating the I/O subsystem. This results in performance degradation for a large number of processors.

Table 4: Pre-processing Time in sec. for MD Kernel for 14K Atom Case

Processors	Slab_size				
	8 K	16K	32K	64K	128K
2	194.61	190.81	188.85	187.83	190.95
4	134.79	129.84	127.54	130.27	138.45
8	115.35	99.68	94.62	92.16	98.54
16	127.22	109.77	107.81	111.73	107.25

Table 5: Pre-processing Time in sec. for MD Kernel for 6K Atoms Case

Processors	Slab_size				
	8 K	16K	32K	64K	128K
2	135.79	117.10	113.48	110.98	117.56
4	89.95	85.74	86.33	85.14	85.59
8	82.36	78.60	75.31	72.79	73.88
16	162.62	128.52	111.88	108.21	109.92

Tables 6 and 7 show the computation times for MD kernel for various number of processors and data array slab sizes. The computation time is the time taken to execute the loop once. Typically in a problem this loop will be executed many times. The computation time also includes the communication time to gather and scatter off-processor data. The optimal performance for 14K case is observed for 8 processors (computation time = 20.69 sec). For 6K atoms case, the optimal performance is observed for 4 processors (computation time = 26.88 sec). It can be observed that, in general, as the size of the data slab is increased, performance improves. This is due to reduction in the number of I/O requests. However, the computational cost is always dominated by I/O cost. Computation on each in-core slab requires reading the corresponding indirection array slab from disks. As explained earlier, computation stripmining is performed so that indirection array could be distributed into slabs that can be in-core. Since the number of interactions depends on individual atom, the number of atoms that correspond to a fixed size of indirection array varies. As the computation is a sweep over atoms, the number of I/O requests is a function of the number of data(atoms) slabs. Hence as the number of data slabs increases, the number of I/O requests increases. For eight and sixteen processors, the concurrent file system gets saturated with a large number of I/O requests (and corresponding disk requests), and therefore the performance of computation phase degrades.

Due to space limitations, a detailed discussion of the performance results is not possible in this version of the paper. It should be noted that for both of these problem sizes, an in-core implementation on a system containing upto 16 processors was not possible due to memory size limitation. Further improvements in the performance may be obtained if the file system provides a greater control over stripe size, number of disks on which a file can be opened and prefetching.

Table 6: Computation Time in sec. of the MD Kernel for 14K Atoms Case

Processors	Slab Size				
	8K	16K	32K	64K	128K
2	40.20	39.63	38.98	38.89	40.62
4	26.24	24.83	23.01	22.23	23.42
8	43.54	41.58	34.57	30.52	26.88
16	81.14	78.41	94.38	76.80	75.48

Table 7: Computation Time in sec. of the MD Kernel for 6K Atoms Case

Processors	Slab Size				
	8K	16K	32K	64K	128K
2	41.62	45.05	39.98	38.86	42.55
4	26.22	25.00	23.29	22.40	24.25
8	42.19	39.00	34.15	31.23	28.12
16	95.26	88.44	85.96	82.63	83.66

5 Related Work

There has been some related research in software support for high performance parallel I/O. Vesta is a parallel file system designed and developed at IBM T. J. Watson Research Center [CFPB93, CF94]. Language support for out-of-core data parallel programs are proposed in [BGMZ92, Sni92]. File declustering, where different blocks are stored on distinct disks is suggested in [LKB87]. This is used in the Bridge File System [DSE88], in Intel's Concurrent File System (CFS) [Pie89] and in various RAID schemes [PGK88]. The effects of prefetching blocks of a file in a multiprocessor file system are studied in [Dav90]. Software prefetching for in-core problems is discussed in [MLG92, CKP91]. PARTI/CHAOS runtime support for in-core irregular computations has been developed by Saltz and co-workers [MSS$^+$88, DMS$^+$94, Pon94].

6 Conclusions

We have presented the design and implementation of some of the PASSION runtime procedures for out-of-core regular problems and an initial framework for out-of-core irregular problems. Preliminary performance results are discussed for hand-coded versions of a Laplace Equation solver, 2D FFT and a kernel from a molecular dynamics code. The PASSION runtime procedures perform

- Read and write to disk during computation.
- Interprocessor communication involving out-of-core data.
- Reuse of out-of-core data.
- Prefetching of out-of-core data.
- Pre-processing for out-of-core irregular computations.
- Scheduling communication for out-of-core data.
- Partitioning for out-of-core data.

The PASSION procedures can be automatically embedded by a High Performance Fortran compiler to carry out parallel I/O. We are currently developing PASSION compiler transformation techniques to embed runtime procedures.

Acknowledgements

We thank Geoffrey Fox, Ken Kennedy, Chuck Koelbel and Joel Saltz for many fruitful discussions and helpful comments. We also thank Joel Saltz for providing the CHAOS library. We thank Ravikumar Muppirala of the Chemistry department at Syracuse University for providing us protein molecule strucutres used in this paper.

References

[BBO+83] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamic s calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[BdRC93] R. Bordawekar, J. del Rosario, and A. Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Proceedings of Supercomputing'93*, pages 452–461, November 1993.

[BGMZ92] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima. Concurrent File Operations in a High Performance Fortran. In *Proceedings of Supercomputing '92*, pages 230–238, November 1992.

[CF94] P. Corbett and D. Feitelson. Overview of the Vesta Parallel File System. In *Proceedings of the Scalable High Performance Computing Conference*, pages 63–70, May 1994.

[CFPB93] P. Corbett, D. Feitelson, J. Prost, and S. Baylor. Parallel Access to Files in the Vesta File System. In *Proceedings of Supercomputing '93*, pages 472–481, November 1993.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of ASPLOS 91*, pages 40–52, 1991.

[Dav90] David Kotz and Carla Ellis. Prefetching in File Systems for MIMD Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 218–230, April 1990.

[DdR92] E. DeBenedictis and J. del Rosario. nCUBE Parallel I/O Software. In *Proceedings of* 11^{th} *International Phoenix Conference on Computers and Communications*, pages 117–124, April 1992.

[DMS+94] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The Design and Implementation of a Parallel Unstructured Euler Solver Using Software Primitives. *AIAA Journal*, 32(3):489–496, March 1994.

[dRC94] J. del Rosario and A. Choudhary. High Performance I/O for Parallel Computers: Problems and Prospects. *To appear in IEEE Computer*, 1994.

[DSE88] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A High-Performance File System for Parallel Processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[HB91] S. Hammond and T. Barth. An Optimal Massively Parallel Euler Solver for Unstructured Grids. *AIAA Journal, AIAA Paper 91-0441*, January 1991.

[Jua92] Juan Miguel del Rosario, Rajesh Bordawekar and Alok Choudhary. A Two-Phase Strategy for Achieving High-Performance Parallel I/O. Technical Report SCCS-408, NPAC, Syracuse University, October 1992.

[LKB87] M. Livny, S. Khoshafian, and H. Boral. Multi-Disk Management Algorithms. In *Proceedings of the 1987 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 69–77, May 1987.

[Mav91] D. J. Mavriplis. Three Dimensional Unstructured Multigrid for the Euler Equations, paper 91-1549CP. In *AIAA 10th Computational Fluid Dynamics Conference*, June 1991.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. *Architectural Support for Programming Languages and Operating Systems*, pages 62–73, 1992.

[MSS+88] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principles of Runtime Support for Parallel Processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, July 1988.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[Pie89] P. Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proceedings of* 4^{th} *Conference on Hypercubes, Concurrent Computers and Applications*, pages 155–160, Match 1989.

[Pon94] Ravi Ponnusamy. *Runtime Support and Compilation Methods for Irregular Computations on Distributed Memory Parallel Machines.* PhD thesis, Department of Computer Science, Syracuse University, Syracuse, NY, May 1994.

[Sni92] Marc Snir. Proposal for IO. Posted to HPFF I/O Forum by Marc Snir, July 7 1992.

[TBC94] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proceedings of the* 8^{th} *ACM International Conference on Supercomputing*, July 1994.

[VSM90] P. Venkatkrishnan, J. Saltz, and D. Mavriplis. Parallel Preconditioned Iterative Methods for the Compressible Navier Stokes Equations. In *12th International Conference on Numerical Methods in Fluid Dynamics, Oxford, England*, July 1990.